Das Assemblerbuch

http://kickme.to/tiger/

Das Assembler-Buch

Professionelle Programmierung

Trutz Eyke Podschun

Das Assembler-Buch

Grundlagen und Hochsprachenoptimierung

4., aktualisierte Auflage

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Bonn • Reading, Massachusetts • Menlo Park, California
New York • Harlow, England • Don Mills, Ontario
Sydney • Mexico City • Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Das Assembler-Buch ; Grundlagen und Hochsprachenoptimierung [Medienkombination] /
Trutz Eyke Podschun. – Bonn : Addison-Wesley-Longman, 1999
(Professionelle Programmierung)
ISBN 3-8273-1513-1

Buch. – 4., aktualisierte Auflage – 1999
Gb.

CD-ROM. – 4., aktualisierte Auflage – 1999

Lektorat: Susanne Spitzer und Friederike Daenecke
Satz: Reemers EDV-Satz, Krefeld. Gesetzt aus der Palatino 9,5 Punkt
Belichtung, Druck und Bindung: Kösel GmbH, Kempten
Produktion: TYP*isch* Müller, München
Umschlaggestaltung: viervierteil gestaltung, Köln

Das verwendete Papier ist aus chlorfrei gebleichten Rohstoffen hergestellt und alterungsbeständig. Die Produktion erfolgt mit Hilfe umweltschonender Technologien und unter strengsten Auflagen in einem geschlossenen Wasserkreislauf unter Wiederverwertung unbedruckter, zurückgeführter Papiere.

Inhalt

Vorwort zur vierten Auflage xiii

Vorwort zur dritten Auflage xiv

Vorwort zur zweiten Auflage xvi

Vorwort xvii

Einleitung xxi

Teil I
Grundlagen 1

1 Die Register des 8086 3
1.1 Der Adreßraum des 8086 6
1.2 Ports 12

2 Der Befehlssatz des 8086 16
2.1 Ein- und Ausgabeoperationen 18
2.2 Vergleichsoperationen 25
2.3 Sprungbefehle 32
2.4 Flags 40
2.5 Bitschiebeoperationen 41
2.6 Logische Operationen 46
2.7 Arithmetische Operationen 50
2.8 Stringoperationen 64
2.9 Korrekturoperationen 71
2.10 Sonstige Operationen 78

3 Die Register des 8087 80

4 Der Befehlssatz des 8087 87
4.1 Arithmetische Operationen mit Realzahlen 89
4.2 Weitere arithmetische Operationen mit Realzahlen 98
4.3 Arithmetische Operationen mit Integerzahlen 105
4.4 Weitere Operationen mit Zahlen 108
4.5 Sonderfall: BCDs 108
4.6 Rechenstackmanipulationen 109
4.7 Allgemeine Coprozessorbefehle 111

5 Zusammenarbeit zwischen 8086 und 8087 115

6 Änderungen beim 80186/80188 120

7 Der 80187 122

8 Änderungen beim 80286/80287 123

9 Zusammenarbeit zwischen 80286 und 80287 127

10 Änderungen beim 80386/80387 128

11 Der Adreßraum des 80386 140

12 Änderungen beim 80486 143

13 Zusammenarbeit zwischen dem 80486 und der Floating-Point-Unit 146

14 Änderungen beim Pentium 146

15 Änderungen beim Pentium Pro 148

16 Änderungen beim Pentium II 150

17 Die MMX-Technologie 151

17.1 Neue MMX-Datenformate 152
17.2 Die »neuen« MMX-Register 153
17.3 »Herumwickeln« und »Sättigen« 155
17.4 Neue MMX-Befehle 157
17.5 Beispiele für die Nutzung von MMX-Befehlen 170
17.6 MMX und die Floating-Point Unit 173

18 Windows 9x und Windows NT 176

18.1 Die Berechnung von Adressen Teil 1 – Theorie: Von der effektiven zur physikalischen Adresse 179
18.1.1 Die logische Adresse 179
18.1.2 Segmente und ihre Deskriptoren 180
18.1.3 Die globale Deskriptortabelle und ihr Register 184
18.1.4 Segmentselektoren und ihre Register 186
18.1.5 Lokale Deskriptortabellen und deren Register 188
18.1.6 Die virtuelle Adresse 189
18.1.7 Die Page – Abbildung der kleinsten Einheit des segmentierten physikalischen Speichers 190

18.1.8 Die Page Table 194
18.1.9 Das Page Table Directory und sein Register 196
18.1.10 Die physikalische Adresse 196
18.1.11 Der Paging-Mechanismus – ein Beispiel 197
18.2 Die Berechnung von Adressen Teil 2 – Praxis: Adressen aus der Sicht des Assemblerprogrammierers 202
18.2.1 Präfix und OpCode 202
18.2.2 Das MOD-Byte in 16-Bit-Umgebungen 207
18.2.3 Die MOD- und SIB-Bytes in 32-Bit-Umgebungen 208
18.3 Die Berechnung von Adressen Teil 3 – praktische Beispiele 210

19 Multitasking 214

19.1.1 Der Task 215
19.1.2 Der Task-State 216
19.1.3 Das Task-State-Segment 217
19.1.4 Der Task-State-Segmentdeskriptor 219
19.1.5 Das Task-Register des Prozessors 220
19.1.6 Task Switches 220
19.2 Gates 223
19.2.1 Tore zu fremdem Terrain 223
19.2.2 Gate-Deskriptoren 224
19.2.3 Das rechte Tor zur rechten Zeit 226
19.3 Interrupts und Exceptions 228
19.3.1 Interrupts 229
19.3.2 Exceptions 230
19.3.3 Die Interrupt-Deskriptortabelle (IDT) 230
19.3.4 Das Interrupt-Deskriptortabellenregister des Prozessors 232
19.3.5 Nutzung von Interrupt und Exceptions 232
19.4 Schutzmechanismen 233
19.4.1 Die Prüfung der Segmentgrößen 233
19.4.2 Die Prüfung der Segmenttypen 235
19.4.3 Zugriffsprivilegien 235
19.4.4 Schutzmechanismen auf Page-Ebene 240
19.4.5 Kommunikation mit der Peripherie: Die Ports 241
19.4.6 Weitere Schutzmechanismen 245

Teil 2
Arbeiten mit dem Assembler 247

20 Hallo, Welt! 249

20.1 Die Segmente eines Assemblerprogramms 252
20.2 Das erste Programm 263
20.3 Eine nicht ganz unwichtige Assembleranweisung 265
20.4 Nachtrag 267

21 Hallo, Coprozessor 270
21.1 Unser zweites Programm 270
21.2 Das eigentliche Programm 273

22 Wie heißt Du, Coprozessor? 279
22.1 Die Strategie 279
22.2 Ein Stapel Worte 281
22.3 Bürokratie! 283
22.4 Unterprogramme 285
22.5 Ein weiterer Stapel Worte 286
22.6 Coprozessorunterscheidung 293
22.7 Function or not function – that is a decision! 298
22.8 Kombination der Routinen zum Programm 307
22.9 Die Geschwindigkeit des Coprozessors 311

23 Makros 318

24 Namen für Werte: EQU 325

25 Assembler und Hochsprachen 327
25.1 Sind Sie im Bilde? Aber fallen Sie bloß nicht aus dem Rahmen! 329
25.2 Parameter für Routinen 334

26 Assembler und Pascal 336
26.1 Parameter in Pascal 336
26.2 Funktionswerte in Pascal 345
26.3 Einbindung von Assemblermodulen in Pascal-Programme 350
26.4 Wie arbeitet der Compiler? 357

27 Assembler und C 359
27.1 Parameter in C 359
27.2 Funktionswerte in C 366
27.3 Parameter und Funktionen in C, Teil 2 369
27.4 Einbinden von Assemblermodulen in C-Programme 374
27.5 Wie arbeitet der Compiler? 380

28 Assembler und Delphi 381

29 Der integrierte Assembler 398

30 Optimierungen beim Programmieren 410
30.1 »Optimierte« Compiler oder »optimiertes Denken«? 410
30.2 Optimieren mit dem Assembler 417
30.3 Noch ein Beispiel 427

30.4 Spezielle Optimierungen 431
30.5 Optimierungen bei 32-Bit-Betriebssystemen und -Prozessoren 433
30.6 Generelle Tips zum Optimieren 435

31 8087-Emulation 437

32 Strukturen 444
32.1 STRUC 444
32.2 UNION 448
32.3 Verschachtelte Strukturen 450
32.4 Andere Strukturen 451
32.5 Assembler und OOP 451

33 Vereinfachungen und Ergänzungen 451
33.1 Speichermodelle 451
33.2 Vereinfachte Segmentanweisungen 453
33.3 Sprachenvereinbarung 455
33.4 Argumentanweisungen 456
33.5 Lokale Variablen 458
33.6 Register retten 459
33.7 Automatische Sprungzielanpassung 461
33.8 Lokale Sprungziele 462
33.9 IDEAL, INVOKE usw. 465

34 Tips, Tricks und Anmerkungen 466
34.1 CMPXCHG, und wie die Probleme gelöst werden können 466
34.2 CPUID 472
34.3 Optimierung für Fortgeschrittene 473
34.4 Daten im Assemblermodul 484

Teil 3
Referenz 487

35 Prozessorbefehle 489

36 Coprozessorbefehle 685

37 MMX-Befehle 786

38 Condition Codes 814
38.1 Condition Codes bei Vergleichsbefehlen 814
38.2 Condition Codes bei Restbildung 814
38.3 Condition Codes nach FXAM 815

39 Exceptions 816
39.1 Exception-Klassen 816
39.2 CPU-Exceptions 818
39.3 FPU-Exceptions 835

40 Darstellung von Zahlen im Coprozessor- und MMX-Format 847
40.1 Internes Format und TEMPREAL-Format (FPU) 847
40.2 LONGREAL-Format (FPU) 847
40.3 SHORTREAL-Format (FPU) 848
40.4 LONGINT-Format (FPU) 849
40.5 SHORTINT-Format (FPU) 849
40.6 WORDINT-Format (FPU) 850
40.7 BYTEINT-Format (FPU) 850
40.8 BCD-Format (FPU) 850
40.9 PACKED BYTES-Format (MMX) 851
40.10 PACKED WORDS-Format (MMX) 851
40.11 PACKED DOUBLEWORDS-Format (MMX) 851
40.12 QUADWORD-Format (MMX) 852
40.13 Normale, nicht normale und unnormale Zahlen 852
40.14 Null und NANs 854
40.15 Unendlichkeiten 855
40.16 Zusammenfassung 855

41 Assembleranweisungen 858

Anhang 899

A Die verschiedenen Zahlensysteme 899
A.1 Das Dezimalsystem 899
A.2 Das Binärsystem 899
A.3 Das Hexadezimalsystem 901
A.4 Rechnen mit Binärzahlen 903
A.5 Negative Zahlen 904

B Interrupts 905

C tpASM 907
C.1 Der tpASM-Editor 907
C.2 Das Menü von tpASM 908

D Glossar 910

E Die Routinen der Routinensammlung Mathe 920
E.1 Neue Typen der Bibliothek Mathe 921
E.2 Die Routinen der Bibliothek Mathe 923
E.2.1 Turbo Pascal 923

E.2.2 C 925
E.3 Die Fehlerbearbeitung von Mathe 929

F Tabelle der Assemblerbefehle 931

G Die Register und Flags der Prozessoren 941
G.1 CPUID 941
G.2 Die Allzweckregister des Prozessors 943
G.3 Die Allzweckregister des Prozessors mit besonderen Funktionen 944
G.4 Das Flagregister (E)Flags des Prozessors 944
G.5 Die Segmentregister des Prozessors 946
G.6 Die Adreßregister des Prozessors 946
G.7 Das Machine-Status-Word (MSW) 946
G.8 Die Kontrollregister des Prozessors 947
G.8.1 CR0 947
G.8.2 CR1 949
G.8.3 CR2 949
G.8.4 Das Page-Directory-Base-Register (PDBR; CR3) 949
G.8.5 CR4 950
G.9 Die Debugregister des Prozessors 951
G.9.1 Breakpoint-Address-Register DR0 bis DR3 951
G.9.2 DR4 und DR5 951
G.9.3 Debug-Statusregister DR6 952
G.9.4 Debug-Kontrollregister DR7 953
G.10 Die FPU-Register des Prozessors 954
G.11 Die MMX-Register des Prozessors 955

H ASCII- und ANSI-Tabelle 956

I Neue MASM- und TASM-Versionen 958

J Neue Stringtypen 960

Index 965

Vorwort zur vierten Auflage

Lassen Sie sich durch die neue Aufmachung nicht täuschen! Ja, Sie halten die vierte Auflage des Assembler-Buches in den Händen, eines Buches, das seit nunmehr exakt fünf Jahren äußerst erfolgreich auf dem Markt ist. Das bedeutet, daß die Assemblerprogrammierung auch heute noch, unter Windows 98 und NT, unter Programmierern eine nicht unbedeutende Stellung hat.

Dieser Situation möchte ich gern Rechnung tragen. Als ich 1993 begann, das Buch zu schreiben, wollte ich meinen Lesern dabei helfen, mit Hilfe von Assembler bestimmte Funktionen eines Programms zu optimieren. Es ging niemals darum, vollständige Programme in Assembler zu entwickeln – ein Ansinnen, das schon unter DOS nur in wenigen Ausnahmesituationen sinnvoll war. Noch weniger ist das unter Windows 3.x oder den echten 32-Bit-Systemen der Fall. Genau aus diesem Grunde habe ich in den vorangegangenen Auflagen wenig Aufmerksamkeit auf Windows gerichtet – ja ich habe es fast sogar gänzlich gemieden, beispielsweise dadurch, daß ich wesentliche Prozessorbefehle gar nicht erst besprochen habe. Mir war es wichtig, in die Assemblerprogrammierung einzuführen und aufzuzeigen, wie Assemblermodule in Hochsprachenprogramme eingebunden werden können.

An dieser Einstellung hat sich nichts geändert. Auch heute bin ich der Meinung, daß man Assembler nur dann richtig einsetzt, wenn man nicht versucht, mit Kanonen auf Spatzen zu schießen. Es besteht absolut kein Grund, die mächtigen Hochsprachencompiler durch den Assembler zu ersetzen. Weshalb es eigentlich keinen Grund für eine neue Auflage gibt – denn Revolutionäres hat sich beim Thema Assembler seit fünf Jahren nicht ereignet. Dennoch habe ich mich entschieden, das Buch ein wenig zu überarbeiten und um einige Kapitel zu ergänzen, in denen ich einige Prinzipien der modernen Betriebssysteme erläutere, so z.B. den Protected-Mode und seine Schutzkonzepte.

Das soll nun nicht heißen, daß ich Ihnen im Rahmen dieses Buches Wege aufzeigen möchte, mit den einfachen, ja manchmal geradezu brachialen Möglichkeiten des »primitiven« Assemblers sinnvolle Beschränkungen, die der Protected-Mode den Anwendungsprogrammierern auferlegt, außer Kraft zu setzen. Doch hat sich an vielen Leserreaktionen gezeigt, daß ein wenig Aufklärungsbedarf besteht: Nicht alles, was früher unter DOS und Windows 3.x möglich war, ist auch heute noch machbar. Interrupts und freie Portzugriffe sind hier die Stichworte. Mit der vierten Auflage möchte ich Ihnen einige Hintergründe schildern, ohne den Anspruch erheben zu wollen oder zu können, Ihnen erschöpfend Auskunft zur Betriebssystemprogrammierung und -überlistung zu geben.

Trutz Podschun *München, im Januar 1999*

Vorwort zur dritten Auflage

Seit Erscheinen der zweiten Auflage dieses Buches hat sich einiges getan. Eine neue Prozessorgeneration kam auf den Markt, deren Eintreffen mit gemischten Gefühlen aufgenommen wurde – teilweise wohl auch aufgrund von Unwissenheit zu Unrecht verdammt. Windows 95 wurde eingeführt und mit diesem Betriebssystem neue Anwendungsprogramme, die die schönen neuen Möglichkeiten mehr oder weniger gut nutzen. Programmentwicklung aus Komponenten – ein Schlagwort, das nun die Runde macht – Visual Basic, Delphi etc. lassen grüßen. Grund also, nachzuprüfen, ob diese Neuerungen sich auch irgendwie auf Assembler ausgewirkt haben oder auswirken werden.

Trotz aller Weiterentwicklung, visueller Programmierung mit Komponenten, objektorientierten Betriebssystemen und superskalaren Prozessorarchitekturen hat mein Vorwort zur ersten Auflage nichts an Aktualität verloren – ganz im Gegenteil! Je einfacher und »intuitiver« anspruchsvolle Programmierung wird, je größer und umfangreicher Software heute ist, je leistungsfähiger und modularer Programme geschrieben werden können, desto eher vergißt man, daß auch heute noch RAM-Ausstattungen von 16 MByte für Otto Normalanwender eher die Ausnahme sind und nicht jeder über Festplatten jenseits der 1-GByte-Grenze verfügt, die auch schnell genug sind, zur Auslagerung von Programmteilen *on the fly* benutzt werden zu können.

Leider tendieren aber auch heute noch die meisten Programmhersteller dazu, in Prozessorgeschwindigkeit, RAM und *disk space* zu schwelgen – man betrachte nur die gängigsten Anwendungsprogramme (die natürlich auch ich benutze). Resultat: größere Programme brauchen schnellere Prozessoren / mehr RAM / größere und schnellere Festplatten, was dann größere, »bessere« und umfangreichere Programme nach sich zieht, mit noch schnelleren Prozessoren / mehr RAM / größeren und schnelleren Festplatten ... Und dann das große, ungläubige Erstaunen, wenn trotz erheblich schnellerer Arbeitsgeschwindigkeit des Pentium Pro im Vergleich zu seinem Vorgänger die Programme langsamer werden!

Was war passiert? Ob all der nur noch von wenigen Super-Insidern verfolgbaren, rasanten Entwicklungen war wohl nur wenigen aufgefallen, daß man hin und wieder ein bißchen Pause machen sollte – um nachzudenken. Auf Prozessoren, die für 32-Bit-Systeme optimiert wurden, wird natürlich jeder bestraft, der 16-Bit-Programme nutzt! Wohlgemerkt: ich möchte die Entwicklung nicht aufhalten – auch ich freue mich über die Fähigkeiten heutiger Programme! Aber muß dies alles so schnell gehen und, vor allem, so unreflektiert? Nein, man sollte etwas nachdenklicher werden. Und Programmierung in Assembler zwingt zum Nachdenken – auch wenn man nur kleine Programmteile entwickelt.

Gerne hätte ich etwas zu Visual Basic 4.0 von Microsoft geschrieben, da es ja angetreten ist, Delphi von Borland Paroli zu bieten. Doch leider kann ich dies nicht. Auch Visual Basic 4.0 ist, wie all seine Vorgänger, eine interpretierende Hochsprache, was bedeutet, daß der Compiler keinen lauffähigen Code wie bei C oder Pascal erzeugt, sondern einen als p-Code bezeichneten »Pseudocode«, der dann zur Laufzeit des Programmes durch einen Interpreter bearbeitet und ausgeführt werden muß. Und dieser mag wohl in Assembler programmierte Module nicht! Denn leider waren das einzige, was in der Hilfe zu Visual Basic 4.0 zum Thema Assembler zu erfahren ist, drei Einträge, deren Zusammenhang mit Assembler/Assembly mir nicht klar ist und die jede andere Information geben, nur nicht die gewünschte. Nachdem auch Primär- und Sekundärliteratur, selbst anglo-amerikanischer Herkunft, keinerlei Information zu entlocken war und alle noch so heftigen Versuche am Objekt selbst erfolglos waren, muß ich es wohl bei der Bemerkung bewenden lassen, daß Visual Basic 4.0 lediglich zum sogenannten *mixed language programming* in der Lage ist, was bedeutet, daß es DLLs einbinden kann, die in anderen Sprachen programmiert sind. Und das wiederum bedeutet, daß Sie DLLs schreiben können, die in Assembler programmiert wurden – entweder vollständig, oder teilweise eingebettet in Module von anderen Hochsprachen wie Pascal oder C.

Mein Dank gilt Herrn Rainer Römer von Microsoft und Frau Martina Prinz von Borland, die mir wertvolle Unterstützung bei der Erstellung der vorliegenden Auflage haben zukommen lassen. Ganz besonders herzlichen Dank schulde ich auch Herrn Daniel Reischert und den Mitarbeitern von Addison-Wesley. Ohne all die Helfer im Hintergrund, die bei der Realisierung des Buches, dessen Weiterentwicklung und bei der Beantwortung und Weiterleitung von Kritik, Fragen und Anregungen meiner Leser mitgewirkt haben, hätte Das Assembler-Buch sicherlich nicht den Erfolg gehabt, der zur dritten Auflage führte.

Trutz Podschun *München, im Mai 1996*

Vorwort zur zweiten Auflage

Totgesagte leben länger! Diese Erfahrung haben auch der Verlag und ich gemacht, nachdem sich kurze Zeit nach Erscheinen der ersten Auflage die Frage stellte, ob sie nachgedruckt oder eine zweite hergestellt werden soll. Dies zeigt zum einen, daß ganz offensichtlich ein großes Interesse an Literatur zum Thema *Assembler* besteht, zum anderen, daß es wenig gutes Material gibt. Und das wiederum spornt natürlich an!

Wir haben uns entschieden, eine zweite Auflage zu drucken, vor allem deshalb, weil es von Borland eine neue Version des *Turbo Assemblers*, die Version 4.0, gibt. Ich wollte nicht versäumen, Ihnen zumindest einen groben Überblick über die Veränderungen zu geben, die mit dieser Version zusammenhängen. Da sich an den Assembler*befehlen* nichts geändert hat, handeln wir diese Version in einem Kapitel im Anhang ab.

Auch dieses Mal möchte ich dem bewährten Team bestehend aus Susanne Spitzer und Barbara Lauer vom Verlag recht herzlich für ihre Hilfe danken. Dank schulde ich auch Martina Prinz von Borland für ihre wertvolle Unterstützung.

Trutz Podschun *München, im Oktober 1994*

Vorwort

Der *Assembler* ist tot! Glaubt man den Aussagen der meisten sogenannten Fachleute, so ist das Ende dieses Ungetüms aus den grauen Vorzeiten der modernen Programmierung nicht nur gewiß, sondern schon längst da. Kein wirklich ernstzunehmender Mensch, so wird uns eingeredet, programmiert heute noch in *Assembler*! So sind ganze Heerscharen von Softwareschmieden stolz darauf, endlich Programme mit dem Qualitätsnachweis »programmiert in *C*« vermarkten zu können. Auch *Pascal* hat sich ein wenig mehr etabliert, seit man in den Sprachumfang auch Elemente der *objektorientierten Programmierung* aufgenommen hat. Und das große Leittier aller Softwarehersteller, die Firma Microsoft, wirbt ebenfalls damit, daß »*Windows* praktisch vollständig in *C* programmiert ist«.

Ist der *Assembler* also wirklich tot? Kann sich ein Werkzeug, das mit einem schlichten Schraubenzieher verglichen werden kann, im Zeitalter der elektronischen Schraubendreher und Akkuschrauber mit und ohne Drehmomentüberwachung überhaupt noch halten? Die Antwort ist ganz eindeutig: ja! Und mehr noch: Der *Assembler* wird immer wichtiger, wie dieses Buch zeigt.

Natürlich wird sich heutzutage niemand mehr das Mammutprojekt vornehmen, ein Betriebssystem in *Assembler* zu entwickeln oder komplexe Programme! Ja, nicht einmal mehr sehr einfache. Moderne Software muß flexibel sein, leicht portierbar, pflegeleicht und vor allem schnell entwickelbar. In einer Zeit, in der die Halbwertszeit von Anwendungsprogrammen in der Größenordnung von einem Jahr liegt, die Entwicklungszeit aber mehrere hundert Mannjahre beträgt, kann sich tatsächlich niemand mehr erlauben, einen Großteil seiner Zeit damit zu verbringen, das Rad neu zu erfinden. Modulare und objektorientierte Programmierung und Bibliotheken sind deshalb ganz wichtige Elemente bei der Programmierung von heute. Und diese sind nun einmal mit einer Hochsprache wie *C* oder *Pascal* leichter realisierbar.

Doch sollte man sich nicht täuschen! Auch der *Assembler* wurde weiterentwickelt und ist, ebenso wie andere »richtige« Programmiersprachen auch, mit seinen Vorgängern nicht mehr vergleichbar. Wer z.B. lächelt heute nicht über die damals revolutionäre, heute längst überholte Version 2.0 von *Turbo Pascal*?

In der neuesten Version unterstützt *Assembler* sogar *OOP* und *Windows*! Daher wird es, wie überall in unserer Welt, auch für den *Assembler* noch eine ökologische Nische geben. Und die liegt immer dort, wo es darum geht, Programmcode zu optimieren. Bedingt durch den Formalismus, den eine Hochsprache betreiben muß, damit alles reibungslos abläuft, kann es in bestimmten Situationen sinnvoll sein, korrigierend einzugreifen. Beispiele für solche Nischen sind Programmteile, die zeitkritisch sind oder, aus welchen Gründen auch immer, einen bestimmten Codeumfang nicht übersteigen dürfen.

Zwar ist bei den heutigen Preisen für RAM die Speichergröße eines Rechners kein wirklich ernsthaftes Kriterium mehr. *Windows* macht es uns ja vor: Unter 4 MByte RAM ist ein zufriedenstellendes Arbeiten heute praktisch nicht mehr möglich, *OS/2* fängt unter

4 MByte gar nicht erst an, und somit wird so ziemlich jeder moderne Rechner mit mindestens 4 MByte RAM ausgeliefert! Was bedeutet es da, wenn man durch Codeoptimierung ein paar kByte spart?

Auch sind die Prozessoren von heute eigentlich recht schnell. Taktfrequenzen von 25 bis 33 MHz sind *state of the art*, über 50 MHz regt sich außer ein paar Herstellern von Kühlkörpern für unechte »50-MHzer« niemand mehr auf, und mit den neuen »Frequenzverdopplern« sind »locker 100 MHz drin«. Auch unterstützen die modernen Prozessoren die 32-Bit-Adressierung, und mit *Windows NT* und *OS/2* liegen Betriebssysteme vor, die diese Adressierung nutzen können! Also könnte man doch annehmen, daß von der Geschwindigkeitsseite her eine *Assembler*-Optimierung nicht mehr erforderlich ist. Aber wer schon einmal ernsthaft Grafikprogrammierung betrieben hat, der wird zustimmen, daß die Optimierung einer Routine zum Zeichnen von Geraden, Ellipsen und Flächen nicht ganz unwichtig ist! So wurde ja *Windows 3.0* gerade aufgrund der geringen Geschwindigkeit bei der Grafikausgabe anfangs kritisiert – zurecht! Und Hardwarefirmen verdienten und verdienen sich heute noch mit *Accelerator*-Karten unterschiedlicher Wirksamkeit goldene Nasen. In solchen Fällen kann ein Geschwindigkeitszuwachs von 30% bei der Optimierung einer Routine schon Wunder wirken und die Nerven des Anwenders schonen. Aber was sagen Sie dann zu Zuwächsen von ca. 3000% (in Worten: dreitausend!)?

Oder denken Sie an komplexe (durchaus auch im Sinne des Wortes!) mathematische Rechenoperationen, wie sie z.B. bei der Berechnung von Fraktalen notwendig werden. Dieser Bereich lebt quasi davon, daß die Routinen, die zum Einsatz kommen, geschwindigkeitsoptimiert sind. Lassen Sie einmal, auch auf einem 33-MHz-486er, ein *Apfelmännchen* mit den *Pascal*- oder *C*-Routinen berechnen, und vergleichen Sie das Ergebnis mit einem Programm, das an den wichtigen Stellen optimierte Routinen besitzt. Ein Zuwachs an Geschwindigkeit von 1000% ist hierbei keine Seltenheit! Und auch der Speicher ist trotz der derzeitigen Situation nicht unendlich groß. Wer einmal *Word for Windows* und *Publishers Paintbrush* parallel hat laufen lassen und sich dann den Spaß gemacht hat, die noch zur Verfügung stehenden Ressourcen zu überprüfen, der wird dem zustimmen können. Auch im Jahre des ersten PCs glaubte man, 640 kByte RAM sind für die nächsten Jahrhunderte ausreichend!

Doch es gibt noch einen weiteren Aspekt. Manche Dinge sind in Hochsprachen sehr schwer realisierbar, da diese meistens nicht auf »exotische« Problemstellungen eingestellt sind. Anders der *Assembler*, der »unspezifisch« genug ist, die Hochsprachenhürden zu umgehen. Wie würden Sie beispielsweise ein *Type-Casting* einer Fließkommazahl in eine Integerzahl realisieren? In *Pascal* z.B. ist dies nicht vorgesehen – der Compiler hilft bei diesem Problem nicht. Aber bevor Sie sich nun daran machen, darüber nachzudenken und komplizierte Routinen zu entwickeln: Ich habe die optimale Lösung! Einfach in Ihren Hochsprachentext **zwei** Coprozessorbefehle (= 8 Bytes) in *Assembler* eingebunden, und das war's. Sie glauben es nicht? Sehen Sie auf Seite 408 nach!

Offensichtlich hat sich diese Einschätzung auch bei einigen Profis durchgesetzt. So erlebte der *Assembler* eine erste Renaissance, als Borland ihn mit *Turbo Pascal 5.0* in den Sprachumfang dieser Hochsprache integrierte. Parallel war noch der »*stand alo-*

ne« Assembler TASM aus der gleichen Softwareschmiede verfügbar. Allerdings als eher ungeliebtes Unikum für die ganz Hartgesottenen, diejenigen, die auch die Kommandozeilenversion von *Turbo Pascal* der integrierten Entwicklungsumgebung vorziehen. Denn über eine *IDE* wie für die Hochsprachen verfügte *TASM* ebenso wenig wie bis vor kurzem sein Vorbild *MASM* von Microsoft.

Das hat sich bis heute nicht grundlegend verändert. Zwar verbreitet Microsoft, meines Wissens bis heute allein auf weiter Flur, mit *MASM 6.0* auch eine integrierte Entwicklungsumgebung namens *PWB* oder ***P**rogrammer's **W**ork **B**ench*. Doch bis zu wem hat sich das eigentlich herumgesprochen? *TASM* kennt solchen Luxus noch nicht – wenn auch nun mit *Turbo Pascal 7.0* dieser *TASM* in die IDE von *Borland Pascal* integriert wurde. So kann mit dem Editor von *Turbo Pascal* ein *Assembler*-Quelltext erstellt und über das *TOOLS*-Menü aus der *IDE* heraus assembliert werden. Die auf diese Weise erstellten OBJ-Dateien können ganz zwanglos in *Turbo-Pascal*-Programme eingebunden werden. In einer Sitzung! Sogar die Fehlerbehandlung ist in der IDE möglich. Und das ganze nicht nur in der *DOS*-Version! Auch unter *Windows* scheint der *Assembler* wieder salonfähig zu werden.

Sicherlich muß es Gründe haben, wenn so innovationsfreudige Unternehmen wie Microsoft und Borland sich nicht nur nicht vom *Assembler* trennen, sondern ihm zu neuer Größe verhelfen. Fehlt eigentlich nur noch, den *Programmierern*, also **Ihnen**, dieses machtvolle Instrument näherzubringen. Denn ob all der stürmischen Entwicklungen in den Hochsprachen wurde selbst von den Firmen, die den *Assembler* nicht vollständig abgeschrieben hatten, sehr wenig unternommen, ihn auch zu vermitteln. Oder sind Sie sich als *C*-Programmierer eigentlich darüber im klaren, daß Ihr *C*-Compiler sogar in der Lage ist, *Assembler*-Listings aus den *C*-Quelltexten zu erstellen? Ja, mehr noch – daß der *C*-Compiler eigentlich nichts anderes als ein *Assembler*-Code-Generator ist, der eigentliche ausführbare Code dann aber von einem *Assembler* erstellt wird? Die gnadenlose Auflistung der sogenannten *Mnemonics* oder *Opcodes* in den Handbüchern ohne jegliche Erklärung, ergänzt um die kurze Vorstellung der häufig genug wenig aussagekräftigen *Assembler*anweisungen, führte nicht gerade dazu, Interesse zu entwickeln. Und auch die wenigen wirklich guten Bücher zu diesem Thema konnten dies nicht ändern. Aber vielleicht wird dies ja nun ein wenig anders.

An dieser Stelle möchte ich noch recht herzlich Herrn Bernhard Wopperer von der Firma Intel und Herrn Ralph Machholz von der Firma Microsoft danken, die mich tatkräftig bei der Realisation dieses Buches unterstützt haben. Sehr großen Dank schulde ich Judith Muhr, die wesentliche Impulse und Ideen beigetragen hat und entscheidend mit dafür verantwortlich ist, daß das Buch die vorliegende Form erhalten hat. Last but not least möchte ich Susanne Spitzer von Addison-Wesley danken, bei der ich während der Realisation dieses Buches jederzeit auf offene Ohren und wertvolle Unterstützung gestoßen bin, sowie Frau Barbara Lauer, die wesentlich an der Verwirklichung in der vorliegenden Form beteiligt war.

Also: der *Assembler* ist tot! Es lebe der *Assembler*! Und noch möglichst lange.

Trutz Podschun *Würzburg, im Januar 1994*

Einleitung

Was ist eigentlich *der Assembler*? Während man z.B. von einem *Turbo Pascal-Compiler*, einem Microsoft *C++-Compiler* oder einem *dBase-Compiler* namens CLIPPER spricht, erschlägt man alles, was sich mit dem Assemblieren von sogenannten *Mnemonics* beschäftigt, mit dem Begriff *»der Assembler«*. Dabei sind die Unterschiede zwischen den einzelnen auf dem Markt befindlichen *Assemblern* z. T. ebenso groß wie die zwischen unterschiedlichen Pascal- oder C-Compilern!

Dennoch ist es nicht ganz unrichtig, von *dem Assembler* zu sprechen. Denn im Unterschied zu Compilern, die, je nach Programmierung, unterschiedliche Compilate des gleichen Quelltextes erzeugen, erzeugen alle Assembler das gleiche Assemblat! Schaut man sich z.B. den compilierten Code des Microsoft C++-Compilers an und vergleicht ihn mit dem vom Borland-Compiler, so findet man z. T. drastische Unterschiede, auch im Laufverhalten, in der Codegröße, in Eigenheiten.

Assemblierte Codes von TASM und MASM dagegen sind gleich! Sie unterscheiden sich nicht und können bedenkenlos miteinander ausgetauscht werden. So weigert sich der Turbo Debugger nicht, auch Code zu debuggen, den Microsofts »**M**acro **As**se**m**bler« MASM erzeugt hat. Und ebenso wenig Schwierigkeiten bereitet es, mit SYMDEB, dem ehrwürdigen »**Sym**bolischen **Deb**ugger« oder mit dem neueren CODEVIEW, beide von Microsoft, Code anzusehen, den der **T**urbo **As**se**m**bler TASM assembliert hat. Ja, selbst das gute, alte DOS-DEBUG verweigert seine Dienste nicht!

Dies ist eigentlich ganz einfach zu erklären. Eine Hochsprache ist eine Programmiersprache, die im Prinzip nur für den Menschen, und nicht einmal für alle, verständlich ist. Das Arbeitstier im Computer aber, der Prozessor, kann mit den Befehlen einer Hochsprache nichts anfangen. Er versteht nur einige wenige Befehle, sogenannte *Operation Codes* oder kurz *Opcodes*. Diese Codes, in der richtigen Reihenfolge aneinandergehängt, lassen den Prozessor das machen, was man von ihm erwartet.

Hochsprachen*compiler* machen nichts anderes, als die für Menschen verständlichen Hochsprachenbefehle in eine Reihe von Opcodes zu übersetzen, die der Prozessor abarbeiten kann. Nun ist es aber mit den Übersetzern in der EDV ähnlich wie mit solchen im richtigen Leben. Jeder hat seine Eigenheiten, Redewendungen und Lieblingswörter. Jedem Dolmetscher billigt man zu, einen eigenen Übersetzungsstil zu haben! Was dabei herauskommt, ist (hoffentlich!) dennoch verständlich, auch wenn der eine eine etwas andere Satzstellung benutzt als der andere.

Nicht anders verhält es sich mit den Compilern. Auch sie produzieren letztlich alle einen Code, der den Prozessor zu einem bestimmten Handeln veranlaßt. Aber auch Compiler haben eigene Übersetzungsangewohnheiten. Wenn z.B. ein *Pascal-Compiler* ein Unterprogramm aufruft, so übergibt er die notwendigen Parameter »über den Stack«, eine bestimmte Struktur im Speicher. Er überläßt es der aufgeru-

fenen Routine, wieder aufzuräumen. Dagegen übergibt diese Routine ihre Funktionswerte (wenn überhaupt) in einer Weise, die der Pascal-Compiler ihr vorgegeben hat.

C-Compiler übergeben ihre Parameter auch über den Stack, jedoch in umgekehrter Reihenfolge. Und sie geben die Verantwortung, den Stack wieder aufzuräumen, auch an das aufrufende Programmteil weiter, nicht an das aufgerufene! Die Befehle, die in beiden Fällen benutzt werden, sind die gleichen! Es sind ja nur die Opcodes, die der Prozessor versteht. Die Reihenfolge und Art, in der diese Opcodes erzeugt werden, bewirken, daß sich die Hochsprachen untereinander nicht so ohne weiteres mischen lassen.

Aus diesem Grunde macht es Sinn, von verschiedenen Compilern zu sprechen, aber nur von »dem Assembler«. Denn – egal von welchem Hersteller er ist – alle Assembler benutzen und erstellen die gleichen Opcodes. Schließlich sind diese ja nichts anderes als die zu einer Zahl gewordenen Prozessorbefehle, die Mnemonics. Und die Reihenfolge, in der diese abzuarbeiten sind, bestimmt in diesem Fall nicht der Compiler, sondern **Sie**, der Assemblerprogrammierer!

Um Ihnen bei diesem Vorhaben behilflich zu sein, wurde dieses Buch geschrieben. Es soll eine Einführung in die Programmierung mit Assembler sein und Sie in die Geheimnisse einer faszinierenden Art der Programmierung einweihen. So werden im Teil 1 die Grundlagen gelegt, um überhaupt mit dem Prozessor »reden «zu können. Wir werden hierzu mit einem typischen Vertreter beginnen, dem 8086. Er ist zwar lange nicht mehr Maß aller Dinge, jedoch bilden die Opcodes, die er kennt, auch in den supermodernen Prozessoren 80486 und Pentium die Basis jedes Befehlssatzes. Und das wird vermutlich noch eine ganze Weile so bleiben.

Wir werden uns in diesem Teil mit dem sogenannten *Real-Mode* des Prozessors beschäftigen und die Register kennenlernen, mit denen der Prozessor arbeitet. Wer nun enttäuscht das Buch am liebsten wieder weglegen möchte, da er hoffte, etwas über den *Protected-Mode* zu erfahren, sei getröstet: Alle Befehle des Real-Mode arbeiten auch im Protected-Mode. Sie können also durchaus Module für Windows oder OS/2 erstellen! Lediglich die Befehle, die *speziell* für den Protected-Mode oder den *Virtual-8086-Mode* geschaffen wurden, werden wir nicht besprechen, denn diese brauchen Sie nur, wenn Sie ein Betriebssystem entwickeln oder an ihm herumbasteln wollen! Dies zu vermitteln, sprengt den Rahmen dieses Buches bei weitem. Anschließend werden wir das Geheimnis des Adreßraumes lüften und somit einige Beschränkungen verstehen, unter denen das ebenfalls betagte Betriebssystem DOS auch heute noch leidet. Den weitaus größten Part des ersten Teils wird naturgemäß die Besprechung der Befehle des 8086 ausmachen. In den einzelnen Kapiteln werden wir lernen, mit den Prozessorregistern zu arbeiten und die Opcodes einzusetzen.

Ein ebenfalls großer Part des ersten Teils wird sich mit dem Coprozessor beschäftigen. Auch hier werden wir zunächst die Register kennenlernen und erfahren, wie ein mathematischer Coprozessor mit dem Prozessor zusammenarbeitet. Natürlich

werden wir auch seine Befehle genauer studieren. In weiteren Kapiteln werden dann die Veränderungen und Ergänzungen besprochen, die seit dem 8086 in den Befehlssatz der Nachfolger Eingang gefunden haben.

Teil 2 wird sich zunächst damit beschäftigen, die im Teil 1 erworbenen theoretischen Kenntnisse in die Praxis umzusetzen. Wir werden die ersten Befehle assemblieren und Schritt für Schritt in die Prozessorprogrammierung einsteigen. Anschließend werden wir uns mit der fortgeschrittenen Programmierung des Prozessors befassen. Wir werden so nützliche Dinge wie Makros, lokale Symbole usw. kennenlernen. Wir werden Schnittstellen zu Hochsprachen analysieren und erfahren, wie man Assemblerprogramme in Hochsprachen einbindet. Danach werden wir uns mit den Eigenheiten des *Integrierten Assemblers* beschäftigen. Damit das ganze nicht zu trocken wird, werden wir an verschiedenen Stellen Beispiele sehen, die von Hochsprachencompilern erzeugt wurden, und lernen, worin die Eigenheiten des erzeugten Codes liegen und wie man diesen optimieren kann.

Teil 3 des Buches ist ein Referenzteil, in dem die Befehle und Anweisungen des *Assemblers* alphabetisch aufgeführt sind.

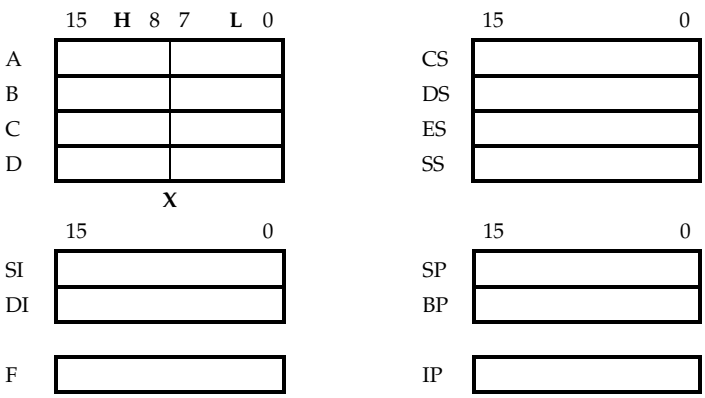
Abschließend sei noch bemerkt: Die Assemblerteile wurden unter *MASM 6.0* von Microsoft erstellt und mit *TASM 3.2* von Borland ausgetestet. Für die Anbindung an Hochsprachen wurden *Borland Pascal 7.0*, *Visual C++* von Microsoft und *Turbo C++ 3.0* von Borland verwendet. Es wurde versucht – und weitgehend sogar erfolgreich! – auf spezifische Erweiterungen und Besonderheiten verschiedener Assembler-, aber auch Hochsprachendialekte zu verzichten. Dies bedeutet insbesondere, daß, wann immer möglich, nur die grundlegenden Assemblerbefehle verwendet werden. Auch wurden die beiden C++-Compiler nur dazu benutzt, C-kompatiblen Code zu erzeugen. Daher dürfte es keine großen Schwierigkeiten bereiten, die dem Buch zugrundeliegenden Erkenntnisse auch in anderen Dialekten und Sprachen und damit in einem breiten Bereich anwenden zu können.

Teil 1 Grundlagen

1 Die Register des 8086

Wenn wir in diesem Kapitel explizit von den Registern des 8086 sprechen, ist das eigentlich nicht ganz richtig. Treffender wäre der Titel »Die Register der Intelprozessoren im Real-Mode«. Denn in dieser Betriebsart verfügen alle derzeitigen Prozessoren der Baureihe x86 von Intel, also auch die Prozessoren 80286, 80386, 80486 und Pentium, nur über diese Register. Dies hat zur Folge, daß der Registersatz des Begründers der Prozessorenfamilie als allgemein gültiges Modell für alle Prozessoren herhalten kann. Nach der Besprechung dieses Registersatzes kennen wir auch alle anderen – aber, wie gesagt, nur im sogenannten Real-Mode. Wir kommen gleich darauf zurück, was das ist.

Wir besprechen den 8086 als typischen Vertreter seiner Zunft. Dies gilt insbesondere auch für seinen Zwillingsbruder, den 8088. Dieser Prozessor unterscheidet sich vom 8086 in nur einem Punkt. Er besitzt nur einen 8 Bit breiten Datenbus, während der 8086 über einen 16 Bit breiten Datenbus auf seine Daten zugreift. Sie als Programmierer braucht das nicht zu interessieren! Daß der 8088 zweimal hintereinander auf den RAM zuzugreifen hat, wenn er ein Wort aus dem Datenspeicher holen oder es dort ablegen will, weiß er selbst ganz genau und tut es daher auch recht zuverlässig!



Der Registersatz des 8086 besteht zunächst aus vier *Rechenregistern*, die alle eine Breite von 16 Bit haben. In diesen Registern finden die Manipulationen der Daten statt, mit denen der Prozessor rechnet. Die Namen dieser Register sind A, B, C und D. Nun könnte man der Meinung sein, daß dies eine wenig einfallsreiche, alphabetische Benennung ist; dem ist

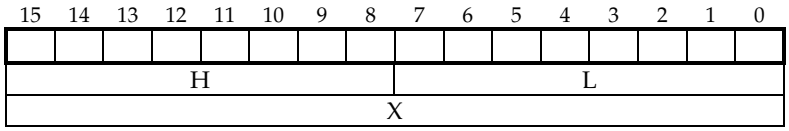
aber bei weitem nicht so! So steht »A« für *Akkumulator* und bezeichnet ein Register, in dem Daten »akkumuliert« werden können. »B« kennzeichnet ein *Base-Register*, »C« ein *Counter-Register* und »D« ein *Datenregister*.

Das heißt, daß diese Register nicht einheitlich sind. Jedes Register hat spezielle Aufgaben, die im folgenden erläutert werden. Diese Register haben dennoch viel gemeinsam. So können die meisten Operationen, die auf diese Rechenregister anwendbar sind, auf alle vier Register angewendet werden, wie z.B. einige Befehle zum Berechnen von Ergebnissen arithmetischer Operationen. Ferner können diese 16-Bit-Register aufgeteilt werden in jeweils zwei 8 Bit breite Register. Dies unterscheidet sie von allen anderen Registern des 8086 (und, um es ein letztes Mal ausdrücklich zu sagen, aller anderen Prozessoren).

Um nun festzulegen, ob in den Akkumulator z.B. ein 16-Bit-Wort eingetragen werden soll oder ein 8-Bit-Byte, erhalten die Namen einen weiteren Buchstaben, der dies definiert. So wird jedes dieser vier Register immer mit »X« versehen, falls ein *Wort* (16 Bit) Verwendung finden soll, also AX, BX, CX und DX.

Sollen *Bytes* (8 Bit) verarbeitet werden, muß entschieden werden, an welche Position im Register dieses Byte gesetzt werden soll. Es besteht die Möglichkeit, das Byte an die Bitpositionen 15 bis 8 zu schreiben. Diesen Teil eines Worts nennt man *High Byte (höherwertiges Byte)*, weshalb die entsprechende Registerposition auch mit »H« abgekürzt wird. Umgekehrt ist auch das *Low Byte (niederwertiges Byte)* beschreibbar: die Positionen 7 bis 0. Die Abkürzung hierfür ist »L«.

Schauen wir uns ein solches Rechenregister noch einmal etwas genauer an.



Die Bitpositionen 15 bis 8 bilden zusammen das höherwertige Byte H des Registers, die Bitpositionen 7 bis 0 das niederwertige Byte L. H und L zusammen ergeben das 16-Bit-Register X. Jede einzelne Zelle kann einen Wert von 0 oder 1, eben ein *Bit*, aufnehmen. Da wir pro Register 16 Bits haben, die zwei Zustände annehmen können, kann ein Register also 2^{16} unterschiedliche Zustände besitzen. Dies sind 65.536 Zustände, die Werte zwischen 0 und 65.535 repräsentieren können. Im Anhang finden Sie detailliertere Informationen zur binären Darstellung von Zahlen.

HINWEIS

Halten wir also fest: Jedes der vier Rechenregister kann entweder 16 Bit breite Worte aufnehmen und muß dann mit AX, BX, CX bzw. DX angesprochen werden. Es kann jedoch auch zwei 8 Bit breite Bytes aufnehmen. Je nach Position spricht man diese Bytes dann mit AH, AL, BH, BL, CH, CL, DH bzw. DL an.

Neben den Rechenregistern gibt es auch vier Register, die Adressen aufnehmen können. Diese Register heißen Segmentregister, da in ihnen die Anfangsadressen der Segmente verzeichnet sind, die der Prozessor für seine Tätigkeit braucht. Was ein Segment ist, klären wir im nächsten Kapitel.

Diese Segmentregister lassen sich **nicht** wie die Rechenregister in zwei 8-Bit-Teile aufteilen. Ihre Breite beträgt immer 16 Bit. Sie heißen nach dem Segment, auf das sie deuten, Codesegment- (CS), Datensegment- (DS), Extrasegment- (ES) und Stacksegmentregister (SS). Sie sind, obwohl lebenswichtig, recht langweilig; denn man kann sie nur auslesen oder beschreiben.

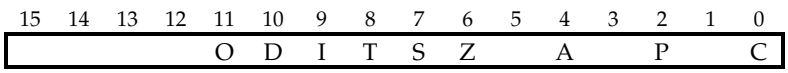
Es gibt zwei weitere wichtige Register, sogenannte Indexregister: das *Source-Indexregister* und das *Destination-Indexregister*. Diese mit SI und DI abgekürzten Register können benutzt werden, wenn ein Zeiger (Index) auf eine Datenstruktur benötigt wird. Wir werden das noch sehen, wenn wir auf die Stringbefehle des Prozessors zu sprechen kommen.

Für die korrekte Arbeitsweise des Prozessors spielen noch zwei weitere Register eine Rolle: SP und BP. Auch dies sind Indexregister, doch zeigen sie auf eine Struktur, die der Prozessor intern bei bestimmten Arten der Adressierung benötigt. Diesen sogenannten Stack werden wir erst im zweiten Teil des Buches näher kennenlernen. Für den Augenblick reicht es zu wissen, daß es das *Stack-Pointer-Register* und das *Base-Pointer-Register* gibt.

Es fehlen nur noch zwei Register, das *Instruction-Pointer-Register* und das *Flagregister*. Das erste von beiden, IP, ist zwar eines der wichtigsten Register überhaupt, denn in ihm merkt sich der Prozessor, an welcher Stelle im Programm er sich befindet, was also als nächster Befehl abgearbeitet werden soll. Dennoch können wir nichts mit diesem Register machen, es wird intern ausschließlich durch den Prozessor verändert.

Das Flagregister F dagegen ist für uns höchst interesssant, wie wir bei der Besprechung der Befehle noch sehen werden. Wir können es manipulieren, zumindest teilweise. Andererseits verändert auch der Prozessor bei der Abarbeitung bestimmter Befehle dessen Inhalt. Dies ermöglicht eine gewisse Art der Kommunikation zwischen Prozessor und Programmierer.

Das Flagregister ist auch in anderer Weise ein ausgefallenes Register: es läßt sich bitweise interpretieren und, zumindest teilweise, manipulieren. Nicht alle 16 Bits dieses Registers haben, zumindest beim 8086, eine Bedeutung. Sehen wir es uns etwas genauer an.



An Bit 11 des Wertes im Flagregister steht das sogenannte *Overflow-Flag* O. Dieses Flag wird vom Prozessor immer dann gesetzt, wenn nach einer Operation ein Überlauf stattgefunden hat. Was es damit auf sich hat, werden wir zu gegebener Zeit klären. Es schließen sich das *Direction-Flag* D, das *Interrupt-Enable-Flag* I sowie das *Trap-* oder *Single-Step-Flag* T an. Diese Flags haben ganz bestimmte Aufgaben, die wir, mit Ausnahme vom Direction-Flag, im Rahmen dieses Buches nicht ansprechen werden.

Das Sign-Flag S, das Zero-Flag Z, das Auxiliary-Flag A sowie das Parity-Flag P werden im übernächsten Kapitel behandelt. Diese Flags spiegeln bestimmte Eigenschaften von Registerinhalten nach verschiedenen Operationen wider, die die Grundlage für Entscheidungen sein können. Sie sind zusammen mit dem Carry-Flag C die für den Programmierer wichtigsten Flags. Alle anderen Bitpositionen sind beim 8086 nicht definiert.

1.1 Der Adreßraum des 8086

Als Adreßraum bezeichnet man bei Prozessoren einfach den Bereich des Speichers, den der Prozessor ansprechen kann, in den er also Daten ablegen oder aus dem er sie wieder lesen kann. Dies ist nicht auf Daten beschränkt. Im Prinzip sind auch die Prozessorbefehle Daten, so daß also auch der Teil des Speichers dazugerechnet wird, in dem sich die Programme befinden. Kurz gesagt ist also der Adreßraum des 8086 der Speicherbereich, mit dem der 8086 arbeiten kann.

Eng verknüpft mit dem Begriff Adreßraum sind zwei weitere Begriffe, die Sie sicherlich kennen, zumindest aber schon einmal gehört haben: Datenbus und Adreßbus. Als Datenbus wird die Menge aller Datenleitungen bezeichnet, über die der Prozessor Daten aus dem Speicher holen oder in den Speicher ablegen kann. Anhand der Anzahl der Datenleitungen teilt man die Prozessoren in 8-Bit-Prozessoren ein (8088, 80188), bei denen acht Datenleitungen vorhanden sind, also 8 Bits parallel übertragen werden können. Prozessoren wie 8086, 80186, 80286 verfügen über 16 Datenleitungen und besitzen somit einen 16-Bit-Datenbus. Die neuen Prozessoren 80386, 80486 und Pentium stellen mit ihrem 32-Bit-Datenbus das derzeitige *Nonplusultra* im PC-Bereich dar.

Doch der Datenbus hat mit dem eigentlichen Adreßraum zunächst nichts zu tun. Denn für den Adreßraum ist der Adreßbus zuständig, also die Summe aller Adreßleitungen, über die ein Prozessor verfügt. Dies sind beim 8088 und 8086 sowie deren Clones, aber auch bei den Nachfolgern 80188 und 80186 20 Stück.

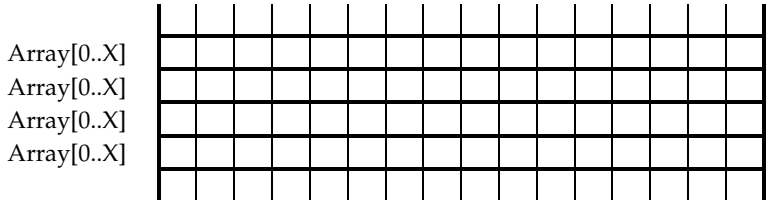
Was heißt das nun? Mit 20 Adreßleitungen kann man 2^{20} Speicherstellen ansprechen. Das sind genau 1.048.576 Bytes oder, wie in der Datenverarbeitung üblich, in Einheiten von sogenannten kByte (= 2^{10} Byte) ausgedrückt: 1024 kByte, was identisch mit 1 MByte ist. Kommt Ihnen diese Zahl irgendwie bekannt vor?

1 MByte ist der Adreßraum, den das Betriebssystem DOS auch heute noch maximal ansprechen kann, selbst in der Version 6.x! Der Hintergrund dafür ist, daß die Entwicklung von DOS eng mit dem Prozessor 8086 verknüpft war. Somit wird DOS als abwärtskompatibles Betriebssystem die bei der Entwicklung der ersten Version programmierten Hindernisse, den Adreßraum zu erweitern, vermutlich auch in die nächsten fünf Versionen mitschleppen müssen. Die »magische Grenze« liegt also nicht etwa im Betriebssystem DOS begründet, sie beruht auf der Anzahl der verfügbaren Adreßleitungen des damals existierenden Prozessors.

Erinnern wir uns an die Register des 8086. Bei deren Besprechung im vorletzten Abschnitt haben wir gelernt, daß sie alle maximal 16 Bit breit sind. Das aber heißt, daß die Adressen des gesamten Adreßraums nicht in ein Register passen, da es unmöglich ist, 20 Bits in ein 16-Bit-Register zu pressen. Und dies ist ein Dilemma! Wir könnten über 20 Adreßleitungen 1 MByte Speicher ansprechen, können es nun aber doch nicht, weil die Register, in die man die Adressen schreiben muß, nur 16 Bit breit sind!

Wie sieht der Ausweg aus? Man benutzt zwei Register für Adressen. Nun stellt sich aber ein anderes Problem ein: Zwei Register à 16 Bit können eine Adresse von 32 Bit fassen. Der 8086 hat jedoch nur 20 Adreßleitungen, womit 12 Bits ungenutzt blieben. Das wollte man nicht. Daher kam man bei Intel auf eine geniale Idee: Speichersegmentierung.

Was heißt das nun? Erklären wir zunächst einmal den Begriff Segment! Gehen wir davon aus, daß ein Segment eine bestimmte Anzahl von Bytes ist, ähnlich z.B. einem *Array[0..X] of Byte* in Pascal. Definieren wir weiterhin, daß es im Prozessor ein Register gibt, das die Nummer eines solchen Segments aufnehmen kann. Ein solches Register ist 16 Bit breit – wir kennen schon aus der Besprechung der Register die vier Segmentregister des 8086: CS, DS, ES und SS.



Der gesamte Speicher läßt sich auf diese Weise als Aneinanderreihung von *Arrays[0..X] of Bytes* ansehen, also als *Array[0..Y] of Array[0..X] of Bytes*.

Wie groß aber ist ein Segment? Nun, ganz einfach: Dividieren wir die maximal mögliche Größe des Adreßraums durch die Anzahl von Segmenten, die man erzeugen kann, so bekommen wir eine Größe für ein Segment. Das aber bedeutet, daß wir 2^{20} (den Maximalwert) durch 2^{16} (den Maximalwert für ein Register) dividieren müssen. Das Resultat ist 2^4. Mit anderen Worten: Durch die Segmentierung des Speichers in Blöcke zu je 16 Bytes kann man den gesamten Adreßraum des 8086 ansprechen. Dazu muß nur die Nummer des Segments in das gewünschte Segmentregister geschrieben werden: Segment 0, wenn man die ersten 16 Bytes ansprechen möchte, Segment 1 für die nächsten 16 Bytes, bis 65535, um die letzten 16 Bytes anzusprechen. Im Pascal-Beispiel:

Arbeitsspeicher = Array[0..65535] of Array[0..15] of Byte

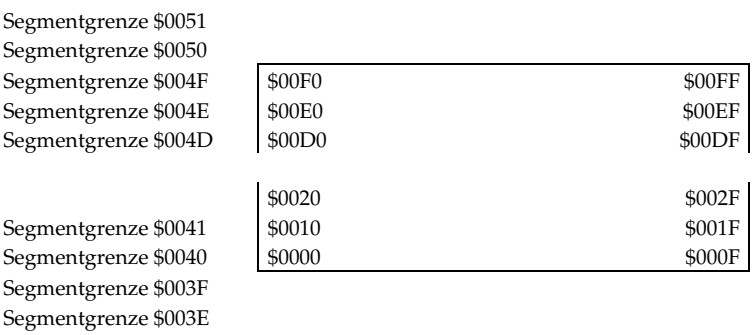
	80	81	82	…												
Segment 4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Segment 3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Segment 2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Segment 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Segment 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Anders ausgedrückt heißt das, daß beginnend mit 0 alle 16 Bytes eine sogenannte Segmentgrenze liegt: zwischen den Bytes 15 und 16, 31 und 32, 47 und 48 etc. Genau diese Segmentgrenzen werden in die Segmentregister eingetragen! Doch eine Frage interessiert uns dennoch brennend! Wir können über die Segmentregister Segmente ansprechen. Aber wie erhält man das Byte Nummer 10 aus Segment 20? Oder Byte 15 aus Segment 40.794? Allgemein formuliert: Wie kann man auf die einzelnen Bytes eines Segments zugreifen?

Hier kommt das zweite Register ins Spiel. In einem zweiten Register wird ein Zeiger gespeichert, der auf das gewünschte Byte im Segment zeigt. Will man also Byte 10 in Segment 20 ansprechen, so muß man in

das Segmentregister die Zahl 20 schreiben und in das andere (»Zeiger«)-Register den Wert 10. Beide Register zusammen können also eindeutig das gewünschte Byte ansprechen!

Doch um die 16 Bytes eines Segments ansprechen zu können, braucht man ja nur 4 Bits. Das zweite Register hat jedoch genau wie das Segmentregister 16 Bits, kann also auf erheblich mehr Bytes als die erforderlichen 16 zeigen. Genauer gesagt: auf 65536! Wenn man ein 16-Bit-Register verwenden muß und nicht 12 Bits ungenutzt verschenken will, so heißt das, daß man mit dem »Zeigerregister« immer mehrere Segmentgrenzen überschreiten kann.



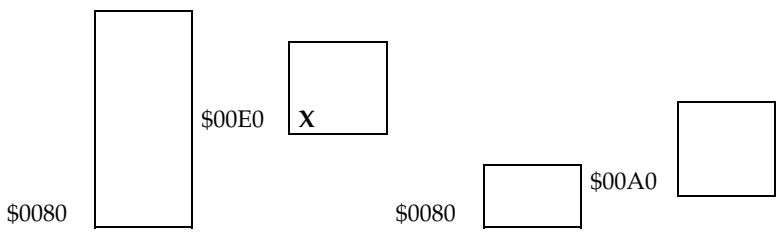
In diesem Beispiel ist das verwendete Segment 256 Byte groß und beginnt an der Segmentgrenze $0040. Man sagt: »Das Segment an Adresse $0040 ist 256 Bytes groß«. Segmente haben also folgende Eigenschaften:

- Segmente können immer nur an Segmentgrenzen beginnen. Als Segmentgrenze gelten Speicherstellen, deren »physikalische« (= vollständige 20-Bit-) Adressen ohne Restbildung durch 16 teilbar sind.
- Es gibt daher insgesamt 65.536 Segmentgrenzen, da 1 MByte maximal verfügbarer Speicherplatz dividiert durch 16 eben diesen Wert ergibt.
- Die Nummer des zu verwendenden Segments wird mit der Nummer der Segmentgrenze bezeichnet, an der es beginnt. Das allererste Segment im Speicher beginnt an der Segmentgrenze 0 und hat daher die Nummer 0. Das letzte Segment beginnt an der Segmentgrenze 65.535 und hat daher genau diese Nummer.
- Segmente sind *mindestens* 16 Bytes groß, können aber *bis zu* 65.536 Bytes groß werden, weil der Zeiger auf den Segmentinhalt ebenfalls Werte zwischen 0 und 65.535 annehmen kann. *Wie groß ein Segment tatsächlich ist, kann mit der Segmentregister-Zeigerkombination nicht festgestellt werden.* Falls nicht irgendwo anders ver-

merkt wird, wie groß ein Segment ist, hat es keine definierte Größe! Man kann dann den gesamten Bereich, angefangen von der Segmentgrenze bis zu einem Zeigerwert von 65.535, ansprechen – egal, ob das sinnvoll ist oder nicht!

- Segmente können sich überlappen. Ein Segment, das an der Segmentgrenze 0 beginnt und 48 Bytes groß ist, überdeckt ein Segment, das an der Segmentgrenze 1 beginnt und 16 Bytes groß ist.

Vor allem der letzte Punkt ist extrem wichtig! Grafisch kann das folgendermaßen ausgedrückt werden:



Auf das markierte Byte *X* kann man zugreifen, indem man die Segmentgrenze $0080 zugrunde legt und einen Zeiger auf das Byte berechnet. Zum absolut gleichen Ergebnis kommt man auch mit den Segmentgrenzen $00E0 oder $00A0 und zwei entsprechenden Zeigern. Aus dem Schaubild können Sie auch ersehen, daß es durchaus sein kann, daß ein Zeiger in einem Fall eine korrekte Stelle im definierten Segment anspricht (linke Spalte), während der gleiche Zeiger, obwohl mathematisch vollkommen korrekt, in der 3. Spalte auf ein Byte zeigt, das außerhalb des angenommenen gültigen Bereichs liegt. Eine Prüfung und/oder Fehlermeldung des Assemblers muß unterbleiben, da dieser nicht wissen kann, wie groß der gültige Bereich an der Segmentgrenze $0080 im obigen Beispiel im jeweiligen Fall ist!

Dies birgt eine große Gefahr, denn durch diesen Mechanismus kann *jedes* Programm auf *jede* Speicherstelle zugreifen. Lesende Zugriffe wären ja nicht so schlimm; aber es geht auch schreibend – beabsichtigt und auch unbeabsichtigt. Dies ist letztlich der Grund dafür, daß es im Real-Mode zu Problemen kommen kann, wenn sich ein (TSR-) Programm nicht an die Restriktionen hält. Das war ein Grund für die Entwicklung des Protected-Mode ab dem 80286. Denn in diesem Modus können Segmente vor Zugriffen geschützt werden – egal, ob lesend oder schreibend! Dies bedeutet, daß nun ein Programm ein anderes nicht mehr stören kann! Aber wie gesagt: nur im Protected-Mode.

Daraus läßt sich folgende Berechnungsregel für die Adresse eines Bytes im Speicher ableiten:

Adresse = 16 · Segmentnummer + Zeiger.

Auf den Sprachgebrauch des *Assemblers* übertragen heißt das

Adresse = 16 · Segment + Offset.

Es werden also Segment und Segmentnummer gleichgesetzt, und der Zeiger wird als Offset (= »Versatz«) zu diesem Segment bezeichnet. In Assemblerschreibweise wird immer das Segment zuerst genannt, gefolgt von einem Doppelpunkt und dem Offset, also z.B. $4711:$0815.

Diese eigenwillige Adressierungsart führt zu der seltsamen Erscheinung, daß Adressen, die auf ein und dieselbe Speicherstelle verweisen, unterschiedlich angegeben werden können. Lassen Sie uns dies am Beispiel des Videospeichers demonstrieren, in dem die Informationen gespeichert sind, die auf dem Bildschirm dargestellt werden.

Dieser Bereich beginnt im Textmodus bei verschiedenen Videokarten an der physikalischen (d.h. aufgerechneten 20-Bit-) Stelle $B8000. Nun gibt es verschiedene Möglichkeiten, diese physikalische Adresse zu berechnen. Es ist zunächst wichtig, das Segment festzulegen, das für diese Adresse zuständig ist. Hier gibt es jedoch mehrere Möglichkeiten. So beginnt für Besitzer von Hercules-Videokarten der Videospeicher an der Segmentgrenze $B000, da dies die allererste Speicherstelle ist, an der Videoinformationen abgelegt werden können. $B000 multipliziert mit 16 ergibt $B0000. Ziehen wir diesen Wert von der obengenannten Adresse ab, so bleibt ein Offset von $8000 übrig. Also kann die physikalische Adresse $B8000 segmentiert dargestellt werden als $B000:$8000.

Dagegen beginnt für Besitzer von CGA-Karten der nutzbare Videospeicher erst an der Segmentgrenze $B800! Dieser Wert mit 16 multipliziert ergibt $B8000, was zu einem Offset von 0 führt. Somit ergibt sich als Adresse $B800:$0000. Dies wiederum bedeutet, daß beide Adressen das gleiche Byte beschreiben: $B000:$8000 ist mit $B800:$0000 identisch ebenso wie $B400:$4000 oder $B765:$09B0.

Halten wir daher fest, daß die Adressierung durch Segmentierung redundant ist, d.h. daß mehrere Adressen auf die gleiche physikalische Adresse zeigen können.

HINWEIS

Obwohl es nicht zwingend erforderlich ist, wird üblicherweise ein Segment immer auf eine Segmentgrenze gelegt, an der das erste nutzbare Byte liegt, das somit mit einem Offset 0 angesprochen werden kann. Falls z.B. ein Bereich angelegt werden soll, in dem Daten abgelegt werden, so ist es offensichtlich sehr sinnvoll, dieses sogenannte

Datensegment an der Segmentgrenze beginnen zu lassen, an der das erste Datenbyte liegt (um so optimal wie möglich zu programmieren, sollte daher das erste Datenbyte immer an einer Segmentgrenze liegen! Dies nennt man »die Daten ausrichten«, engl. *Data Alignment*).

1.2 Ports

Daten kann der Prozessor nicht nur aus seinem Speicher holen oder dort ablegen. Sondern er kann auch dadurch an Daten kommen, daß er mit seiner Umwelt, seiner Peripherie, kommuniziert. Denken Sie an die Tastatur oder an die Maus. Das sind »Geräte«, die nichts mit dem Speicher zu tun haben – und von denen er dennoch nicht weniger wichtige Daten erhält. Oder denken Sie daran, daß ja auch der Speicher mit Daten gefüllt werden muß. Woher? Von der Diskette, Festplatte oder einem CD-ROM-Laufwerk. Im Gegenzug muß der Speicher auch Daten wieder ausgeben können: auf dem Bildschirm, an die Laufwerke, ins Netzwerk, auf das Modem oder auf den Drucker.

All diese Beispiele zeigen, daß der Datenaustausch mit dem Speicher nur ein Teil der Kommunikationsmöglichkeiten des Prozessors ist – wenn auch ein zugegebenermaßen wichtiger. Doch wie kommuniziert er mit der mindestens genauso wichtigen Peripherie?

Klar ist, daß die Art der Daten, die im Spiel sind, die gleiche ist wie bei dem Datenaustausch mit dem Speicher: *Bytes* und *Words* oder andere Datenstrukturen. Also ist es nicht verwunderlich, wenn die Art, wie der Datenaustausch mit der Peripherie erfolgt, mit vergleichbaren Methoden erfolgt. Wenn ein Datenwort im Speicher in eine Variable eingetragen oder von dort ausgelesen werden soll, so muß diese Variable vom Typ *word* sein, also zwei *Bytes* aufnehmen können – oder mit anderen Worten: Es muß eine »Struktur« im Speicher geben, die zwei Adressen des adreßierbaren Bereiches belegt.

Dasselbe gilt auch bei der Kommunikation mit der Peripherie. Wenn ein Datum von der seriellen Schnittstelle abgeholt oder auf die parallele gelegt werden soll, so müssen die Schnittstelle und der Prozessor sich einig darüber sein, welche Datenstrukturen Verwendung finden werden und an welchen »Adressen« diese Daten stehen. Also muß es auch einen »Adreßraum« geben, der für den Datenaustausch mit der Umwelt zuständig ist.

Den gibt es auch! Neben dem Adreßraum, in dem der Speicher anzusiedeln ist, gibt es einen Adreßraum für die Peripherie. Dieser Adreßraum ist, verglichen mit dem Raum für den Speicher, sehr klein: Er umfaßt lediglich 64 kByte. Um diesen Raum ansprechen zu können, sind 16 »Adreßleitungen« nötig: 2^{16} = 65.768 = 64 k. Wenn also der

Prozessor ein Datum an die Peripherie ausgeben will, so muß er nur anstelle des Adreßraumes des Speichers den Adreßraum der Peripherie ansprechen. So einfach ist das!

Es erhebt sich nur die Frage: Wohin, also an welche Stelle in diesem zweiten Adreßraum muß der Prozessor das Datum schreiben, wenn er z.B. ein Zeichen ausdrucken will? Und wo kann er ein über das Modem eingetroffenes Datum abholen? Auch hier hilft ein Vergleich mit dem Speicher. Um gezielt und eindeutig Daten im Speicher abzulegen und von dort auch zuverlässig wieder zu bekommen, benutzt man in den Programmiersprachen Variablen. Das sind eigentlich nichts anderes als Adressen, die einen Namen bekommen haben. Wenn eine Variable in einem Programm definiert wird, so sucht sich der Prozessor nur einen freien, noch nicht belegten Platz, merkt sich die Adresse und reserviert diese Adresse sowie – je nach Größe des zu speichernden Datums – eventuell die nächsten folgenden. (Genauer gesagt tut das nicht der Prozessor, sondern der Interpreter oder Compiler der Programmiersprache – Der eben der Assembler, wenn wir schon von Assemblern reden.)

Warum also nicht auch bei der Kommunikation mit der Peripherie mit »Variablen« arbeiten? Genau das wird getan – nur heißen die dort nicht »Variablen«. Sie heißen *Ports*, nach dem englischen Wort für »Tor« (zur Außenwelt). Ein Port ist also nichts anderes als eine genau definierte Adresse in einem Adreßraum, an den der Prozessor Daten schreiben oder von wo er sie holen kann: mehr nicht. Sie sehen, Ports und Variablen haben eine Menge Gemeinsamkeiten. Sie repräsentieren beide Adressen, haben eine bestimmte Größe und sind über Adreßleitungen ansprechbar.

Aber sie haben auch Unterschiede. Denn die Lage einer Variable im Speicher ist nicht statisch! Wenn Sie im Programmcode vor der Definition einer Variable eine andere Variablendefinition einschieben, so verändert sich die Adresse der folgenden Variable. Das geschieht auch, wenn Sie eine vorangehende Definition löschen. Und auch das Betriebssystem hat ein kleines Wörtchen bei der Adreßvergabe mitzureden. Denn je nachdem, wie viele Programme oder Treiber vor dem eigentlichen Programm zu liegen kommen, kann sich die absolute Lage von Variablen im Speicher ändern. Dies ist ja auch der Grund dafür, daß das Starten eines Programms durch das Betriebssystem so relativ schwierig und umständlich erscheint. Denn ein Teil dieses Ladevorgangs ist es, die genaue Lage der »Datenbereiche« im Speicher festzulegen und dem Programm mitzuteilen.

Diese Variabilität macht beim Speicher Sinn. Denn er sollte möglichst effektiv ausgenutzt werden können, weshalb sich je nach Situation auch Adressen ändern können müssen – und zwar während der

Rechner läuft. Nicht so die Peripherie. Denn hier ist die Umgebung des Prozessors ja spätestens beim Booten des Rechners klar festgelegt – kaum jemand wird im Betrieb des Rechners sein Modem wechseln wollen oder den Drucker. (Das erklärt recht gut zwei Dinge, mit denen sich der Rechnerbesitzer auseinandersetzen muß: Wenn Hardware installiert wird, so muß der Rechner neu gebootet werden, damit die Peripherie korrekt angesprochen werden kann. Daher ist es nicht trivial, während des Rechnerbetriebs via Plug&Play Hardware nachrüsten zu können. Aber gehen wir nicht zu sehr ins Detail!)

Das alles heißt, daß man Adressen für Peripheriegeräte festlegen kann. Im Gegensatz zu Variablen repräsentieren Ports also festgelegte Adressen im Input/Output-Adreßbereich, sind also – wenn Sie so wollen – Konstanten. Jedes Peripheriegerät hat »seinen« Port, seine konstante Adresse. Es wird dann Probleme bekommen, wenn ein anderes Peripheriegerät ungefragt den gleichen Port, also die gleiche Adresse benutzt. (Womit wieder ein altbekanntes Problem erklärt wäre: I/O-Adressenkonflikt!)

Irgendwann einmal hat man sich darauf geeinigt, daß Ports bestimmte Größen haben können – und müssen. So gibt es Ports, über die *Bytes* übertragen werden können oder *words*, ja selbst *doublewords*. Dementsprechend sind solche Ports dann auch entweder 8 Bit groß oder eben 16 oder 32. Das ist ein weiterer Unterschied zu Variablen. Während mir bei der Programmierung niemand verbietet, die Variable *MeineVariable* nicht mehr auf ein Byte zeigen zu lassen, sondern auf ein *word*, weil sich herausgestellt hat, daß das Byte nicht ausreicht, so ist ein Port nach seiner einmaligen Definition ein für allemal festgelegt, weil sich die Hersteller der Peripheriegeräte und der Betriebssysteme darauf einstellen können müssen.

Fassen wir also zusammen: Der Prozessor kennt zwei Adreßräume. Der eine dient zum Austausch von Daten mit dem Speicher. Der andere wird von den Ports gebildet, über die der Prozessor mit Peripheriegeräten Daten austauschen kann. Ports sind in ihrer Größe und absoluten Lage in diesem Adreßraum eindeutig festgelegt.

Es ergibt sich eine ketzerische Frage: Wenn Ports nichts anderes als Adressen in einem Adreßraum sind, die nur deshalb Ports heißen, weil sie nicht den Adreßraum benutzen, der den Speicher betrifft – könnte man dann nicht auch »Ports« im »Speicheradreßraum« definieren? Ja, man kann. Und man tut das – Sie kennen alle so einen »Port«: Es ist der Bildschirmspeicher. Der Bildschirmspeicher ist ja nichts anderes als ein reservierter Bereich RAM, an genau festgelegter Adresse mit genau definierter Größe, auf den von der einen Seite der Prozessor, von der anderen die Videokarte zugreifen kann: ein klassischer »Port«. Da dieser Port aber eben nicht im Adreßraum für Ports angesiedelt ist, heißt er auch nicht so.

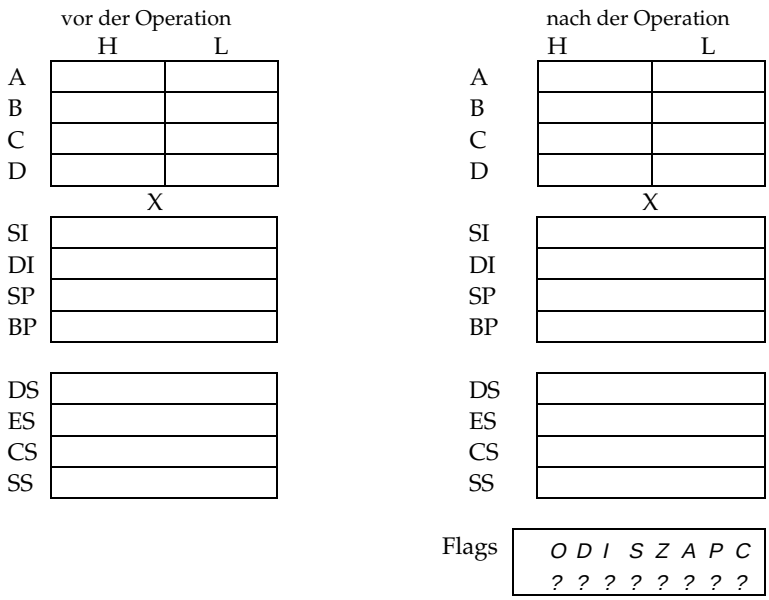
Das aber erklärt ganz selbstverständlich einen Begriff, der häufig herumgeistert und den viele nicht verstehen. Memory-Mapped I/O, auf den Speicher abgebildete Ein-/Ausgabe. Peripheriegeräte, die in der Lage sind, wie Speicher angesprochen zu werden, können über einen solchen Memory-Mapped I/O bedient werden. Beim Bildschirmspeicher macht das Sinn: Der ist viele *Bytes* groß (64 kByte beim monochromen Textmodus, bis zu mehreren MByte bei Grafiken mit True-Color-Darstellung). Bei der seriellen oder auch der parallelen Schnittstelle dagegen macht das keinen Sinn – aus technischen Gründen ist hier die Größe der Ports auf wenige *Bytes* beschränkt. (Andernfalls müßten Sie Ihren Drucker oder Ihr Modem mit oberschenkeldicken Kabeln verbinden! Na ja, fast ...)

Wann ist Memory-Mapped I/O sinnvoll? Immer dann, wenn von zwei Seiten auf einen gemeinsamen Speicher zurückgegriffen werden muß (z.B. Prozessor – Videokarte) und die Speicherinhalte wie zur Speicherung von Programmdaten eine gewisse Lebensdauer haben sollen. Dann kann man sich einen zweiten Speicher im Peripheriegerät schenken und den des Prozessors nutzen (was ja beim Bildschirmspeicher realisiert wurde). Wenn dagegen der Speicher, in den der Prozessor ein Datum schreibt oder aus dem er es liest, lediglich ein Puffer ist, aus dem das Peripheriegerät / der Prozessor seine Daten holt, um sie unmittelbar darauf weiterzuverarbeiten (Drucker, Netzwerk, Modem), so macht Memory-Mapped I/O keinen Sinn. Dann werden die Ports benutzt. Der Adapter für das Peripheriegerät hat dann die Pflicht, diesen Pufferspeicher bereitzustellen – und dafür zu sorgen, daß die I/O-Adresse, mit der er dann angesprochen werden kann, so eindeutig ist, daß es keinen Konflikt mit anderen Peripheriegeräten gibt. Der Prozessor stellt lediglich den Adreßraum und die Befehle zur Verfügung, um ihn zu nutzen!

Halten wir auch hier zusammenfassend fest: Memory-Mapped I/O ist eine Abart des »regulären« I/O, bei dem der »Port« in den Adreßraum des Speichers eingeblendet ist und wie dieser angesprochen wird (weshalb man eine Variable namens »Screen« definieren kann, ihr die absolute Adresse des betreffenden RAM-Bereichs zuordnen und dann über einen einfachen Zugriff auf diese Variable Daten austauschen kann). Bei dieser »Port-Version« kommen die gleichen Prozessorbefehle zum Einsatz, die auch beim Arbeiten mit Variablen verwendet werden. Demgegenüber steht die »reguläre« I/O-Methode, bei der zum Zugriff auf die Ports spezielle Prozessorbefehle eingesetzt werden.Der Adreßraum des 8086

2 Der Befehlssatz des 8086

Um die Wirkung der verschiedenen Prozessorbefehle zu verdeutlichen, wird an den Stellen, an denen es sinnvoll und zweckmäßig ist, das folgende Schema zur Illustration verwendet. Beachten Sie bitte, daß dabei jedoch nur die Register dargestellt werden, die wirklich betroffen sind. So hat es z.B. keinen Sinn, alle Register darzustellen, wenn nur das AL-Register betroffen ist.



Die vier Datenregister werden als 16 Bit breite Register dargestellt, die zweigeteilt sind. Links sind jeweils die Bits 15 bis 8 dargestellt, das sogenannte *höherwertige Byte* des Worts, während rechts das *niederwertige Byte*, also die Bits 7 bis 0 verzeichnet sind.

Das Schema stellt auf der linken Seite die Situation vor und auf der rechten die nach der Operation dar. Daher werden auch nur auf der rechten Seite die Flags verzeichnet. Ein dort eingetragenes »*« bezeichnet die Veränderung des entsprechenden Flags in Abhängigkeit von der Operation. » « heißt, daß dieses Flag nicht beeinflußt wird, und »?« bedeutet, daß es zwar verändert wird, jedoch keine Rückschlüsse oder Auswertungen möglich sind, weil es intern vom Prozessor für eigene Zwecke verwendet wird. Man sagt dazu: »Der Zustand des Flags ist *undefiniert*.« »0« und »1« stehen für explizites Löschen bzw. Setzen des Flags.

Im folgenden werden wir sehr viel mit Operanden zu tun haben, also mit Parametern, die einzelne Befehle benötigen. Um dies so allgemein wie möglich zu halten, werden wir bei der Beschreibung der Operationen folgende Abkürzungen benutzen:

- *mem*, wenn eine Speicherstelle irgendwo im RAM des Rechners gemeint ist. *mem* selbst steht für eine beliebige allgemeine Speicherstelle, ohne daß ihre Größe genauer spezifiziert werden soll. Soll jedoch mit einer solchen Speicherstelle auch gleichzeitig ausgedrückt werden, von welchem Datentyp der enthaltene Wert sein soll, so werden wir die Größe der Variablen in Bits anhängen, also z.B. *mem8* für Bytes, *mem16* für Worte etc.
- *reg*, wenn ein Register des Prozessors gemeint ist. Auch hier wird unterschieden zwischen *reg8*, wenn die Byte-Teile der Register angesprochen werden sollen, und *reg16* bei den Wortregistern. Natürlich steht auch hier *reg* für ein allgemeines Register ohne Angabe des Datentyps.
- *const*, wenn ein direkter Zahlenwert gemeint ist, also z.B. 4711. Es gilt: *const8* für Bytes, *const16* für Worte etc. Auch hier gilt: *const* ist eine beliebige Konstante.
- *adr*, wenn ein Ziel angegeben werden soll. Man unterscheidet hierbei zwei Arten: sogenannte *Short Addresses* und *Long Addresses*. Für den Programmierer spielt diese Unterscheidung eine untergeordnete Rolle, da er üblicherweise nur den Namen eines Labels im Programmcode als Ziel angibt. Wir werden diese Zusammenhänge im zweiten Teil des Buchs klären.
- *segreg*, wenn ein Segmentregister betroffen ist. Es gibt jedoch nur sehr wenige Operationen, die zwischen 16-Bit-Registern und Segmentregistern unterscheiden. Für diese wenigen Befehle gilt diese Abkürzung.

ACHTUNG

Die Tatsache, daß hier von allgemeinen Registern, Speicherstellen und Konstanten geredet wird, die keine bestimmte Größe haben, heißt nicht, daß Sie solche Operanden auch tatsächlich assemblieren können! Es soll hiermit lediglich der Tatsache Rechnung getragen werden, daß sich bestimmte Operationen mit verschiedenen Datentypen durchführen lassen. Um nicht immer »Operation reg8,reg8 oder Operation reg16,reg16 oder Operation reg32,reg32« schreiben zu müssen, wurde dieser »allgemeine Datentyp« eingeführt. Er lautet dann einfach »Operation reg,reg«. Beim Programmieren dagegen müssen Sie unterscheiden! Aber es werden noch andere Akürzungen gewählt, die in Beispielen zur Geltung kommen:

- *ByteVar*, wenn eine Byte-Variable gemeint ist
- *WordVar*, wenn eine Wortvariable benutzt werden soll
- *Label*, wenn eine Zieladresse angesprungen werden soll

Statt *ByteVar* und *WordVar* könnten Sie natürlich auch Variablennamen wählen.

2.1 Ein- und Ausgabeoperationen

Wenn der Prozessor arbeiten soll, braucht er neben den Befehlen auch Daten, die er bearbeiten kann. Schauen wir uns daher zunächst einmal an, wie er an Daten herankommt. Zur Verfügung stehen ihm hierzu die Ein- und Ausgabeoperationen, die sich einteilen lassen in:

- Operationen, die den Zugriff auf eine Speicherstelle vorbereiten (LDS, LES, LEA)
- Zugriffsoperationen auf Speicherstellen (MOV, XCHG, XLAT, XLATB)
- temporäres Zwischenspeichern im »Arbeitsbereich« des Prozessors (PUSH, POP)
- Operationen zum Datenaustausch mit der Peripherie (IN, OUT)

Die I/O-Operationen (für *Input/Output* also Ein- und Ausgabe) beschränken sich hierbei nicht auf einzelne Werte. Vielmehr können ganze Strukturen bearbeitet werden. Die hierzu notwendigen Befehle werden wir jedoch in einem der folgenden Abschnitte betrachten, wenn es um »Strings« geht.

Überlegen wir zunächst, wie der Prozessor auf Daten zugreifen soll. Daten werden sinnvollerweise in einem eigenen Segment aufgehoben[1]. Dieses Segment heißt dann auch sehr sinnig »Datensegment«, und seine Adresse wird in einem bestimmten Segmentregister, dem Data Segment Register (DS) aufgehoben. Daraus können Sie ersehen, daß der Verwaltung von Daten eine spezielle Bedeutung beigemessen wird.

Der Prozessor kann am Inhalt des DS-Registers feststellen, in welchem Segment des RAMs die Daten zu suchen sind. Segmente können jedoch bis zu 65.536 Bytes groß sein. Wo also befindet sich das gewünschte Datum?

---

[1] Es gibt auch Gründe, bestimmte Daten nicht in einem eigenen Segment aufzuheben. Das sind dann aber Spezialfälle, die im zweiten Teil des Buches ausführlich besprochen werden.

LEA

Der Befehl LEA, *Load Effektive Address,* dient dazu, dem Prozessor den Ort mitzuteilen, an dem sich das gesuchte Datum befindet. LEA erwartet als Operanden ein 16-Bit-Register, also z.B. AX, BX oder SI. Die Adresse, genauer der Offsetanteil der Adresse des zweiten Operanden, wird in dieses Register eingetragen. Allgemein lautet der Befehl somit:

LEA reg16, adr.

So findet sich z.B. nach

```
...
lea       si,WordVar
...
```

der Offset der Variable *WordVar* in SI wieder. Diese Angabe ist nicht auf Variablen im Datensegment beschränkt. Es läßt sich z.B. durch

```
Marke1:   ...
          ...
          ...
          lea   bx,Marke1
          ...
```

auch der Offset einer Stelle im Codesegment ermitteln.

Üblicherweise besitzen die Befehle, die Daten manipulieren, einen Operanden, mit dem man das Ziel der Manipulation angeben kann. In *CMP AX,WordVar* beispielsweise wird durch die Angabe *WordVar* automatisch der Offset von *WordVar* berechnet und dem Befehl übergeben. Wozu also dient LEA? LEA kommt immer dann zum Einsatz, wenn indirekt adressiert werden soll, also der Offset der Adresse nur ein Teil der tatsächlich gewünschten Adresse ist. Beispiele hierzu werden wir ausgiebig im zweiten Teil des Buches besprechen. Merken wir uns also lediglich, daß es einen Befehl gibt, mit dem man ein Register mit dem Offset der Adresse eines Ziels beladen kann.

ACHTUNG

Beachten Sie bitte, daß LEA nur den Offsetanteil der Adresse ermittelt, nicht die gesamte Adresse! Für eine vollständige Adresse ist aber auch der Segmentanteil erforderlich. Falls Sie im letzten Beispiel mit LEA in BX den Offest von *Marke1* laden, heißt das nicht, daß auch der Inhalt eines der Segmentregister verändert wird! Ein häufiger Fehler besteht darin, mit Befehlen wie LEA eine vermeintlich korrekte, vollständige Adresse zu ermitteln!

LDS, LES

LDS und LES sind zwei Befehle, die vollständige Adressen laden können. Analog zu LEA erwarten diese Befehle ein 16-Bit-Register, in das der Offsetanteil der Adresse geladen werden kann. Der Segmentanteil wird bei LDS in DS und bei LES in ES geschoben.

ACHTUNG Anders als bei LEA kann man jedoch mit diesen Befehlen kein direktes Laden einer Adresse durchführen! Der Operand von LDS und LES ist nämlich eine Speicherstelle, an der eine vollständige Adresse steht, die dann geladen werden soll! Bei einer Assemblerzeile wie

```
LDS     SI, WordVar
```

wird nicht, wie man im ersten Moment glauben könnte, der Prozessor angewiesen, den Offsetanteil der Adresse von *WordVar* in SI und den Segmentanteil in DS zu laden! Vielmehr soll der Prozessor veranlaßt werden, an der Stelle, die durch *WordVar* bezeichnet wird, einen Offset und ein Segment zu holen und in die spezifizierten Register DS und SI zu laden! Also wird der Assembler hier eine Fehlermeldung erzeugen. Denn *WordVar* ist per definitionem eine Marke, die auf ein Wort zeigt. Der Prozessor jedoch benötigt ein Doppelwort für eine vollständige Adresse!

LDS und LES dienen zum indirekten Laden einer Adresse. Sie werden häufig bei Routinen eingesetzt, denen ein Zeiger auf eine Speicherstelle übergeben wird. Halten wir fest: Soll die Adresse eines Datums direkt bestimmt werden, so müssen wir mit LEA ihren Offset bestimmen! Das dazugehörige Segment ist klar: Es kann nur das Codesegment sein, wenn Labels betrachtet werden oder Segmente, deren Adressen schon in DS oder ES stehen!

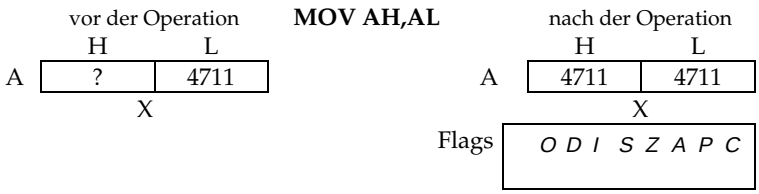
Steht dagegen an irgendeiner Speicherstelle eine Adresse, die auf ein Datum zeigt, was verändert werden soll, so müssen wir indirekt adressieren, wozu zunächst die vollständige Adresse in die Registerkombination DS:reg oder ES:reg geladen werden muß, was LDS oder LES durchführen.

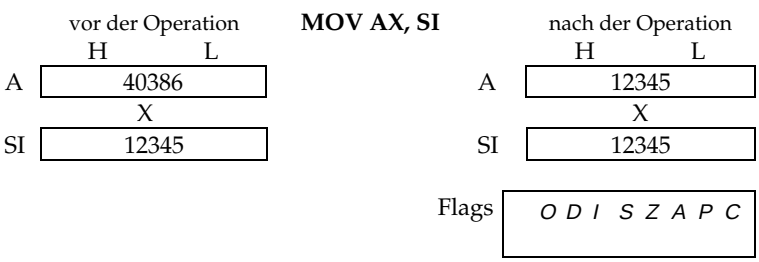
MOV Wohl der wichtigste Befehl des gesamten Prozessorbefehlssatzes ist der Befehl MOV, das für *move* steht. Mit diesem Befehl kann man Daten aus einer Speicherstelle in die Register des Prozessors laden, Daten innerhalb der Register verschieben und wieder an Speicherstellen schreiben. Ohne diesen Befehl ist kein sinnvolles Arbeiten des Prozessors möglich. Dementsprechend vielfältig sind die Möglichkeiten, die man mit MOV hat:

- `MOV reg, reg` z.B.
  ```
  MOV AX, BX
  MOV CL, DH
  ```
- `MOV reg, mem` z.B.
  ```
  MOV SI, WordVar
  MOV DH, ByteVar
  ```
- `MOV mem, reg` z.B.
  ```
  MOV WordVar, DX
  MOV ByteVar, AL
  ```

- MOV reg, const z.B. MOV CX, $4711
  MOV BH, $FF
- MOV mem, const z.B. MOV WordVar, $0815
  MOV ByteVar, $AB

Sehen wir uns dies einmal im Schaubild an. Im ersten Fall wird ein Wert innerhalb eines 16-Bit-Registers vom niedrigen Byte ins hohe kopiert, im zweiten Fall aus einem Indexregister in ein 16-Bit-Register.





ACHTUNG

Nach einer Konvention, die der Prozessorhersteller Intel eingeführt hat, folgt dieser Befehl der allgemeinen Struktur:

Befehl Ziel, Quelle

Das heißt, daß der erste Operand eines Befehls das Ziel der Aktion angibt und der zweite die Quelle (Dies ist auch der Grund, warum es ein *MOV 4711, WordVar* nicht geben kann!). Das ist insbesondere deshalb wichtig, da MOV Daten nicht verschiebt, wie man anhand des Namens annehmen könnte. Vielmehr kopiert MOV den Inhalt aus der Quelle an das Ziel. Dieser Vorgang ist *nicht kommutativ*, das heißt, man erhält unterschiedliche Ergebnisse, wenn man die Reihenfolge der Operanden vertauscht. Ein *MOV AX, WordVar* kopiert daher den Inhalt von *WordVar* in AX, während *MOV WordVar, AX* umgekehrt den Inhalt von AX in *WordVar* kopiert. In beiden Fällen wird der ursprüngliche Inhalt überschrieben.

XCHG

Möchte man Daten nicht von der Quelle an das Ziel kopieren, sondern deren Inhalte vertauschen, so kommt XCHG, *Exchange*, zu Einsatz. Es arbeitet wie MOV, mit der Ausnahme, daß sich nach erfolgreicher

Operation der Inhalt der Quelle am Ziel und der Inhalt des Ziels in der Quelle befindet. XCHG arbeitet *kommutativ*, also hat XCHG AX, SI das gleiche Ergebnis wie XCHG SI, AX!

XLAT, XLATB Eine weitere Möglichkeit, an Daten zu kommen, bietet der Befehl XLAT. Mit diesem Befehl, der für *Translate* steht, können Daten aus Tabellen gelesen werden. XLAT gehört zu den 8086-Befehlen, die nur von wenigen Assemblerprogrammierern eingesetzt werden. Der Befehl scheint recht unbekannt zu sein, was sehr schade ist, da er sehr leistungsfähig ist und bequem für die verschiedensten Dinge benutzt werden kann – unter anderem, wie der Name schon sagt, auch für »Übersetzungen« von Bytes.

XLAT kennt nur einen Operanden. Das heißt aber nicht, daß der Befehl nicht mehrere benutzt! Vielmehr sind fast alle Operanden fest vorgegeben und können nicht verändert werden. So befindet sich in AL ein sogenannter *Index* auf ein Feld, das maximal 256 Bytes groß sein kann. Der Offset dieses Feldes wird in BX erwartet. Hier haben wir also das erste Beispiel dafür, daß die Rechenregister des 8086 nicht ganz gleichberechtigt sind. Im Akkumulator AL befindet sich ein Index, im Basisregister BX der Offset einer Adresse. Ferner zeigt sich auch hier schon eine erste Anwendung von LEA: Der Offset des Feldes kann über LEA in BX geladen werden.

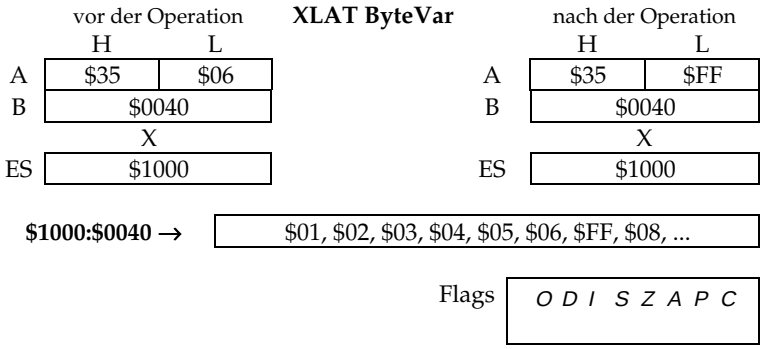
Das führt aber, wie wir eben gesehen haben, zu Problemen. Wir haben nun zwar einen Offset auf eine Tabelle; aber in welchem Segment steht diese? Denn wenn BX z.B. mit LEA geladen werden kann, woher kennt der Prozessor dann das dazugehörige Segment? Die Antwort lautet: durch den übergebenen Parameter! Denn XLAT hat die Form:

```
XLAT  mem8.
```

Der Parameter kann also explizit eine Variablenadresse übernehmen. Und dementsprechend bezieht XLAT den in BX stehenden Offset der Adresse auf das Segment, in dem die Variable steht.

ACHTUNG Obwohl mit *mem8* eine vollständige Variablenadresse übergeben wird, muß dennoch in BX der Offset von *mem8* verzeichnet sein, denn XLAT extrahiert aus *mem8* nur den Segmentanteil. Daher stellen Sie sich *XLAT mem8* am besten als *xS:XLAT* vor, wobei *xS* dasjenige der Segmentregister ist, in dem der Segmentanteil von *mem8* steht, also z.B. ES:XLAT oder CS:XLAT. Es kann hier also zu Inkompatibilitäten kommen, wenn BX nicht der Offsetanteil von *mem8* ist: Dann nämlich setzt XLAT die Adresse aus dem Segmentanteil von *mem8* und dem Offsetanteil aus BX zusammen – und dies kann irgendwelchen Unsinn ergeben!

Was macht nun XLAT? Zunächst wird zum Offset in BX der Index in AL addiert. Dies geschieht nur intern, das heißt, daß die Werte in den Registern nicht verändert werden. Mit der so berechneten Adresse wird dann praktisch ein *MOV AL,ByteVar* ausgeführt, das heißt, AL wird mit dem Byte überschrieben, auf das AL zeigt. Ergebnis: In AL steht abschließend das *AL-te* Byte aus der Tabelle, deren Adresse bei BX beginnt.



XLATB ist lediglich eine Variante von XLAT und macht haargenau das gleiche wie XLAT. Einziger Unterschied: XLATB besitzt keinen Operanden, als Segmentregister dient immer DS. XLATB ist somit das gleiche wie DS:XLAT.

PUSH dient zur kurzfristigen Zwischenspeicherung eines Register- /Variableninhalts in einer Struktur, die üblicherweise *Stapel* oder besser *Stack* genannt wird. Diese Struktur ist so etwas wie der lokale Zwischenspeicher, in dem der Prozessor Daten ablegen und verwalten kann. Auf den Stack werden wir im zweiten Teil des Buches genauer eingehen. Merken wir uns daher für den Moment nur, daß man mit **PUSH**

PUSH reg16

den Inhalt eines 16-Bit-Registers »auf dem Stapel ablegen kann«, während man mit

PUSH mem16

das gleiche mit dem Inhalt von Wortvariablen tun kann. Durch diesen Befehl werden keine einzelnen Bytes kopiert: Es können immer nur ganze Worte sein.

POP holt das oberste Wort auf dem Stapel wieder zurück. Auch hier kann mit **POP**

POP reg16

ein Wortregister als Ziel angegeben werden, aber auch eine Wortvariable:

```
POP     mem16
```

IN, OUT

IN und OUT sind die letzten Befehle, über die der Prozessor an Daten gelangen kann (ausgenommen die Stringbefehle). Mit IN und OUT kann der Prozessor Ports ansprechen, also Daten mit der Peripherie austauschen. Den Befehlen wird dazu als Operand der Port angegeben, der angesprochen werden soll:

```
IN      AL, Port
```

holt Daten von einem Port und legt sie in AL ab, während über

```
OUT     Port, AL
```

der Inhalt von AL auf den Port ausgegeben werden kann (Bitte beachten Sie auch hier die Konvention »Befehl Ziel, Quelle«; der erste Operand nennt immer das Ziel, der zweite die Quelle.). Neben dem byteweisen Zugriff ist auch der wortweise Zugriff möglich, indem als beteiligtes Register AX verwendet wird: IN, AX, Port und OUT Port, AX.

Wir haben hier ein zweites Beispiel für die Spezialisierung der Rechenregister: Das Einlesen und Ausgeben von Bytes in/von einem Port ist nur über den Akkumulator AL bzw. AX möglich!

Doch noch ein Wort zu der Portadresse. Die Portadressen können im Bereich $0000 bis $FFFF liegen. Dieser Bereich läßt sich (nicht ganz willkürlich!) aufteilen in den Bereich $00 bis $FF sowie in $0100 bis $FFFF. Im ersten Bereich finden sich Ports, die recht häufig angesprochen werden müssen. Für sie wurde ein Spezialfall der IN/OUT-Befehle reserviert, bei dem man die Portadresse als Konstante angeben kann, also

```
IN      AL, const8      z.B.    IN      AL, 060h
OUT     const8, AL              OUT     041h, AL
```

Beachten Sie bitte, daß hier der Inhalt von AL/AX nicht an eine Konstante übergeben wird, sondern an einen Port, der direkt über eine Konstante angegeben werden kann!

Für den Rest steht der »normale« Befehl zur Verfügung, bei dem die Portadresse in DX stehen muß:

```
IN      AL, DX
OUT     DX, AL
```

Beachten Sie bitte auch hier, daß es nur diese zwei Anweisungen gibt. Sie können also den gewünschten Port nur in DX angeben und Daten nur via AL/AX austauschen!

2.2 Vergleichsoperationen

Was aber kann der Prozessor mit den Daten tun, die er sich mit Hilfe der eben besprochenen Befehle beschaffen kann? Er kann sie z.B. vergleichen. Hierzu kennt der Prozessor zwei Befehle:

- arithmetischer Vergleich (CMP)
- logischer Vergleich (TEST) zweier Werte

Streng genommen kennt der Prozessor sogar noch ein paar Befehle mehr, jedoch werden wir auch diese erst im Zusammenhang mit den Stringoperationen kennenlernen.

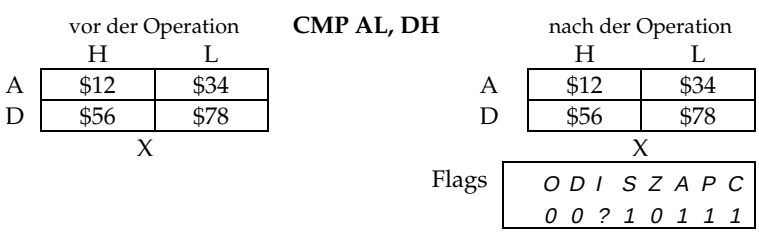
CMP

Der Befehl CMP vergleicht zwei Operanden miteinander. Dieser Befehl ist recht flexibel, da dadurch ein Vergleich zweier Register, eines Registers mit einer Speicherstelle im RAM, eines Registers mit einer Konstanten oder einer Speicherstelle mit einer Konstanten möglich ist. Es ist möglich, vollständige 16-Bit-Worte zu vergleichen, aber auch lediglich 8-Bit-Anteile der Register und/oder Variablen.

Die Anweisungen hierzu lauten:

- CMP reg, reg z.B. CMP AX, BX
CMP CL, DH
- CMP reg, mem z.B. CMP SI, WordVar
CMP DH, ByteVar
- CMP mem, reg z.B. CMP WordVar, DX
CMP ByteVar, AL
- CMP reg, const z.B. CMP CX, $4711
CMP BH, $FF
- CMP mem, const z.B. CMP WordVar, $0815
CMP ByteVar, $AB

Auch hier ein Beispiel:



Der CMP-Befehl arbeitet wie eine Subtraktion, bei der der zweite Operand vom ersten abgezogen wird, weshalb man ihn auch »arithmetischen Vergleich« nennt. Das Ergebnis dieser Subtraktion wird ausgewertet, indem bestimmte Flags verändert werden. Abschließend wird das Ergebnis der Rechnung verworfen, d.h. daß alle Register und Speicherstellen unverändert bleiben; lediglich das Flagregister kann Auskunft über das Ergebnis des Vergleichs geben.

ACHTUNG Die Reihenfolge, in der die Operanden angegeben werden, ist nicht egal! Wie bei jeder Subtraktion ist das Ergebnis davon abhängig, ob ein größerer Wert von einem kleineren abgezogen wird oder umgekehrt. So liefert der Vergleich *CMP 5, 3* natürlich ein anderes Ergebnis als *CMP 3, 5*.

In Abhängigkeit vom Ergebnis werden die Flags wie folgt gesetzt:

- *Zero-Flag*, wenn das Subtraktionsergebnis 0 war. Hierbei spielt es keine Rolle, ob die Operanden vorzeichenlos oder vorzeichenbehaftet interpretiert werden.
- *Carry-Flag*, wenn der erste Operand kleiner war als der zweite, das Subtraktionsergebnis also kleiner als 0 war. Das Carry-Flag bezieht sich aber nur auf vorzeichenlose Zahlen, bei denen Bit 7 bzw. Bit 15 als Teil der Zahl, nicht etwa als Vorzeichen zu interpretieren sind.
- *Overflow-Flag*, wenn der erste Operand kleiner war als der zweite. Der Unterschied zum Carry-Flag ist der, daß das Overflow-Flag einen Überlauf vorzeichenbehafteter Zahlen anzeigt, Bit 15 bzw. 7 also als Vorzeichen betrachtet werden.
- *Auxiliary-Flag*, wenn ein Überlauf bei BCDs stattgefunden hat.
- *Parity-Flag*, wenn das Ergebnis der Subtraktion eine gerade Anzahl an Bits lieferte, also z.B. bei 0, 3, 5 und 6, nicht aber bei 1, 2, 4 und 7 etc.
- *Sign-Flag*, wenn entweder 16 Bit breite Zahlen verglichen wurden und Bit 15 gesetzt ist oder wenn beim Vergleich von 8 Bit breiten Zahlen Bit 7 gesetzt ist.

Alle anderen Flags bleiben unverändert (es sind ja auch nur noch drei, die mit Arithmetik nichts am Hut haben!).

Das *Parity-Flag* ist ein sehr unbedeutendes Flag. Es spielt eine so untergeordnete Rolle, daß man es bei der Besprechung glatt vergessen könnte. Oder was für eine Bedeutung würden Sie der Information beimessen, daß im Ergebnis eine gerade Anzahl von Bits gesetzt ist, ohne zu erfahren, wie viele und welche?

Tatsächlich spielt das Paritiy-Flag im wesentlichen auch nur dann eine Rolle, wenn eine Prüfsumme im Spiel ist. Prominentes Beispiel hierfür ist die Datenkommunikation, bei der die zu versendenden oder zu empfangenden Daten häufig mit einem *Prüfbit* zusammen verschickt werden. Dieses Prüfbit gibt dann genau an, ob eine gerade oder ungerade Anzahl von Bits im übertragenen Byte gesetzt ist. Auf diese Weise wird eine einfache, aber wirkungsvolle Prüfung auf korrekte Übertragung erreicht.

Stellen Sie sich vor, es wird ein Byte empfangen. Dies kommt irgendwie in eines der Register des Prozessors, sagen wir in AL. Mit der einfachen Überprüfung der Parität des Bytes über das Parity-Flag können wir nun den Programmablauf steuern. Ist das Flag gesetzt, ist also eine gerade Anzahl von Bits im Byte vorhanden, so springen wir an eine Stelle im Programm, die nur noch überprüft, ob das mitverschickte Prüfbit ebenfalls gesetzt war. Ist das der Fall, ist alles in Ordnung. Andernfalls hat bei der Übertragung ein Fehler stattgefunden. Ist das Parity-Flag dagegen nicht gesetzt, so machen wir an anderer Stelle im Programm weiter. Auch hier erfolgt eine Überprüfung des mitgesendeten Prüfbits; allerdings ist diesmal nur dann alles in Ordnung, wenn dieses auch nicht gesetzt ist. Sie sehen, sehr viel ist mit dem Parity-Flag nicht los, weshalb wir es mit dieser Exkursion auch als besprochen ansehen wollen und uns nicht mehr darum kümmern werden!

HINWEIS

Erfahrungsgemäß macht es etwas Schwierigkeiten, sich mit den Flags bei arithmetischen Operationen zurechtzufinden. Daher soll hier eine kurze Erklärung folgen, die für alle arithmetischen Operationen gilt, also auch für CMP als verkappte Subtraktion.

Wenn der Programmierer einen Wert in ein Register schreibt, weiß er üblicherweise ziemlich genau, was für ein Wert dies ist. So ist es ganz klar, daß die Bitfolge 0000 1001 ein Byte mit dem Wert 9 ist. Auch das Eintragen des Bytes 133 wird den Programmierer vor kein großes Problem stellen. So ist offensichtlich, was passiert, wenn man 9 mit 133 vergleicht: Das Ergebnis ist -124 und somit kleiner als 0. Es findet also ein Überlauf statt, und das Carry-Flag wird gesetzt.

Andererseits erwarten wir natürlich keinen Überlauf und somit ein gelöschtes Carry-Flag, wenn wir 9 mit -123 vergleichen, denn die Subtraktion von -123 von 9 liefert 132, was größer als 0 ist. »Alles schön und gut«, werden Sie nun sagen, »aber was soll das?« Ganz einfach: 133 ist, binär ausgedrückt, 1000 0101. Genau diese Bitfolge hat aber auch die Zahl -123. In beiden Fällen wird also die Bitfolge 0000 1001 mit der Bitfolge 1000 0101 verglichen. Wir erwarten aber in beiden Fällen unterschiedliche Ergebnisse.

Damit stellen wir den Prozessor vor ein Problem. Wie soll er wissen, daß wir im einen Fall die vorzeichenlose Zahl 9 mit der vorzeichenlosen Zahl 133, im anderen Fall aber die vorzeichenbehaftete Zahl 9 mit der vorzeichenbehafteten Zahl -123 vergleichen? Die Antwort heißt: Er kann es nicht wissen! Daher setzt er auch alle Bits, die in solchen Fällen betroffen sind, in entsprechender Weise. Hat der Prozessor die beiden Zahlen als vorzeichenlos zu interpretieren, so findet ein Überlauf statt. Dies zeigt das gesetzte Carry-Flag an. Andererseits findet bei vorzeichenbehafteter Interpretation kein Überlauf statt. Das Overflow-Flag bleibt daher gelöscht. Fehlt noch das Sign-Flag, da dies anzeigt, ob ein Underflow oder ein Overflow stattgefunden hat. Schließlich kann ja das Ergebnis negativ oder positiv sein. In unserem Falle bleibt auch das Sign-Flag gelöscht, da das Subtraktionsergebnis positiv ist.

Und was macht das Auxiliary-Flag? Der Prozessor kennt neben vorzeichenlosen und vorzeichenbehafteten Zahlen auch noch BCDs. Wenn Sie nicht wissen, was das ist, schauen Sie bitte in den Anhang! BCDs können definitionsgemäß nur maximal vier Bits eines Bytes beanspruchen, nämlich die Bits 3 bis 0 oder das untere Nibble des Bytes, wie man häufig sagt. Bei BCDs müssen die Bits 7 bis 4, also das obere Nibble, immer gelöscht sein!

Woher soll aber der Prozessor wissen, daß die Bitfolge 0000 1001, die z.B. in AL steht, die Binärzahl 9 (egal ob vorzeichenbehaftet oder nicht) oder die gültige BCD-Ziffer 9 ist? Auch hier lautet die Antwort: Er weiß es nicht, berücksichtigt aber neben den eben geschilderten Fällen auch den Fall, daß es sich tatsächlich um eine BCD-Ziffer handeln *könnte*!

Setzt nun irgendeine arithmetische Operation irgendein Bit im oberen Nibble des Bytes, so muß offensichtlich ein Überlauf stattgefunden haben, falls der Registerinhalt als BCD zu interpretieren ist. Genau das zeigt das Auxiliary-Flag an.

ACHTUNG Wenn BCDs verglichen werden sollen, achten Sie bitte darauf, daß tatsächlich nur Byte-Register wie AH, AL, BH, BL etc. verwendet werden. Die gleichzeitige Überprüfung von zwei BCD-Ziffern in einem Wortregister funktioniert nicht so wie erwartet. Der Vergleich von $0606 in AX mit z.B. $0009 in BX über *CMP AX,BX* setzt zwar das Auxiliary-Flag ebenso wie der Vergleich von $0606 mit $0909, allerdings ist nach dem Vergleich von $0606 mit $0900 das Flag gelöscht. Dies zeigt, daß Wortregister wie Byte-Register behandelt werden: Die »obere« BCD-Ziffer wird schlichtweg ignoriert. Aber darauf verlassen sollten Sie sich dennoch nicht, da die einzelnen Chip-Hersteller verschiedenen Konventionen folgen können.

TIP Denken Sie daher bei allen arithmetischen Operationen daran: Der Prozessor *kann* nicht wissen, ob die Zahlen, die er manipuliert, vorzeichenlos oder vorzeichenbehaftet gemeint sind oder vielleicht sogar BCDs darstellen. Er führt daher quasi für alle drei Fälle die Operation durch und setzt die entsprechenden Flags. Zuständig für vorzeichenlose Zahlen ist das Carry-Flag, für vorzeichenbehaftete das Overflow-Flag und für BCDs das Auxiliary-Flag, um einen Überlauf anzuzeigen. Welches Flag nun das im jeweiligen Fall richtige ist, hat der Programmierer zu entscheiden, da nur er weiß, was für Zahlen im Spiel sind und wie das Ergebnis zu interpretieren ist. Das Sign-Flag ist nur bei vorzeichenbehafteten Binärzahlen definiert, da BCDs definitionsgemäß kein Vorzeichen haben.

HINWEIS In der Regel werden Sie nur mit dem Carry-Flag, dem Overflow-Flag und dem Sign-Flag arbeiten. In der überwiegenden Zahl der Fälle wird Sie die Stellung dieser drei Flags auch nicht interessieren, da üblicherweise nach Vergleichen wie CMP, aber auch nach anderen arithmetischen Operationen Befehle folgen, die ihrerseits die Flags auswerten. Dies können bedingte Sprünge sein, Korrekturen nach Additionen etc. oder auch die Korrektur bei der Manipulation von BCDs. Dennoch ist es für das Verständnis wichtig zu wissen, was passiert.

TEST Der zweite Vergleich, den der Prozessor kennt, ist der »logische Vergleich« TEST, dessen Ergebnis durch eine logische »Und«-Operation entsteht. Dabei werden alle Bits des einen Operanden mit den Bits des anderen »und«-verknüpft (was das heißt, werden wir im Abschnitt über die logischen Operationen noch kennenlernen). Auch hier wird das Ergebnis dieser Verknüpfung verworfen, wenn die Flags korrekt gesetzt wurden, so daß auch bei diesem Vergleich die Operanden unverändert bleiben.

Bei TEST spricht man üblicherweise nicht von zwei Operanden, obwohl die Parameter von TEST natürlich Operanden sind. TEST verknüpt vielmehr einen Prüfwert (Operand 1) mit einer Prüfmaske (Operand 2). Etwa nach dem Motto: »Schau nach, ob im Bitfeld *WortVar* die Bits 9, 4 und 3 gesetzt sind«. *WortVar* ist dann der Prüfwert, die Bitfolge 0000 0010 0001 1000, also der hexadezimale Wert $0218, die Prüfmaske.

TIP Bei der logischen Operation TEST ist die Reihenfolge der Operanden egal. Im Gegensatz zu CMP liefert *TEST Prüfwert,Prüfmaske* das gleiche Ergebnis wie *TEST Prüfmaske,Prüfwert*. Dennoch ist es vor allem für den Anfänger sehr hilfreich, die oben genannte Reihenfolge *Prüfwert,Prüfmaske* einzuhalten – um so mehr, als die TEST zugrunde-

liegende logische Operation AND hier anders reagiert. Bei ihr spielt, wie wir weiter unten noch sehen werden, die Reihenfolge sehr wohl eine Rolle.

Ebenso wie CMP kann man TEST sehr flexibel einsetzen:

- TEST reg, reg z.B. TEST AX, BX
  TEST AL,AH
- TEST reg, mem z.B. TEST CX,WordVar
  TEST AH, ByteVar
- TEST mem, reg z.B. TEST WordVar, DX
  TEST ByteVar, AL

  Bemerkung: Dies führt zum gleichen Ergebnis wie *TEST reg, mem*, wie man leicht einsieht: Die Bitverknüpfung ist die gleiche, Register und/oder Speicherstellen werden nicht verändert, das Ergebnis steht beide Male in den Flags.
- TEST reg, const z.B. TEST CX, $FFFF
  TEST BH, $0F
- TEST mem, const z.B. TEST WordVar, $0707
  TEST ByteVar, $FF

Auch bei diesem Vergleich werden Flags verändert, und zwar wie folgt:

- *Zero-Flag*: wenn nach der Und-Verknüpfung alle Bits 0 sind
- *Parity-Flag*, wenn das Ergebnis eine gerade Anzahl von Bits aufweist
- *Sign-Flag*, wenn Bit 15 beim Testen von Wortregistern bzw. Bit 7 beim Testen vom Byte-Registern gesetzt ist
- *Carry-Flag* und *Overflow-Flag* werden definitiv und immer gelöscht.

Alle anderen Flags bleiben unverändert.

ACHTUNG Sehr im Gegensatz zu den Behauptungen, die in anderen Abhandlungen und Referenzen zum Assembler stehen, spielen bei TEST die Flags *Carry*, *Overflow* und *Auxiliary* keine Rolle! Das ist auch ganz in Ordnung so. Die genannten Flags sollen ja anzeigen, daß bei arithmetischen Operationen der maximal für die betreffende Zahl darstellbare Bereich über- oder unterschritten wurde, das Resultat also nicht mehr korrekt dargestellt werden kann.

Bei logischen Operationen dagegen kann es keinen Überlauf geben. Entweder ist eine Behauptung wahr oder sie ist es nicht. Entweder ist ein Bit gesetzt oder es ist gelöscht. Die entsprechenden Manipulationen sind eindeutig.

Daher macht es überhaupt keinen Sinn, anzunehmen, das logische Negieren eines Bits, also sein »Umpolen«, könnte zu einem Überlauf führen. Überlauf heißt ja, daß etwas »größer« wird als ein bestimmter Schwellenwert. Den gibt es hier aber nicht. Die Werte, die bei logischen Operationen zum Einsatz kommen, sind Folgen von einzelnen, voneinander unabhängigen Bits. Ein Byte besteht somit aus 8 und ein Wort aus 16 Bits, die nichts miteinander zu tun haben. Sie beeinflussen sich nicht gegenseitig, selbst wenn durch die verfügbaren logischen Operationen bis zu 16 Bits gleichzeitig manipuliert werden können. Das Umpolen von Bit 4 wirkt sich weder auf Bit 3 noch auf Bit 5 aus.

Einen Überlauf kann es somit nicht geben, und das Setzen oder Löschen von Flags, die einen Überlauf anzeigen sollen, ist daher sinnlos. Die Befehlssequenz

```
test      ax,WortVar
jc        @@L1
...
...
test      al,$FF
jo        @@L2
```

wird niemals zu den Labels *L1* und *L2* verzweigen, egal, welche Werte getestet werden.

TIP

Verwenden Sie daher den Vergleich TEST nur in Verbindung mit Befehlen, die das Zero-, Sign- oder Parity-Flag berücksichtigen. Auch das *Sign-Flag* spielt in der Logik eine sehr untergeordnete Rolle: Es ist lediglich eines von 16 (bei Worten) bzw. 8 (bei Bytes) gleichberechtigten Bits. Daß es den Zustand einer prominenten Stelle (Position 15 bzw. 7) widerspiegelt, verdankt es lediglich der Tatsache, daß mit den Registern des Prozessors nicht nur logisch, sondern auch arithmetisch gerechnet werden kann und daß der Prozessor ähnlich wie bei den arithmetischen Operationen nicht vorhersehen kann, ob der Programmierer ein Bitfeld für logische Operationen oder eine Zahl für arithmetische Berechnungen in das Register schreibt.

Letztendlich läuft das alles darauf hinaus, daß man in Verbindung mit TEST nur das Zero-Flag einigermaßen sinnvoll benutzen kann. Es signalisiert, ob nach der verkappten »Und«-Verknüpfung von *Testwert* mit *Prüfmaske*, die TEST ja darstellt, noch irgendein Bit gesetzt ist. Wenn ja, ist *zero* gelöscht, wenn nein, gesetzt. Weitergehende Informationen müssen dann anders beschafft werden.

2.3 Sprungbefehle

Doch was machen wir nun mit dem Ergebnis eines Vergleichs, wie auch immer dieser geartet ist? Wir müssen darauf reagieren. Aber wie?

Was ist das Problem? Wir stellen z.B. mit CMP fest, daß die eine Zahl größer ist als die andere. Offensichtlich haben wir aber tatsächlich vor, in Abhängigkeit davon, ob nun die eine Zahl größer ist oder die andere, unterschiedliche Programmabläufe zu verfolgen. Also läuft praktisch jeder Vergleich auf eine sogenannte Verzweigung im Programm hinaus: Wenn die Bedingung erfüllt ist, so tue dies, ansonsten das. In einem linear angeordneten Speicher kann man nur verzweigen, indem man springt.

In fast allen Fällen folgt auf einen Vergleich zweier Werte ein bedingter Sprung, also ein Sprung an verschiedene Ziele, je nachdem, welches Resultat der Vergleich gebracht hat.

Doch es gibt nicht nur Sprungbefehle, die von Entscheidungen abhängig sind, vielmehr kennt der Prozessor noch zwei weitere Varianten. Wenn wir diese zusammenfassen, lassen sich die Sprungbefehle des Prozessors in folgende Kategorien einteilen:

- Unbedingte Sprungbefehle (JMP)
- Bedingte Sprungbefehle (Jcc)
- Schleifenbefehle (LOOP, LOOPcc)

JMP

Der Sprungbefehl *jmp* zwingt den Prozessor, an eine andere Stelle zu springen – und zwar auf jeden Fall, egal wie die Flags stehen oder was sonst passiert! Aus diesem Grund heißt dieser Befehl auch *unbedingter Sprung*, weil seine Ausführung ohne jede Bedingung erfolgt.

Die Verwendung des Befehls erfolgt, logischerweise, unter Angabe eines Sprungzieles. Dies wird durch

JMP adr

erreicht, wobei als *adr* der Name eines Labels verwendet wird, also z.B.:

```
        JMP   Dahin
        ...
        ...
Dahin:  ...
        ...
```

Prozessorintern wird einfach die Adresse des Zieles in das Instruction-Pointer-Register eingetragen. Ganz so unbeeinflußbar ist dieses Register also offensichtlich doch nicht! Dennoch stimmt es, daß wir nur über den

Umweg von Sprungbefehlen, nie aber durch direkte Manipulationen auf seinen Inhalt Einfluß nehmen können.

HINWEIS

Durch den Befehl JMP können Ziele ganz allgemein angesprungen werden. Es spielt bei der Erstellung des Quelltextes keine Rolle, wie weit vom Ausgangspunkt dieses Ziel entfernt liegt. Dennoch gibt es Unterschiede. So existieren verschiedene Codes für JMP, je nachdem, über welche Distanzen der Sprung erfolgen soll. Die genauen Zusammenhänge sollen uns aber erst im zweiten Teil interessieren. Halten wir bis dahin fest, daß der Befehl JMP vom Assembler unterschiedlich codiert werden kann, aber immer das gleiche tut: Er springt zu einem Zielpunkt.

Bedingte Sprünge

Bedingte Sprünge erfolgen, wie der Name schon sagt, aufgrund einer Bedingung. Doch wer legt die Bedingung fest? Erinnern wir uns an die Vergleichsoperationen, die im letzten Kapitel besprochen wurden. Dort wurden durch die Vergleichsbefehle Flags gesetzt, die über die Situation, die nach dem Vergleich herrscht, Auskunft geben können (z.B. eine Zahl ist größer als die andere, Bit 6 im Prüfwert war gesetzt usw.).

Genau diese Bedingungen, also Flagstellungen, können wir mit den bedingten Sprüngen auswerten. Dazu stehen folgende Sprungbefehle zur Verfügung (im folgenden steht CF für Carry-Flag, ZF für Zero-Flag, SF für Sign-Flag, OF für Overflow-Flag, op1 für Operand #1 und op2 für Operand #2):

- JA: *jump if above*; CF=0 *und* ZF=0; op1 > op2. Wenn der erste Operand größer ist als der zweite, ist das Ergebnis einer Subtraktion größer 0. Daher sind immer das Zero-Flag und das Carry-Flag gelöscht. Ein Test auf CF=0 und ZF=0 bringt also das gewünschte Resultat: Sprung.
- JAE: *jump if above or equal*; CF=0; op1 >= op2. Dies ist ein Spezialfall von JA: Sind die beiden Zahlen gleich groß gewesen, so ist das Ergebnis 0 und das Zero-Flag ist gesetzt. Dieser Test darf daher nur das Carry-Flag berücksichtigen. Damit ist er aber gleichbedeutend mit JNC, wie wir noch sehen werden.
- JB: *jump if below*; CF=1; op1 < op2. Ist dagegen der erste Operand kleiner als der zweite, so ist das Ergebnis einer Subtraktion negativ, das Carry-Flag somit gesetzt. Das Zero-Flag spielt keine Rolle, da es ein »-0« beim 8086 nicht gibt, CF und ZF also nicht gleichzeitig gesetzt sein können. Der Test auf CF=1 ist gleichbedeutend mit JC.
- JBE: *jump if below or equal*; CF=1 *oder* ZF=1; op1 <= op2. Hier kann das Ergebnis entweder negativ sein oder 0. Dann ist ZF=1 und CF=0. Ein Test auf ZF=1 oder CF=1 ist also das Richtige!

- JE: *jump if equal*; ZF=1; op1 = op2. Wenn beide Operanden gleich sind, so ist ihre Differenz 0 und damit das Zero-Flag gesetzt.
- JNE: *jump if not equal*; ZF=0; op1 <> op2. Die Differenz zwischen beiden Operanden ist nicht 0, wenn die Operanden nicht gleich sind. In diesem Fall ist auch das Zero-Flag gelöscht.

Es folgen noch ein paar bedingte Sprünge, die lediglich sprachliche Negationen der eben besprochenen Sprünge und somit programmierbar sind. Allerdings sind es keine echten Befehle, der Assembler »übersetzt« sie in die eben besprochenen.

- JNA: *jump if not above* = JBE: *jump if below or equal*. Wenn etwas nicht größer ist, muß es kleiner oder gleich groß sein.
- JNAE: *jump if not above or equal* = JB: *jump if below*. Kleiner ist etwas immer dann, wenn es nicht größer oder gleich groß ist!
- JNB: *jump if not below* = JAE: *jump if above or equal*. Wenn etwas nicht kleiner ist als etwas anderes, so ist es größer oder gleich groß.
- JNBE: *jump if not below or equal* = JA: *jump if above*. Wenn es nicht kleiner oder gleich groß ist, muß es wohl größer sein!

Die analogen Zusammenhänge bestehen auch bei vorzeichenbehafteten Zahlen des Typs *ShortInt* und *Integer*. Hier sind die Dinge allerdings ein wenig komplizierter, da nun noch die beiden Vorzeichen der Zahlen berücksichtigt werden müssen. Anstelle des Carry-Flag interessieren hier also das Sign- und das Overflow-Flag!

- JG: *jump if greater*; ZF = 0 *und* SF=OF; op1 > op2. Hier wird auch sprachlich der Unterschied zu vorzeichenlosen Zahlen deutlich gemacht: »above« wird durch »greater« ersetzt. Wann nun ist Operand 1 größer als Operand 2? Sehen wir uns dazu die Ergebnisse einer Subtraktion an. Wird von einer positiven Zahl eine kleinere, positive Zahl abgezogen, entsteht eine positive Zahl, ohne daß ein Überlauf entsteht. Auch beim Abziehen einer kleineren negativen Zahl von einer größeren negativen entsteht kein Überlauf! Das Ergebnis bleibt negativ. Zieht man dagegen eine größere positive Zahl von einer kleineren positiven ab, so passiert zweierlei: Es entsteht ein Überlauf, und das Ergebnis ist negativ. Nächster Fall: Zieht man von einer negativen Zahl eine größere positive ab, so erfolgt ebenfalls ein Überlauf; auch hier ist das Ergebnis negativ. Spielt man nun alle Kombinationen durch und vergleicht, was mit OF und SF passiert, so kommt man auf die Bedingung, daß Operand 2 immer dann größer als Operand 1 ist, wenn das Overflow-Flag und das Carry-Flag die gleiche Stellung haben. Da ja auf »größer« getestet wird, muß noch zusätzlich das Zero-Flag 0 sein!

- JGE: *jump if greater or equal*; SF=OF; op1 >= op2. Wie beim Vergleich vorzeichenloser Zahlen unterscheiden sich auch hier JGE und JG nur dadurch, daß im vorliegenden Fall nicht auf ZF geprüft wird.
- JL: *jump if lower*; SF<>OF; op1 < op2. Wenn sich SF und OF voneinander unterscheiden, muß der erste Operand größer sein als der zweite!
- JLE: *jump if lower or equal*; ZF=1 oder SF<>OF. Wenn auch noch das Zero-Flag berücksichtigt wird, können die Zahlen auch noch gleich sein!

Die bedingten Sprünge JE und JNE, die lediglich das Zero-Flag prüfen, sind natürlich auch bei vorzeichenbehafteten Zahlen gültig und definiert.

Auch hier, wie im obigen Fall, gibt es die sprachlichen Negationen:

- JNG; *jump if not greater*, identisch mit JLE
- JNGE; *jump if not greater or equal* = JL
- JNL; *jump if not lower* oder auch JGE
- JNLE; *jump if not greater or equal*, was gleichbedeutend mit JG ist

Wenn wir schon bei den Flags sind: Natürlich können Flags auch einzeln abgeprüft werden:

- JC; *jump if carry set*. Wie wir wissen, zeigt ein gesetztes CF einen Überlauf vorzeichenloser Zahlen an. Somit ist JC identisch mit JB. Allerdings kann das Carry-Flag auch von Operationen gesetzt werden, die nichts mit Zahlen zu tun haben. Das Carry-Flag ist ein ziemlich intensiv genutztes Flag. Daher diese »eigene« Behandlung, auch wenn rein physikalisch natürlich das gleiche passiert wie bei JB.
- JZ; *jump if zero flag set*. Nicht nur Vergleiche können ergeben, daß das Resultat 0 ist, auch andere Operationen, die wir noch kennenlernen werden.
- JS; *jump if sign flag set*. Dieser Sprung ist verlockend, stellt er doch eine bequeme Reaktion auf das Vorliegen einer negativen Zahl dar! *Aber bitte denken Sie daran: Nicht bei jeder Subtraktion einer größeren von einer kleineren Zahl ist das Ergebnis negativ! Verwechseln Sie also niemals JS mit JL!*
- JO; *jump if overflow flag set*. Wer das Carry-Flag berücksichtigt, muß auch das Overflow-Flag betrachten.
- JP; *jump if paritiy flag set*. Das Parity-Flag ist gesetzt, wenn eine ungerade Anzahl von Bits nach dem Vergleich gesetzt ist.

Was hier fehlt, ist das Auxiliary-Flag. Aber wie oben schon gesagt, ist es wirklich so bedeutungslos, daß noch nicht einmal Sprungbefehle existieren, die es berücksichtigen. Es findet tatsächlich nur intern bei Korrekturbefehlen Verwendung!

Hier die korrespondierenden sprachlichen Negationen der Befehle:

- JNC; jump if no *carry* set.
- JNZ; jump if no *zero flag* set.
- JNS; jump if no *sign flag* set.
- JNO; jump if no *overflow flag* set.
- JNP; jump if paritiy *flag* set.

Es fehlen noch ein paar Exoten. Bei der Besprechung des Parity-Flags verwiesen wir auf die Datenkommunikation und ein Prüfbit, welches mit den Bytes gesendet wird. Man kann nun den dafür zuständigen Baustein dazu veranlassen, immer dann ein gesetztes Bit zu schicken, wenn die Parität des Bytes gerade ist, also eine gerade Anzahl von Bits im Byte vorliegen. Man kann ihm aber auch das genaue Gegenteil befehlen: ein gesetztes Prüfbit zu schicken, wenn die Parität ungerade ist. Um bei der Interpretation die Arbeit etwas zu erleichtern, existieren daher für das Parity-Flag noch ein paar Redundanzen, diesmal sogar ganz echte:

- JPE; *jump if parity even*. Wenn die Parität gerade ist, so ist das Parity-Flag gesetzt. Daher entspricht dieser bedingte Sprung JP.
- JPO; *jump if parity odd*. Dies ist immer dann der Fall, wenn das Parity-Flag nicht gesetzt ist, entspricht also JNP.

So weit, dies auch noch sprachlich zu negieren, etwa bei »JPNE – jump if *parity* not even«, ging man dann doch nicht! Wenn eine Parität nicht gerade ist, so muß sie wohl ungerade sein.

Es folgt ein letzter Sprungbefehl, aber ein äußerst wichtiger:

- JCXZ; *jump if cx zero*. Dies ist ein bedingter Sprung, der immer dann ausgeführt wird, wenn der Inhalt des Registers CX, und nur dieses Registers, 0 ist.

Warum dieser Befehl so nützlich ist, werde ich Ihnen zeigen, wenn wir über Schleifen sprechen. Soviel an dieser Stelle: Wie bei der Besprechung des Registersatzes schon angedeutet wurde, hat jedes der Rechenregister spezielle Aufgaben. So heißt CX auch Count-Register und ist für das Zählen in Schleifen verantwortlich. Ein kleiner Tip als Vorgriff auf die Erklärung: Schleife – Zähler – »springe, wenn Zähler = 0«.

TIP

Scheuen Sie sich nicht, tatsächlich die Befehle zu benutzen, die in Ihren Augen die Bedingung richtig beschreiben, da der Assembler ggf. die Redundanzen auflöst. Falls Sie glauben, einen Sprung zu veranlassen, wenn der Operand 1 nicht kleiner oder gleich dem Operanden 2 ist, dann ver-

wenden Sie JNLE. Wenn Sie an anderer Stelle einen Sprung durchführen müssen, wenn Operand 1 größer als Operand 2 ist, dann benutzen Sie JG! Kümmern Sie sich einfach gar nicht darum, ob diese Befehle vielleicht identisch sind oder nicht, der Assembler weiß das schon! Durch falsch verstandene »Optimierung« werden in der Regel nur mehr Fehler erzeugt, als Nutzen gezogen werden kann. Denn diese beiden Befehle sind bis in die letzte Aktion absolut identisch, Sie gewinnen durch ihren Austausch nichts!

ACHTUNG

Worauf Sie jedoch achten müssen, ist, ob Sie vorzeichenlose oder vorzeichenbehaftete Zahlen manipuliert haben und nun darauf reagieren wollen! JA ist eben nicht das gleiche wie JG, auch wenn die Befehle in sehr vielen Fällen (nämlich bei positiven Zahlen mit einem Wert, der dem der größten im Register darstellbaren vorzeichenbehafteten Zahl entspricht) identisch reagieren. Aber es könnte ja einmal das oberste Bit betroffen sein, und dann geht's los!

Daher nochmals die Warnung! Solche Fehler sind sehr schwer zu entdecken und zu korrigieren! Sie müssen bei der Programmierung in Assembler zu jedem Zeitpunkt wissen, mit welchen Daten Sie arbeiten. Hilfen wie z. B. die Typüberprüfung von Pascal gibt es im Assembler nur sehr, sehr wenige!

HINWEIS

Bedingte Sprünge sind immer sogenannte *Short Distance Jumps* oder *Short Jumps*, also Sprünge, die eine maximale Sprunglänge nicht überschreiten können. Dies klingt zunächst wie eine Einschränkung, ist es aber in Wirklichkeit nicht. Wie man das Problem ganz einfach in den Griff bekommen kann, werden wir im zweiten Teil das Buches sehen.

Schleifenbefehle

Die Schleifenbefehle des 8086 sind eigentlich nur eine Vereinfachung für den Programmierer. Sie sind nämlich mit anderen Befehlen nachbildbar! Der Hintergrund ist dabei sehr einfach: Bei der Abarbeitung einer Schleife nimmt man ja an, daß der Programmteil innerhalb der Schleife so lange ausgeführt werden soll, bis eine Bedingung erfüllt ist.

In Hochsprachen wie Pascal, C oder Basic gibt es dafür solche Schleifen wie »Repeat ... Until I = 5«, wo ein Programmteil so lange ausgeführt wird, bis die Bedingung I = 5 erfüllt ist. Aber auch das Konstrukt »For I = 1 to 5 Do ...« bildet eine Schleife, die fünfmal durchlaufen wird. Eine weitere Schleifenkonstruktion ist »While I <> 5 Do ...«. Allen diesen Schleifen ist gemeinsam, daß die Ausführung von Programmteilen an eine Bedingung geknüpft ist. Nach dem bisher Bekannten fällt es uns nicht schwer, beispielsweise die »Repeat-Schleife« von eben nachzubilden:

```
...
...
cmp       I,5
jne       Schleifenanfang
```

Der Teil vor dem CMP-Befehl wird zunächst erledigt. Dann wird die Zählvariable I mit der Konstanten 5 verglichen. Solange I nicht 5 ist, verzweigt der bedingte Sprung an eine Stelle, die hier »Schleifenanfang« genannt wurde und irgendwo »oberhalb« des Codeausschnitts steht. Die Schleife funktioniert. Voraussetzung ist natürlich, daß irgend jemand den Schleifenzähler I verändert, ansonsten haben wir eine sogenannte Endlosschleife, aus der es kein Zurück gibt.

Ähnlich sind alle anderen Schleifen realisierbar: Ausführung des Codes, Prüfung, ob die Bedingung erfüllt ist, und Sprung an den Schleifenanfang, wenn dies nicht der Fall ist. Dieses Vorgehen ist aber so elementar für die Programmierung von Schleifen, daß den Entwicklern des 8086 hierfür die Reservierung eines eigenen Befehls sinnvoll erschien: LOOP.

LOOP macht nichts anderes als die beiden Befehle oben: Vergleich einer Zählvariablen mit einem Wert und Sprung an den Schleifenbeginn, wenn der Vergleich dazu zwingt. Auch die Aktualisierung der Schleifenvariablen erledigt LOOP gleich mit: Vor dem eigentlichen Vergleich wird die Zählvariable dekrementiert, also um 1 verringert. Lediglich die Zählvariable kann bei LOOP nicht vorgegeben werden, und es finden auch nur einige Bedingungen Berücksichtigung.

Die Zählvariable bei LOOP ist immer das CX-Register. Andere Register sind nicht verwendbar und auch keine Speicherstellen im RAM. Hier haben wir also ein weiteres Beispiel für die Spezialisierung der vier Rechenregister des Prozessors! CX ist das Count-Register und enthält definitionsgemäß den Schleifenzähler.

Alle Schleifenbefehle haben die Form:

LOOP adr

Die Bedingungen, unter denen LOOP verzweigen kann, sind folgende:

- LOOP adr; *loop until cx=0*; dies ist der »einfache« Schleifenbefehl, der so lange an das Label *adr* verzweigt, bis CX 0 ist. Danach wird das Programm mit dem nächsten auf LOOP folgenden Befehl fortgesetzt.
- LOOPZ adr; *loop until zero or cx=0*; dieser Befehl prüft noch zusätzlich, ob das Zero-Flag gesetzt wurde. Ist dies der Fall oder der Inhalt von CX gleich 0, so wird die Schleife verlassen, also nicht an *adr* zurückgesprungen. Sinnvoll ist dieser Befehl dann, wenn innerhalb der Schleife ein weiterer Befehl Einfluß auf das Zero-Flag nimmt. So könnte z.B. in der Schleife geprüft werden, ob zwei Werte gleich sind. Die Schleife wird nun so lange wiederholt, bis diese Werte gleich sind (Zero-Flag ist dann gesetzt), maximal aber, bis CX 0 ist.

- LOOPNZ adr; *loop until not zero or cx=0*; dieser Schleifenbefehl ist der Gegenspieler zu LOOPZ. Die Schleife wird abgebrochen, wenn CX gleich 0 oder wenn das *Zero-Flag* gelöscht ist.

Auch hier gibt es Redundanzen wie bei den Sprungbefehlen: LOOPE (*loop until equal or cx=0*) ist identisch mit LOOPZ und LOOPNE (*loop until not equal or cx=0*) mit LOOPNZ.

TIP Stellen Sie sich die LOOP-Befehle als Codefolge vor! So ist z.B. LOOP identisch mit folgender Sequenz (DEC bekommen wir gleich; es verringert den Inhalt des Operanden um 1):

```
...
...
dec       cx
jnz       Schleifenanfang
```

LOOPZ und LOOPE dagegen können interpretiert werden als

```
          ...
          ...
          jz    Weiter
          dec   cx
          jnz   Schleifenanfang
Weiter: ...
```

und LOOPNZ bzw. LOOPNE als

```
          ...
          ...
          jnz   Weiter
          dec   cx
          jnz   Schleifenanfang
Weiter: ...
```

ACHTUNG Die genannten Codesequenzen und die Befehle agieren nicht identisch! Während LOOP, LOOPZ/LOOPE und LOOPNZ/LOOPNE die Flags nicht verändern, tut DEC dies sehr wohl.

ACHTUNG Beachten Sie bitte, daß unter Assembler grundsätzlich alle Zähler vorzeichenlose Zahlen sind. CX kann daher keine negativen Werte beinhalten, es können nur Zählwerte zwischen 0 und 65.535 eingetragen werden. Außerdem ist bei LOOP-Befehlen grundsätzlich das 16-Bit-Register CX involviert! Beim Beladen des CX-Registers ist daher darauf zu achten, daß die richtigen Ein-/Ausgabebefehle verwendet werden. Das Beladen des CL-Registers mit dem Wert 50 resultiert nur dann in einer Zählvariablen von 50, wenn sicher ist, daß CH 0 enthält. Ist dort (noch) ein anderer Wert vorhanden, wird die Schleife entsprechend häufiger durchlaufen. Dies ist ein häufig gemachter Fehler, der zu schwer auffindbaren Programmierfehlern führen kann.

2.4 Flags

Es gibt auch ein paar Befehle, die die Flags direkt betreffen. Bisher wurden Flags nur durch die Operationen verändert und ihr Zustand durch Operationen geprüft. Nun wird es Zeit, selbst Hand anlegen zu können. Die hierfür vorgesehenen Befehle sind:

- Befehle, die das gesamte Flagregister betreffen (LAHF, SAHF, PUSHF und POPF) und
- Befehle zum gezielten Verändern von einzelnen Flags (CLC, CMC, STC, CLD, STD, CLI und STI)

LAHF, SAHF

Mit LAHF (*Load AH with Flags*) werden die Bits 7 bis 0 in das AH-Register kopiert. Es sind nur die niederwertigen 8 Bits des Flagregisters, weil die oberen ja von untergeordnetem Interesse sind: Mit ihnen lassen sich wenig Operationen durchführen und steuern. Sie dienen eher zum Einstellen eines bestimmten Zustands.

SAHF macht das Ganze umgekehrt: Es kopiert das Byte in AH in die niederwertigen 8 Bits des Flagregisters. Wozu das alles? Zugegeben, der praktische Nutzen von LAHF ist recht klein. LAHF macht aus den einzelnen Bits des Flagregisters einen Zahlenwert, der auch nur die Flagpositionen repräsentiert. Sicher, man kann mit dieser Zahl Berechnungen durchführen – aber wozu? Ich gebe zu, auch nicht so recht hinter den Nutzen von LAHF gekommen zu sein. Mit SAHF ist das etwas anderes! Dieser Befehl spielt, wie wir im nächsten Kapitel noch sehen werden, in Verbindung mit dem Coprozessor eine große Rolle.

PUSHF, POPF

PUSHF und POPF kopieren den Inhalt des Flagregisters auf den oder vom Stack. Der Stack ist so etwas wie ein lokaler Speicherbereich für den Prozessor, in dem er temporär Daten ablegen kann. Wir kommen auf den Stack im zweiten Teil des Buches zurück. Merken wir uns nur, daß mit diesen beiden Befehlen der Inhalt des Flagregisters gespeichert und wieder geholt werden kann.

CLC, CMC, STC

Diese drei Befehle sprechen das Carry-Flag direkt an. Mit STC (*Set Carry-Flag*) kann das Flag explizit gesetzt und mit CLC (*Clear Carry-Flag*) gelöscht werden. CMC (*Complement Carry-Flag*) schaltet das Flag um: ist es gesetzt, so wird es gelöscht, ist es gelöscht, so wird es gesetzt.

Auf den ersten Blick stellt sich die Frage: Wozu? Aber wir haben ja schon gesehen, daß das Carry-Flag eine bedeutende Rolle spielt und häufig verwendet wird. Mit STC und CLC können wir also bestimmte Anfangsbedingungen vorgeben, auf die dann später reagiert werden kann. CMC wird selten verwendet – aber wenn, dann mit drastischen Auswirkungen. Auch in diesem Punkt verweise ich auf den zweiten Teil des Buches: Dort werden wir genügend Beispiele kennenlernen.

Auch das Direction-Flag hat eine wesentliche Bedeutung: Es steuert die Richtung, in der die Stringoperationen erfolgen sollen. Was es damit auf sich hat, sehen wir ein paar Abschnitte weiter unten. **CLD, STD**

Wie beim Carry-Flag auch kann mit STD (*Set Direction-Flag*) das Flag explizit gesetzt und mit CLD (*clear Direction-Flag*) gelöscht werden. CMD, also das Umschalten, macht beim Direction-Flag keinen Sinn und ist daher auch nicht im Befehlssatz des 8086 enthalten.

Ein letztes, sehr wichtiges Flag ist das *Interrupt-Enable-Flag*. Es steuert, ob ein Interrupt ausgelöst werden darf oder nicht. **CLI, STI**

Interrupts sind Ereignisse, die mitten in der Abarbeitung eines Programms auftreten können, ohne mit dem eigentlichen Programm etwas zu tun haben zu müssen. So muß z.B. die Systemzeit, die der Rechner ja hat, aktualisiert werden. Falls eine Taste gedrückt wird, muß das registriert und darauf reagiert werden. Es gibt noch viele andere Gründe, warum ein derzeit laufendes Programm unterbrochen werden kann.

Nun kann es aber vorkommen, daß ein Programm an einer bestimmten Stelle auf keinen Fall unterbrochen werden darf. Dann muß die Interrupt-Auslösung verboten werden, was man durch Löschen des Interrupt-Enable-Flags erreicht. Im Programm wird an der entsprechenden Stelle also CLI (*Clear Interrupt-Enable-Flag*) verwendet.

Ist die kritische Situation dann vorbei, müssen Interrupts wieder zugelassen werden, damit der Rechner nicht vollständig durcheinanderkommt. Man erreicht dies durch Setzen des Interrupt-Enable-Flags mit STI (*Set Interrupt-Enable-Flag*).

2.5 Bitschiebeoperationen

Zahlen sind, vom Computer aus betrachtet, nichts anderes als eine Folge von Bits, also Zuständen von 0 und 1 (siehe Anhang A). Im Prinzip lassen sich alle Manipulationen an Zahlen auf Manipulationen mit Bits zurückführen. So auch die arithmetischen Operationen.

Dennoch kann man auch bei Zahlen manchmal sinnvollerweise die Bits manipulieren, die sie codieren. Hierzu gibt es dann spezialisierte Befehle des 8086. Die einzigen Bitmanipulationen, zu denen der 8086 fähig ist, beschränken sich auf das Verschieben der Bits eines Bytes bzw. Worts in den betreffenden Registern oder Speicherstellen. Es besteht hierbei grundsätzlich die Möglichkeit,

- die Bits zyklisch zu vertauschen, sie also zu »rotieren« (RCL, ROL, RCR, ROR) oder

- sie lediglich in einer Richtung zu verschieben, zu *shift*en (SAL, SHL, SAR, SHR).

Für alle diese Befehle besteht die Möglichkeit, die Aktion

- »nach links« (RCL, ROL, SAL, SHL) oder
- »nach rechts« (RCR, ROR, SAR, SHR)

vorzunehmen. Ferner gibt es die Möglichkeit, bei den zyklischen Rotationen

- das Carry-Flag nicht einzubeziehen (ROL, ROR) bzw.
- das Carry-Flag einzubeziehen (RCL, RCR).

Alle Befehle haben als Parameter zunächst das Register bzw. die Speicherstelle, in dem bzw. der die Bitmanipulation erfolgen soll und anschließend eine Angabe, um wie viele Positionen verschoben/rotiert werden soll. Hierbei gibt es nur die Auswahl zwischen

- einer Position, also z.B.
```
RCL   AX, 1
ROR   WortVar,1
SAR   ByteVar,1
SHL   CL,1
```
- einer Anzahl Positionen, die in CL verzeichnet ist, also z.B.
```
RCR   BX, CL
ROL   ByteVar, CL
SAL   CL, CL
SHR   WortVar, CL
```

ACHTUNG Beim 8086 ist die Angabe einer Konstante > 1 in den Bitbefehlen nicht möglich. Anweisungen wie *RCL AX, 5* oder *SAL WortVar, 7* sind erst ab dem 80186 implementiert und sollten daher nur dann benutzt werden, wenn die Programme oder Programmteile, die in Assembler erstellt wurden, ausdrücklich nicht auf Computern mit 8086/8088-Prozessoren laufen sollen.

Was aber passiert nun tatsächlich in den Registern? Schauen wir uns dazu ein beliebiges Register an – es könnte auch eine Speicherstelle sein. Wir »vergrößern« es so weit, daß wir die einzelnen Bits, aus denen die Zahl besteht, sehen können. Nehmen wir die Zahl 4711, die binär so aussieht:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1

Bitte beachten Sie, daß die Zahlen über dem Kasten nur Hilfen sind; sie geben die Bitnummer an.

Wir haben eben gesehen, daß Bitverschiebungen nach links und rechts möglich sind, hier zunächst um eine Postion nach links:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1		

Aber es geht auch nach rechts:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
		0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1

Was uns in beiden Fällen auffallen sollte, ist folgendes: Am einen Ende verläßt ein Bit das Register, am anderen Ende fehlt nach der Verschiebung ein Bit. Genau das ist der Punkt, in dem sich die verschiedenen Schiebebefehle unterscheiden: in der Art, wie die Bits, die das Register »verlassen«, sowie die, die »nachgeschoben« werden müssen, behandelt werden!

Einfachster Fall: Das Bit, das das Register verläßt, wird auf der anderen Seite wieder hineingeschoben: **ROL**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1	
0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1		
	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1		**0**
	0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	**0**	

Diesen Vorgang nennt man »zyklisches« Rotieren, da die Bits wie auf einer Kreisbahn bewegt werden. Im Beispiel erfolgte das nach links. Der Befehl, der dies bewerkstelligt, heißt ROL, *Rotate Left*.

Anders herum: **ROR**

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**	
		0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**
1		0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	
	1	0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	

Der passende Befehl heißt ROR, *Rotate Right*. Natürlich geht das auch mit mehreren Positionen gleichzeitig. Allerdings muß, wie schon gesagt, die Anzahl dann in CL stehen.

SHL Die nächste Möglichkeit ist, das Bit, das das Register verläßt, einfach zu vergessen. Wir dürfen es jedoch nicht wirklich vergessen, vielleicht brauchen wir die Information noch! Schieben wir es daher in das Carry-Flag! Doch was schieben wir dann an die frei werdende Position? Antwort: einfach eine 0!

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
?	**0**	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1	
0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1		0
0	0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	0	

Das Fragezeichen soll hierbei entweder für eine 0 oder für eine 1 stehen, je nachdem, welchen Wert das Carry-Flag gerade hatte.

SHR Hier wird der Inhalt nicht rotiert, sondern nur verschoben. Daher heißt der Befehl auch *Shift Left* oder SHL. Auch hier gibt es ein Pendant nach rechts:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	C
	0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**	**?**
0		0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**
	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**

Bitte machen Sie sich den Unterschied zwischen ROL und SHL sowie zwischen ROR und SHR noch einmal klar! Bei den *Rotate*-Befehlen werden die Bits innerhalb des Registers lediglich an neue Positionen im Kreis verschoben, bei den *Shift*-Befehlen werden die Registerinhalte verändert! Das Carry-Flag hält bei den SHFx-Befehlen noch das »verlorengegangene« Bit. Übrigens: Bei mehreren gleichzeitigen *Shifts* über den Inhalt von CL enthält *Carry* immer das letzte Bit, das das Register verließ!

So weit, so gut. Als wir die Vergleichsbefehle kennengelernt haben, haben wir erfahren, daß es vorzeichenlose und -behaftete Zahlen gibt. Bei letzteren ist das »oberste« Bit, hier also Bit 15, das Vorzeichenbit. Beim Verschieben nach links macht das kein Problem: Bit 15 enthält nach dem Schieben zwar Bit 14, was nichts mit dem Vorzeichen zu

tun hat, aber *carry* enthält ja noch die Information, ob die Zahl negativ war oder nicht. Dennoch ist Vorsichtgeboten: Das gilt nur bis zum nächsten Befehl, der das Carry-Flag verändert.

SAR

Schlimmer ist da ein Verschieben nach rechts! Denn hier wird das Vorzeichenbit in Bit Nummer 14 kopiert und Bit 15 mit 0 überschrieben. Die Information, ob die Zahl negativ war, müßte durch eine Analyse von Bit 14 mühselig eruiert werden, wenn es nicht den Befehl SAR gäbe!

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	C
1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**	**?**
←	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	**1**

SAR kopiert einfach das nach Bit 14 verschobene Vorzeichenbit nach Position 15. Somit bleibt eine negative Zahl nach SAR weiterhin negativ, eine positive bleibt positiv. Aus diesem Grunde heißt SAR auch *Shift Arithmetically Right*.

SAL

Ein *Shift Arithmetically Left* gibt es nicht wirklich! Zwar kennt der Assembler den Befehl SAL, jedoch erzeugt dieser das gleiche Bild wie SHL.

RCL

RCL arbeitet absolut identisch zu ROL – mit einer Ausnahme: das Carry-Flag wird mit einbezogen. Das Bit, das links das Register verläßt, wandert ins Carry-Flag, und die Information von dort wird rechts in das Register hineingeschoben (vgl. Abbildung auf der folgenden Seite).

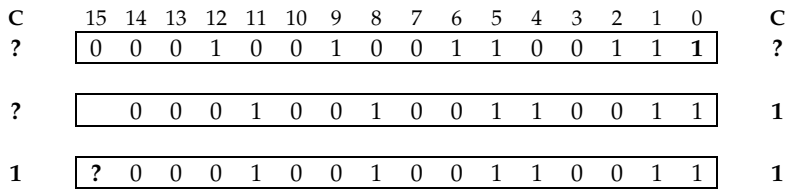
Bitte lassen Sie sich nicht durch die Darstellung des Carry-Flags links und rechts vom Register beeinflussen: Es gibt natürlich nur ein Carry-Flag!

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	C
?	**0**	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1	**?**
0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	1		**?**
0	0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	**?**	**0**

HINWEIS

Hier nun können Sie schon einen Sinn für die Befehle CLC und STC entdecken: Da der Inhalt des Carry-Flags »in das Register hineingeschoben wird«, sollte man vorher bestimmen können, ob da eine 0 oder eine 1 geschoben wird! Steuern können Sie dies mit STC (=1) und CLC (=0).

RCR Das fehlende Gegenstück zu RCL ist RCR. Hier wird das Bit, das das Register rechts verläßt, in das Carry-Flag geschoben. Dessen ursprünglicher Inhalt wandert links in das Register hinein, um die leere Position zu füllen.



Nun kennen Sie die verschiedenen Bitschiebebefehle. Auch wenn es im Moment noch nicht den Anschein haben sollte, diese Befehle sind sehr wichtige Hilfsmittel. Ein kleiner Hinweis: Da die Zahlen immer als Potenz von 2, also binär, verwaltet werden, sind Multiplikationen/Divisionen mit Potenzen von 2 durch diese Schiebebefehle sehr schnell und einfach realisierbar! Wir werden im zweiten Teil des Buches sehr häufig von dieser Möglichkeit Gebrauch machen.

Übrigens: Außer dem Carry-Flag wird bei diesen Befehlen kein Flag verändert – und, wie wir gesehen haben, auch das Carry-Flag nicht bei allen!

2.6 Logische Operationen

Auch wenn Sie es nicht so richtig glauben wollen: Der Prozessor kann auch logisch sein! Besser gesagt: logisch rechnen – zumindest, was die Mathematik darunter versteht. Als logische Operationen kennt der Prozessor

- die UND-Verknüpfung zweier Werte (AND)
- die inklusive ODER-Verknüpfung (OR)
- die exklusive ODER-Verknüpfung (XOR)
- die logische Verneinung oder Negation (NOT)

Bei den logischen Operationen erfolgt die Manipulation der Daten bitweise, also Bit für Bit unabhängig voneinander. Die Inhalte der Register sind bei logischen Operationen als Felder von 8 oder 16 einzelnen, voneinander unabhängigen Bits zu interpretieren. Steht in AL z.B. der Wert $17, so heißt das, daß die Bits 4, 2, 1 und 0 gesetzt sind. Veränderungen des Bits 3 z.B. haben keinen Einfluß auf den Zustand der anderen Bits.

Weil in den Hochsprachen logische Operationen lediglich in Form von Manipulationen *Boolescher Variablen* und *Konstanten* Verwendung finden und der Hintergrund dabei nicht so offensichtlich wird, sollte hier vielleicht ein wenig mehr über logische Operationen gesagt werden. Sie sind wie folgt definiert:

- Nach einer UND-Verknüpfung (AND) ist ein Bit dann und nur dann gesetzt, wenn die beiden zu verknüpfenden Bits vor der Operation auch gesetzt waren. Andernfalls resultiert ein gelöschtes Bit.
- Das Ergebnis einer (»inklusiven«) ODER-Verknüpfung (OR) ist immer dann ein gesetztes Bit, wenn mindestens eines der zu verknüpfenden Bits gesetzt war. Nur wenn beide »Ausgangsbits« gelöscht waren, ist auch das Ergebnis ein gelöschtes Bit.
- Eine »exklusive« ODER-Verknüpfung (XOR) arbeitet genau so wie eine »normale« ODER-Verknüpfung, nur mit dem Unterschied, daß das Ergebnis ein gelöschtes Bit ist, wenn beide Bits ursprünglich entweder gesetzt oder gelöscht waren. Nach einer XOR-Operation ist ein Bit also nur dann gesetzt, wenn entweder das eine oder das andere zu verknüpfende Bit gesetzt waren, niemals aber, wenn beide den gleichen Ausgangszustand hatten.

Zusammenfassend kann also folgendes Schema aufgestellt werden:

Bits	**AND**	
$_1 \backslash ^2$	0	1
0	0	0
1	0	1

Bits	**OR**	
$_1 \backslash ^2$	0	1
0	0	1
1	1	1

Bits	**XOR**	
$_1 \backslash ^2$	0	1
0	0	1
1	1	0

NOT löscht ein gesetztes Bit, wenn es vorher gesetzt war und umgekehrt. AND, OR und XOR verknüpfen zwei Bits miteinander, während NOT als Parameter nur den Zustand des Bits verändert, also keine Bits verknüpft.

ACHTUNG

Beachten Sie bitte, daß eine AND-Verknüpfung mit nachfolgendem Umpolen durch NOT nicht das gleiche ist wie eine OR- oder XOR-Verknüpfung! Dieser Fehler wird vor allem von Anfängern häufig gemacht, aber auch »alten Hasen« unterläuft er manchmal. Falls im Schema oben die einzelnen Felder der AND-Verknüpfung logisch negiert werden, ergibt sich:

Bits	**AND NOT**	
$_1 \backslash ^2$	0	1
0	1	1
1	1	0

Bits	**OR**	
$_1 \backslash ^2$	0	1
0	0	1
1	1	1

Bits	**XOR**	
$_1 \backslash ^2$	0	1
0	0	1
1	1	0

Wie Sie sehen, unterscheiden sich die Bitmuster deutlich. Probieren Sie das einmal für verschiedene Kombinationen der logischen Operationen aus.

Nach dem oben Gesagten sollte es nicht schwer sein herauszufinden, welches Bitmuster sich in AH z.B. wiederfindet, wenn man das Bitfeld 10100101 in AH mit dem Bitfeld 01010101 in BL AND-, OR- und XOR-verknüpft oder einfach mit NOT logisch negiert. Sollte sich dann bei Ihnen nicht 00000101, 11110101, 11110000 und 01011010 ergeben, dann haben Sie eventuell die Bits nicht unabhängig voneinander betrachtet:

AND		**OR**		**XOR**		**NOT**	
AH	10100101	AH	10100101	AH	10100101	AH	10100101
BL	01010101	BL	01010101	BL	01010101		
AH	00000101	AH	11110101	AH	11110000	AH	01011010

Bitte denken Sie an diese Arbeitsweise, wenn Sie den Inhalt der Register ab jetzt nicht mehr bitweise, sondern registerweise interpretieren! Wenn Sie die Bitfelder im obigen Beispiel als Zahl darstellen, liefert 165 AND 85 als Ergebnis 5, 165 OR 85 ist 245, 165 XOR 85 resultiert in 240, und NOT(165) ist 80. Nur zur Übung das Ganze auch hexadezimal: $A5 AND $55 = $05, $A5 OR $55 = $F5, $A5 XOR $55 = $F0 und Not($A5) = $5A. Wie Sie sehen, spiegelt die hexadezimale Darstellung den tatsächlichen Sachverhalt besser wider als die dezimale.

Worauf kann man die logischen Operationen nun anwenden? Es ist ganz einfach: wie die Vergleichsbefehle auch auf jedes der vier Rechenregister, die *Indexregister* DI und SI und jede Byte- oder Word-Speicherstelle. Wenn XXX für AND, OR oder XOR steht, so sind folgende Möglichkeiten gegeben:

- XXX reg, reg z.B. AND AX, BX
OR CL, DH
- XXX reg, mem z.B. XOR SI, WordVar
AND DH, ByteVar
- XXX mem, reg z.B. AND WordVar, DX
XOR ByteVar, AL
- XXX reg, const z.B. OR CX, $4711
XOR BH, $FF
- XXX mem, const z.B. AND WordVar, $0815
OR ByteVar, $AB

Wichtig ist bei diesen Operationen, daß der zuerst genannte Operand gleichzeitig auch Speicherplatz für das Ergebnis ist. Der Befehl *AND WordVar, DX* führt also eine AND-Verknüpfung zwischen dem Register DX und dem Inhalt der Variablen *WordVar* durch und schreibt das Ergebnis in *WordVar*, so daß deren ursprünglicher Inhalt verloren ist. *AND DX, WordVar* führt zwar vom Ergebnis der logischen Verknüpfung her zum gleichen Resultat, da alle logischen Operationen kommutativ sind, überschreibt aber das *DX-Register* mit dem Ergebnis. Daher sind die beiden Befehle vom logischen Ergebnis her gleich, vom Ort aus betrachtet, an dem das Resultat abgelegt wird, aber nicht.

Bei der logischen Verneinung ist nur ein Operand möglich, der auch gleichzeitig das Ziel der Operation ist. Konstanten können also logischerweise logisch nicht negiert werden:

- NOT reg z.B. NOT AX
NOT DL
- NOT mem z.B. NOT WordVar
NOT ByteVar

Wie bei sehr vielen Befehlen werden auch bei den logischen Operationen mit Ausnahme von NOT Flags verändert, und zwar ganz analog zu TEST:

- Zero-Flag wenn nach der Verknüpfung alle Bits 0 sind
- Parity-Flag wenn das Ergebnis eine gerade Anzahl von Bits aufweist
- Sign-Flag, wenn Bit 15 nach dem Verknüpfen von Wortregistern bzw. Bit 7 bei Byteregistern gesetzt ist
- Carry-Flag und Overflow-Flag werden definitiv und immer gelöscht

Alle anderen Flags bleiben unverändert. NOT dagegen läßt alle Flags unverändert!

TIP

Neben dem eigentlichen Verknüpfen der Bits mittels logischer Operationen werden also auch Flags gesetzt. Zwei der logischen Operatoren lassen sich daher recht einfach und effektiv zweckentfremden. Es handelt sich um die Operationen OR und XOR. Wer verbietet uns nämlich, ein Register mit sich selbst zu verknüpfen? Niemand! Also sind z.B. folgende Konstrukte möglich:

```
...
or      ax,ax
xor     bh,bh
...
```

Im ersten Fall wird der Inhalt von AX mit sich selbst ODER-verknüpft. Ergebnis: Der Inhalt wird nicht verändert. Das wäre nicht weiter erwähnenswert, wenn nicht in Abhängigkeit von den nach der Verknüpfung gesetzten Bits die Flags gesetzt würden. Das werden sie aber, so daß nach diesem Befehl anhand des Zero-Flags z.B. geprüft werden kann, ob der Inhalt von AX 0 ist. Das Sign-Flag zeigt an, ob in AX eine negative Zahl steht, wenn der Registerinhalt vorzeichenbehaftet interpretiert werden muß. Auf diese Weise läßt sich recht einfach eine Verzweigung des Programms erreichen:

```
         :
         or    bl,bl
         jz    IsZero
         js    IsLower
         :     ; hier geht's weiter, wenn die Zahl positiv
IsZero:        ; hier geht's weiter, wenn die Zahl 0 ist
         :     ; und so weiter ...
IsLower: ; und hier, wenn sie negativ ist!
```

Zwar ist eine solche Verzweigung auch mit den herkömmlichen Vergleichen möglich, z.B. durch:

```
         cmp   bx,0
         jz    IsZero
         jl    IsLower
         :
IsZero:  :
IsLower: :
```

Jedoch braucht der CMP-Befehl mehr Speicherplatz und dauert auch länger in der Ausführung, da hier unter Verwendung einer Konstanten arithmetisch gearbeitet wird.

Der zweite Befehl macht etwas ganz anderes: Durch die XOR-Verknüpfung des Registerinhalts mit sich selbst werden alle Bits gelöscht. Also ist danach grundsätzlich das Zero-Flag gesetzt und das Sign-Flag gelöscht. Was also soll dies? Ganz einfach: Die Flags interessieren uns hier gar nicht! *XOR CL,CL* macht eigentlich nichts anderes als *MOV CL, 0*, schreibt also eine 0 in das Register. Aber auch hier gilt: Mit XOR geht es viel schneller, und Sie sparen Speicherplatz für den Code!

2.7 Arithmetische Operationen

Natürlich sollte man von einem Prozessor auch erwarten können, daß er rechnen kann. Das kann der 8086 selbstverständlich auch, wenn auch nur in einem sehr beschränkten Ausmaß. So kennt er als Werte, mit denen er arithmetische Operationen durchführen kann, nur sogenannte Integer-

zahlen, also Zahlen, die keinen Nachkommateil besitzen. Die bei der Manipulation solcher Zahlen möglichen Befehle sind:

- Additionsbefehle (ADC, ADD)
- Subtraktionen (SBB, SUB)
- Multiplikationen (MUL, IMUL)
- Divisionen (DIV, IDIV)
- Inkrementationen und Dekrementationen (INC, DEC)
- die arithmetische Verneinung oder Negation (NEG)

Zunächst soviel: Bei den arithmetischen Operationen trennen sich einmal mehr die Geister! Bei Addition und Subtraktion sind alle Kombinationen von Registern und Speicherstellen möglich, die auch bei den logischen Vergleichen zulässig waren. Immer ein Operand muß also ein Register sein.

ADD, ADC, SUB, SBB

Lassen wir XXX für ADD, ADC, SUB oder SBB stehen, so gilt:

XXX	reg, reg	z.B.	ADD	AX, BX
			ADC	CL, DH
XXX	reg, mem	z.B.	SUB	SI, WordVar
			SBB	DH, ByteVar
XXX	mem, reg	z.B.	ADC	WordVar, DX
			SBB	ByteVar, AL
XXX	reg, const	z.B.	ADD	CX, $4711
			SUB	BH, $FF
XXX	mem, const	z.B.	ADD	WordVar, $0815
			SBB	ByteVar, $AB

Auch hier gilt: Der erstgenannte Operand ist gleichzeitig das Ziel der Operation.

Worin besteht nun der Unterschied zwischen ADD und ADC auf der einen und SUB und SBB auf der anderen Seite? ADD steht für *add* und addiert einfach die beiden Operanden. ADC dagegen heißt *add with carry* und addiert die beiden Operanden ebenfalls! Gleichzeitig wird jedoch je nach Stellung des Carry-Flags gegebenenfalls 1 addiert: ein gesetztes Carry-Flag bewirkt die Addition von 1, ein gelöschtes Carry-Flag unterdrückt diese zusätzliche Addition!

Wozu das Ganze? Stellen Sie sich folgendes vor: Sie haben im Register AX den Wert 60.000. In BX steht 48.573. Das Ergebnis von *ADD AX, BX* ist also 108.573, was deutlich größer ist als 65.535, die größte in einem 16-Bit-Register darstellbare Zahl. Nach allem, was wir bisher über Flags, Register und Operationen gelernt haben, können wir da-

von ausgehen, daß durch die Addition ein Flag gesetzt wird. So ist es auch: Die Addition der beiden Zahlen führt zu einem Registerinhalt von 43.037 in AX: der Differenz vom eigentlichen Ergebnis und der maximal darstellbaren Zahl. Gleichzeitig aber wird das Carry-Flag gesetzt, um anzuzeigen, daß für ein korrektes Darstellen des Ergebnisses eine weitere Stelle notwendig wäre.

Stellen wir dies einmal grafisch dar:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
C	1	1	1	0	1	0	1	0	0	1	1	0	0	0	0	0	AX
	1	0	1	1	1	1	0	1	1	0	1	1	1	1	0	1	BX
1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	0	1	AX

Bei der (arithmetischen) Addition der beiden Zahlen entsteht ein Überlauf. Dieses Bit kann nicht mehr im Register dargestellt werden, es wandert ins Carry-Flag und zeigt damit einen Überlauf an. Da in beiden Registern maximal 65.535 stehen kann, ist tatsächlich immer nur ein Bit Überlauf möglich, wozu das Carry-Flag ausreicht.

Der Datentyp *Word* ist nur ein Wort breit, kann also maximal 16 Bits aufnehmen, der Überlauf bleibt bestehen. Die in AX stehende Zahl ist somit falsch, was das Carry-Flag anzeigt.

Es gibt ja aber noch *LongInts*, also Zahlen, die 4 Bytes groß sind und somit 32 Bits aufnehmen könnten. Diese Zahlen sind jedoch in den 16-Bit-Registern des 8086 nicht mehr darstellbar, so daß sich die Prozessorentwickler entschlossen haben, Zahlen dieses Typs auf zwei Prozessorregister zu verteilen. Obwohl es nicht zwingend erforderlich ist, hat es sich doch stillschweigend eingebürgert, für die Darstellung einer *LongInt* das Registerpaar DX – AX zu benutzen. Die Schreibweise hierfür ist DX:AX. Benötigt man für die Addition z.B. eine weitere *LongInt*, so werden die beiden restlichen Register hierzu verwendet. Auch hier hat sich die Kombination CX:BX eingebürgert.

Zwei LongInt-Zahlen können also, wie auf der folgenden Seite gezeigt, dargestellt werden.

15 **DX** 0	15 **AX** 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 1 1 0 1 0 1 0 0 1 1 0 0 0 0 0
15 **CX** 0	15 **BX** 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 1 1 1 1 0 1 1 0 1 1 1 1 0 1

Im obigen Beispiel wurden wiederum die Zahlen 60.000 und 48.573 verwendet, allerdings sind es diesmal Daten vom Typ *LongInt*, die somit die Registerkombinationen DX:AX und CX:BX benutzen. Nun können wir wieder addieren. Allerdings auch nur 16-Bit-weise, da der 8086 nur über Befehle verfügt, die Worte bearbeiten können! Das Resultat ist also das gleiche wie eben: Addition von AX und BX mittels *ADD AX, BX* und Setzen des Carry-Flags.

```
15          DX          0     C  15          AX          0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   ?  1 1 1 0 1 0 1 0 0 1 1 0 0 0 0 0

15          CX          0        15          BX          0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1 0 1 1 1 1 0 1 1 0 1 1 1 1 0 1
```

```
15          DX          0        15          AX          0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1  1 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1
```

Was ist nun gewonnen? Wir haben das gleiche Ergebnis, als ob die beiden Zahlen im Word-Format vorlägen, da wir nur 16-Bit-Zahlen verarbeiten können. Hier kommt nun der Befehl ADC ins Spiel. Wie schon gesagt, macht ADC das gleiche wie ADD, nur daß das Carry-Flag berücksichtigt wird! Addieren wir nun mit ADC CX zu DX, so erhalten wir:

```
15          DX          0     C  15          AX          0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1  1 1 1 0 1 0 1 0 0 1 1 0 0 0 0 0

15          CX          0        15          BX          0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     1 0 1 1 1 1 0 1 1 0 1 1 1 1 0 1
```

```
15          DX          0        15          AX          0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1   0  1 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1
```

Bemerken Sie bitte zweierlei. Erstens: In DX steht nach *ADC DX, CX* ein gesetztes Bit 0, obwohl sowohl in DX als auch in CX 0 standen. Dieses gesetzte Bit ist das nach der letzten Addition gesetzte Bit aus dem Carry-Flag. Zweitens: Das Carry-Flag ist nach ADC gelöscht und zeigt damit an, daß die Addition von CX zu DX keinen Überlauf erzeugte! Lassen Sie sich bitte nicht dadurch verwirren, daß in den beiden letzten Schaubildern die Position des Carry-Flags unterschiedlich

ist. Es soll damit angedeutet werden, daß das Carry-Flag jeweils das Überlaufbit aus den betreffenden Registern aufnimmt! Wenn Sie nun aber das Registerpaar DX:AX als zusammengehörig betrachten, dann erhalten Sie genau das genaue Ergebnis: 108.573.

Fassen wir also noch einmal zusammen:

- Die Addition zweier *LongInt*-Zahlen ist nur über den Umweg der nacheinander ausgeführten Addition zweier 16-Bit-Teile der 32-Bit-*LongInt* möglich, da die Befehle des 8086 lediglich 16-Bit-Worte unterstützen.
- Bei der Addition wird zunächst das jeweils niederwertige Wort der beiden *LongInt*-Doppelworte mit dem Befehl ADD addiert. Dieser Befehl setzt, falls ein Überlauf stattfinden sollte, das Carry-Flag. Findet kein Überlauf statt, wird es gelöscht.
- Anschließend erfolgt die Addition der beiden höherwertigen Worte der *LongInt*-Zahlen. Hierbei wird der Befehl ADC verwendet, der nicht nur analog zu ADD zwei Register addiert, sondern auch den Zustand des Carry-Flags berücksichtigt. Auch hier wird analog zu ADD das Carry-Flag nach der Operation verändert.

Im Programm läßt sich das wie folgt ausdrücken:

```
add     ax,bx
adc     dx,cx
```

Analog der prozessorbedingten Aufteilung der 32-Bit-Zahlen in zwei 16-Bit-Register müssen auch die arithmetischen Operationen in zwei 16-Bitbefehle zerlegt werden. Hierbei hilft die Modifizierung des ADD-Befehls.

TIP

Auf diese Weise läßt sich sogar eine doppelt genaue *LongInt* realisieren. Wenn z.B. die eine »HugeInt« in den Registern CX:BX:DX:AX residiert und die andere in vier Wortregistern, so kann mittels der Sequenz

```
add     ax,WordVar1
adc     dx,WordVar2
adc     bx,WordVar3
adc     cx,WordVar4
```

das Gewünschte erreicht werden. Grenzen sind hierbei nur Ihrer Phantasie gesetzt (Versuchen Sie einmal, dies in einer Hochsprache zu realisieren! In Assembler vier Zeilen, in einer Hochsprache?).

HINWEIS

Wir haben bisher nur die 16-Bit-Register besprochen. Selbstverständlich arbeiten ADD, ADC, SUB und SBB auch mit den 8-Bit-Teilen AH, AL, BH, BL etc. zusammen. Allerdings ist der Sinn von ADC bei 8-Bit-Registern nicht ganz ersichtlich:

```
add     al,bl
adc     ah,bh
```

ist wohl besser (und schneller) mit *ADD AX, BX* realisiert! Dennoch funktioniert es, und man kann sich Anwendungsbeispiele ausdenken.

SUB, SBB

Ähnliches gilt für die Subtraktion! SUB, *Subtract*, ist der normale Subtraktionsbefehl, mit dem zwei 16-Bit-Worte voneinander subtrahiert werden können. Beachten Sie bitte hierbei, daß Subtraktionen nicht kommutativ sind, daß es also sehr wohl eine Rolle spielt, ob Sie *SUB AX, BX* oder *SUB BX, AX* verwenden. Auch hier gibt es, um 32-Bit-Zahlen behandeln zu können, eine Modifikation des SUB-Befehls: SBB, *Subtract With Borrow*. So läßt sich also eine 32-Bit-Subtraktion mit der Sequenz

```
sub     ax,bx
sbb     dx,cx
```

korrekt unter Berücksichtigung eines eventuell stattfindenden Überlaufs durchführen. Die Flags werden bei allen vier Befehlen analog dem CMP-Befehl gesetzt. Dies heißt:

- Zero-Flag, wenn das Ergebnis 0 war. Hierbei spielt es keine Rolle, ob die Operanden vorzeichenlos oder vorzeichenbehaftet interpretiert werden.
- Carry-Flag, wenn ein Überlauf stattgefunden hat. Das Carry-Flag bezieht sich aber nur auf vorzeichenlose Zahlen, bei denen Bit 7 bzw. Bit 15 als Teil der Zahl, nicht etwa als Vorzeichen zu interpretieren ist.
- Overflow-Flag, ebenfalls wenn ein Überlauf stattgefunden hat. Er bezieht sich hierbei wie üblich auf vorzeichenbehaftete Zahlen.
- Auxiliary-Flag, wenn ein Überlauf bei BCDs stattgefunden hat.
- Parity-Flag, wenn das Ergebnis der Operation eine gerade Anzahl Bits lieferte.
- Sign-Flag, wenn entweder Bit 15 bei Manipulationen mit 16-Bit-Registern gesetzt ist oder bei Verwendung von 8 Bit breiten Registern, wenn Bit 7 gesetzt ist.

ACHTUNG

Hier lauert eine Falle! Warum berücksichtigt ADC das Carry-Flag für einen Überlauf, kann aber dennoch vorzeichenbehaftete Zahlen verarbeiten? Beim CMP-Befehl haben wir doch gelernt, daß das Carry-Flag Überläufe bei vorzeichenlosen Zahlen anzeigt!

Das Problem ist gar keins! Wenn eine vorzeichenbehaftete Zahl im Registerpaar DX:AX steht und von ihr eine ebenfalls vorzeichenbehaftete Zahl in CX:BX abgezogen oder zu ihr addiert wird, dann stehen die Vorzeichen beider Zahlen in den Bits 15 der Register DX bzw. CX. In AX und BX ste-

hen nur die niederwertigen Anteile der LongInts. Diese sind ja immer vorzeichenlos! Also ist z.B. in der Sequenz

```
add     ax,bx
adc     dx,cx
```

die erste Addition, also ADD AX, BX, immer vorzeichenlos. Ein Überlauf wird daher vollkommen korrekt vom Carry-Flag angezeigt. Dennoch wird auch das Overflow-Flag gesetzt, da der Prozessor ja nicht wissen kann, daß dieser ADD-Befehl Teil einer Additionssequenz ist. Immerhin könnte es ja auch eine normale Addition zweier vorzeichenbehafteter Words sein! Wir aber wissen, daß dem nicht so ist.

Daher darf ADC auch niemals etwas anderes als das Carry-Flag berücksichtigen. Ob nun die beiden LongInts (als Ganzes) vorzeichenlos oder -behaftet sind, steht also in Bit 15 in CX und DX codiert. Diese werden im obigen Fall mit ADC addiert, das wiederum korrekt Carry-Flag und Overflow-Flag verändert. Nach der Sequenz muß also durch den Programmierer die Entscheidung getroffen werden, ob das Carry-Flag (vorzeichenlos) oder das Overflow-Flag (vorzeichenbehaftet) ausgewertet werden muß.

MUL

Der MUL-Befehl weicht davon gewaltig ab. Zunächst gibt es nur einen Multiplikationsbefehl, also keine Modifikation, mit der man Zahlen mit mehr als 16 Bit verarbeiten könnte. Dann läßt er sich nicht mit jedem Register kombinieren, sondern nur mit dem Akkumulator, also dem AX- bzw. dem AL-Register. Hier haben wir also wieder einen Fall, in dem ein Register eine Spezialfunktion hat.

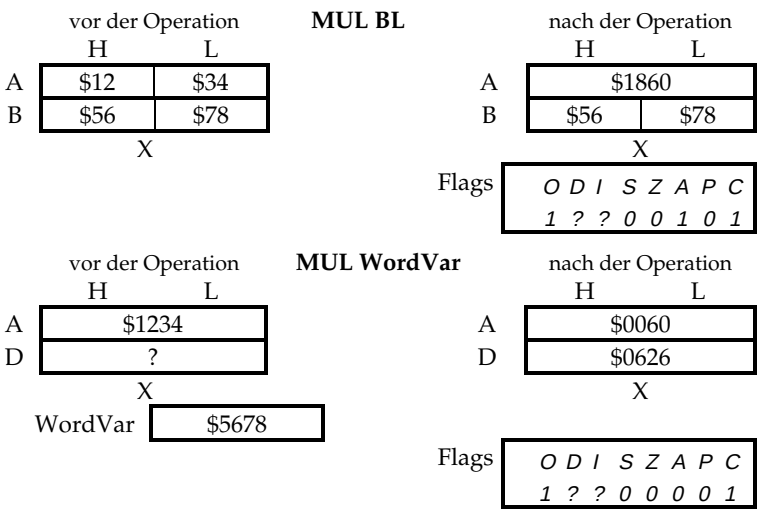
Wenn also grundsätzlich das AL- bzw. AX-Register bei MUL beteiligt sind, so ist auch nur ein Operand nötig. Ferner dürfen keine Konstanten benutzt werden! Der MUL-Befehl erlaubt daher:

- MUL reg z.B. MUL BL
MUL DX
- MUL mem z.B. MUL ByteVar
MUL WordVar

Doch wo wird das Multiplikationsergebnis abgelegt? Bei der Addition bzw. Subtraktion war es einfach. Selbst bei der Manipulation mit Maximalwerten konnte lediglich ein Überlauf stattfinden, der in einem Bit abgefangen werden konnte, wozu das Carry-Flag mißbraucht wurde. Bei der Multiplikation zweier Bytes z.B. können aber Werte entstehen, die sich nicht mehr durch einen Ein-Bit-Überlauf korrigieren lassen. Was also ist die größte Zahl, die z.B. bei der Multiplikation zweier Bytes miteinander entstehen kann? $255 \cdot 255$, ein Wert, der sich bequem in einem Word abfangen läßt.

Analog kann eine Multiplikation zweier Words lediglich zu einer *LongInt* führen, also einer Zahl, die in einem Registerpaar darstellbar ist. Das hat aber mehrere Konsequenzen. Erstens: Das Ergebnis einer Multiplikation nach *MUL BL* oder *MUL ByteVar* benutzt das gesamte AX-Register für das Ergebnis. Wenn vor der Multiplikation etwas in AH gestanden hat, ist es verloren. Die Multiplikation von zwei Worten mit *MUL BX* oder *MUL WordVar* benutzt das Registerpaar DX:AX. Auch hier geht der ursprüngliche Inhalt von DX verloren, egal, ob DX für das Ergebnis notwendig ist oder auf 0 gesetzt werden kann, weil das Produkt in ein *word* paßt!

Zweitens: Es kann niemals einen Überlauf geben. Da die Multiplikation zweier Bytes auf jeden Fall in einem Word und die Multiplikation zweier Words in einem Registerpaar durchgeführt werden kann, haben das Carry-Flag und das Overflow-Flag keine ihrer eigentlichen Aufgabe entsprechende Funktion. Dennoch werden sie verändert, hier also mißbraucht. So werden beide gelöscht, falls AH (bei Byte-Multiplikationen) oder DX (bei Word-Multiplikationen) nach dem Befehl 0 ist, andernfalls werden sie gesetzt. Das können Sie an den folgenden zwei Beispielen sehen:



MUL kann nur eine vorzeichenlose Multiplikation durchführen. Möchte man mit vorzeichenbehafteten Zahlen rechnen, so muß man einen Zwillingsbefehl von MUL, IMUL, benutzen. IMUL steht für *Integer Multiplication* und verwendet *Integer* statt *Words*, also Worte, deren Bit 15 das Vorzeichen repräsentiert (natürlich analog mit Bytes). Alles andere ist identisch mit MUL.

IMUL

HINWEIS Die Verwendung vorzeichenbehafteter Zahlen bedeutet keinen grundsätzlichen Unterschied zur Behandlung vorzeichenloser Zahlen. Auch hier gilt, daß das Ergebnis immer in den verwendeten Registern darstellbar ist, es also niemals zu einem Überlauf kommen kann. Daher haben, wie bei MUL auch, das Carry- und das Overflow-Flag wieder eine zweckentfremdete Bedeutung. Diese ist jedoch bei IMUL etwas anders als bei MUL. Während bei MUL das gesetzte Carry- oder Overflow-Flag anzeigte, daß AH bzw. DX nach der Multiplikation 0 ist, das Ergebnis also in ein Byte- bzw. Wortregister paßte, so wird bei der vorzeichenbehafteten Multiplikation IMUL durch ein gesetztes Carry- bzw. Overflow-Flag angezeigt, daß keine Vorzeichenerweiterung in das AH- bzw. DX-Register erfolgte. Was aber heißt das?

Nehmen Sie z.B. die Zahl -2. Als Byte, hexadezimal dargestellt, ist dies $FE. Das gesetzte Bit 7 zeigt, daß die Zahl negativ ist und somit als sogenanntes 2er-Komplement aufgefaßt werden muß (falls Sie nicht wissen, was das ist, schlagen Sie bitte im Anhang A nach). Multipliziert man diese Zahl mit 2, so sollte sich -4 oder, hexadezimal in Byteform, $FC ergeben, was nach dem eben Gesagten vollständig in das Byte-Register AL paßt. Was muß nun mit AH passieren?

Das Ergebnis der Multiplikation zweier Byte-Zahlen ist immer ein Word. Daher muß auch das Ergebnis der Multiplikation von -2 mit 2 als Word dargestellt werden: $FFFC. Denn in Worten ist Bit 15 das Vorzeichen und das 2er-Komplement vom Wort 4 ist $7FFC. Also muß die Multiplikation von -2 mit 2 tatsächlich $FFFC ergeben, es muß also eine sogenannte »Vorzeichenerweiterung« von Bit 7 von AL in AX erfolgen. Daß dies bei dieser Multiplikation erfolgte, zeigen *Carry* und *Overflow* an, indem sie gelöscht sind. Anders ausgedrückt: ein gelöschtes Carry- bzw. Overflow-Flag zeigt an, daß AH $FF oder DX $FFFF enthält – und bedeutet damit praktisch das gleiche wie bei vorzeichenlosen Zahlen: das Ergebnis paßt vollständig in ein Byte- bzw. Wortregister.

Multiplizieren wir dagegen -127, also $81, mit 2, so erfolgt keine Vorzeichenerweiterung, da das Ergebnis der Multiplikation in Betragform, also ohne Vorzeichen, schon nicht mehr in das Byte-Register AL paßt. Eine Vorzeichenerweiterung des (eigentlich leeren) AH-Registers hat nicht zu erfolgen, das Carry- bzw. Overflow-Flag wird damit gesetzt.

TIP Betrachten Sie also das Carry- bzw. Overflow-Flag als Indikatoren dafür, daß Sie bei Multiplikationen von Bytes nach der Operation tatsächlich mit Bytes weiterarbeiten, den Inhalt von AH also vergessen können. Denn bei vorzeichenlosen Zahlen zeigt ein gelöschtes Carry- bzw. Overflow-Flag an, daß AH 0 ist, das Ergebnis also in AL paßt. Aber auch bei vorzeichenbehafteten Zahlen zeigt ein gelöschtes Car-

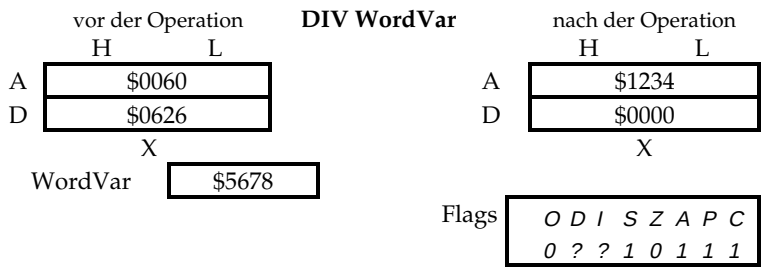
ry- bzw. Overflow-Flag an, daß mit Bytes korrekt weitergearbeitet werden kann, daß das Vorzeichen in Bit 15 von AX ebenfalls in Bit 7 von AL anzutreffen ist! Analoges gilt natürlich für die Multiplikation von Words und dem Register AX bzw. dem Registerpaar DX:AX.

DIV

DIV steht für *Divide* und ist die vierte Grundrechenart, die der 8086 beherrscht. DIV lehnt sich eng an die Eigenschaften seines Gegenparts MUL an. So gibt es auch hier keine Möglichkeit, Zahlen mit mehr als 16 Bit Breite zu verarbeiten. Auch hier kann nur das Register AX bzw. das Registerpaar DX:AX benutzt werden, und zwar in umgekehrter Art und Weise wie bei der Multiplikation. So steht der Dividend entweder in AX, wenn der Divisor ein Byte in einer Speicherstelle oder einem Byte-Register ist, oder in DX:AX, wenn es sich um einen Word-Divisor handelt.

Der 8086 kennt keine Realzahlen, so daß eine Division für den Prozessor immer eine Integerdivision ist. Dies bedeutet, daß nach der Division immer eine Integer als Divisionsergebnis entsteht sowie ein Rest, der bei der Division übrig bleibt. Im Falle der Division eines Wortes (AX) durch ein Byte steht dann das Ergebnis in AL, der Divisionsrest in AH. Wurde eine LongInt (DX:AX) durch ein Wort dividiert, so steht das Ergebnis in AX, der Rest in DX.

Auch hier ein Beispiel:



IDIV

Ganz analog zu MUL gibt es auch hier einen Befehl, der mit vorzeichenbehafteten Zahlen verwendet wird: IDIV oder *Integerdivision*. Er arbeitet ansonsten ganz genau wie DIV, verwendet also die Register AX bzw. DX:AX.

ACHTUNG

Nach IDIV hat der Divisionsrest in AH/DX das gleiche Vorzeichen wie der Dividend. So ist der Rest negativ, falls die zu dividierende Zahl negativ ist und umgekehrt! Das macht auch Sinn, denn der Divisionsrest muß immer zum Produkt aus Ergebnis und Divisor addiert werden, um den Dividenden zurückzuerhalten. Beispiel: 496 IDIV 28 = 17, Rest 20, so daß 28 · 17 + 20 wieder 496 ist. Dagegen muß nach

-496 IDIV 28 der Divisionsrest -20 sein, da umgekehrt 20 von 28 · (-17) abgezogen, also (-20) addiert werden muß, um wieder -496 zu erhalten: 28 · (-17) + (-20) = -496.

Was passiert, wenn durch 0 dividiert werden soll? Es wird nicht einfach das Zero-Flag gesetzt, wie Sie nun vermuten könnten! Eine Division durch 0 ist für den Prozessor ein schwerwiegendes Ereignis; so gravierend, daß es nicht einfach mit einer lapidaren Flagmanipulation angezeigt werden soll. Der Prozessor behandelt diesen Programmierfehler dadurch, daß er eine Ausnahmebedingung schafft. Dies bewirkt, daß ein sogenannter Interrupt ausgelöst wird.

Bitte gedulden Sie sich mit einer Erklärung, was ein Interrupt ist, bis zum zweiten Teil des Buches, in dem wir darauf eingehen werden! Halten wir zunächst also lediglich fest, daß eine Division durch 0 einen Prozeß in Gang setzt, der die Weiterverarbeitung des Programms an dieser Stelle fürs erste unterbricht.

HINWEIS Lassen Sie sich nicht dadurch in die Irre führen, daß das »I« in IDIV für *Integer* steht! Beide Operationen sind Integerdivisionen, können als Resultat also nur Ganzzahlen erzeugen, keine gebrochenen Realzahlen. *Integer* soll hier lediglich anzeigen, daß IDIV mit *Integern*, also vorzeichenbehafteten Zahlen arbeitet, DIV dagegen mit *Words*, also vorzeichenlosen Zahlen.

Und was machen die Flags nach DIV/IDIV? In der einschlägigen Literatur steht, daß die Flags nicht verändert werden oder sich mit ihnen teilweise die Divisionsergebnisse interpretieren lassen. So z.B. mit Hilfe des Zero-Flags, das anzeigen soll, wenn der Divisionsrest 0 ist. Laut Herstellerangaben des 8086 spielen die Flags bei der Interpretation des Ergebnisses keine Rolle. Das scheint auch zu stimmen, da sich die Veränderungen der Flags nach DIV/IDIV nicht in das bisher gewohnte Schema pressen lassen. So zeigt z.B. ein gesetztes Sign-Flag nach IDIV in keiner Weise an, ob Bit 7 bzw. Bit 15 gesetzt ist, das Ergebnis also negativ ist. Auch nach DIV, also einer vorzeichenlosen Division, wird es teilweise gesetzt. Seltsam auch das Verhalten des Carry-Flags. Nach DIV scheint es immer gesetzt zu sein, nach IDIV manchmal. Und das Zero-Flag verhält sich am komischsten: durch DIV wird es offensichtlich nicht verändert, nach IDIV jedoch sehr wohl! Nun aber nicht etwa, wie man meinen könnte, wenn Rest oder Ergebnis 0 sind. Mitnichten: Mit dem Inhalt von irgendwelchen Registern scheint das Zero-Flag nicht zu korrespondieren!

Der Prozessor benutzt sie offensichtlich bei der Ausführung des Microcodes für die Divisionen. Als Microcode bezeichnet man die Abfolge fest verdrahteter arithmetischer und logischer Verknüpfungen auf dem Prozessorchip, die der Prozessor praktisch wie kleine Unterprogramme einsetzt, wenn ein (»richtiger«) Prozessorbefehl wie DIV oder IDIV abgearbeitet werden soll. Was ich finden konnte, ist folgendes:

- Das Carry-Flag zeigt an, ob der Divisor positiv ist. Das bedeutet, daß das Carry-Flag immer dann nach DIV/IDIV gesetzt ist, wenn die Zahl, die dividert werden sollte, positiv war. Logischerweise muß also nach DIV das Carry-Flag immer gesetzt sein, da DIV nur vorzeichenlose (und damit »positive«) Zahlen dividieren kann.
- Das Sign-Flag signalisiert die gleiche Information für den Dividenden. Ist also die Zahl, durch die geteilt wird, positiv, so ist das Sign-Flag gesetzt, andernfalls gelöscht! Nun könnte man der Meinung sein, daß analog zum Carry-Flag bei DIV also das Sign-Flag immer gesetzt sein müßte. Dem ist aber nicht so: Bei DIV signalisiert das Sign-Flag ein gesetztes Vorzeichenbit 7 bzw. 15 des Dividenden, obwohl er bei der Division als positiv betrachtet wird.
- Das Zero-Flag zeigt nun mitnichten an, daß Divisionsrest oder Ergebnis 0 sind! Nein, es gibt nur einen Fall, in dem das Zero-Flag gesetzt wird, und zwar wenn das Vorzeichen des Restes negativ sein müßte, weil der Divisor auch negativ ist, es aber nicht sein kann, weil der Rest 0 und somit per definitionem positiv ist! Logische Konsequenz: Das Zero-Flag wird nur gesetzt, falls nach einer Division einer negativen Zahl mittels IDIV ein 0-Rest herauskommt.
- Das Overflow-Flag wird bei IDIV niemals verwendet, zumindest war es nach vorzeichenbehafteten Divisionen bisher immer gelöscht. Nach vorzeichenlosen Divisionen mittels DIV dagegen zeigt das Overflow-Flag ein nicht bekanntes Eigenleben.
- Paritiy- und Auxiliary-Flag. Was diese Flags betrifft, so konnte ich bisher wirklich keine Regel finden, nach der sie gesetzt bzw. gelöscht werden. Aber sie werden beeinflußt!

Sie sehen, daß die Interpretation der Flags nach DIV/IDIV tatsächlich nicht einfach ist. Vermeiden Sie es, sie zu benutzen, wenn Sie wenig Kenntnisse und/oder Erfahrung in Assemblerprogrammierung haben. Denn wie wir gesehen haben, kommt den Flags nicht nur eine gänzlich andere Bedeutung als die gewohnte zu. Sie werden auch genau entgegengesetzt manipuliert! Denn immerhin sollte man erwarten, daß das Sign-Flag gelöscht ist, um anzuzeigen, daß der Divisor positiv ist – tatsächlich ist es aber gesetzt.

TIP

Da es keinen ernsthaften Grund gibt, die Flags direkt nach der Division auszuwerten, sollten Sie es erst gar nicht versuchen. Das Ergebnis einer Division ist immer korrekt in den Registern AL/AH bzw. AX/DX verzeichnet. Wenn eine Division einmal Unsinn liefern würde, weil durch 0 dividiert wird, so wird mit der Interrupt-Auslösung sowieso eine härtere Gangart des Prozessors eingeschlagen als die bloße Flagmanipulation.

Wenn Sie jedoch aus welchen Gründen auch immer nach der Division etwas über die Art des Ergebnisses wissen müssen, so verwenden Sie zunächst einen Befehl, der die Flags in Abhängigkeit von der Zahl korrekt setzt, bevor Sie sie austesten. So z.B. *OR AL, AL*, mit dem die Zahl nicht verändert, aber die Flags korrekt gesetzt werden.

ACHTUNG Divisionen sind nicht so einfach, wie es zunächst den Anschein hat. Während man bei Multiplikationen davon ausgehen kann, daß das Ergebnis keinen Überlauf erzeugen kann, ist dies bei Divisionen nur nach dem ersten Anschein der Fall! Zwar ist ein Word dividiert durch ein Byte in der Regel auch ein Byte, weshalb der Quotient in AL Platz hat und der Rest in AH; analoges gilt für die Division von LongInt durch Word. Dennoch kann es auch sein, daß das Divisionsergebnis nicht mehr in das Byte paßt: dann nämlich, wenn der Dividend klein genug ist. Dividieren Sie z.B. 40.000 durch 200, so erhalten Sie 200 – das paßt in ein Byte. 40.000 dividert durch 100 dagegen paßt nicht mehr!

Was passiert also in einem solchen Moment? Ganz einfach: Dieses Ergebnis ist für den Prozessor genauso eine Ausnahmesituation wie die Division durch 0. Daher tut er auch das gleiche: Er löst einen Interrupt aus und unterbricht damit zunächst einmal die Ausführung des Programms!

INC, DEC Eine ganz einfache Addition ist INC. INC, *Increment*, addiert einfach zu einem Byte oder Wort den Wert 1. Ziel und Operand sind, wie bei jeder Addition, entweder Byte- oder Wortregister oder -speicherstellen. DEC, *Decrement*, ist das Pendant, also die Subtraktion von 1 von dem entsprechenden Operanden.

Stellen wir XXX für INC oder DEC dar, so gilt:

- XXX reg z.B. INC AX
  DEC CL
- XXX mem z.B. INC WordVar
  DEC ByteVar

Beide Befehle sind äußerst schnell und werden vorwiegend eingesetzt, um Schleifen zu steuern. Auch bei diesen Befehlen werden Flags verändert, und zwar so wie bei jeder Addition bzw. Subtraktion:

- Das Auxiliary-Flag wird gesetzt, wenn ein Übertrag aus Bit 3 in Bit 4 erfolgt; denken Sie als Begründung an BCDs!
- Das Parity-Flag zeigt an, ob die Anzahl der gesetzten Bits gerade oder ungerade ist.
- Das Zero-Flag zeigt an, ob der Inhalt der in-/dekrementierten Variablen/Register 0 ist.

- Das Sign-Flag spiegelt ein gesetztes Bit 7 bzw. Bit 15 wider, je nachdem, ob Bytes oder Worte in-/dekrementiert werden.
- Das Overflow-Flag zeigt einen Übertrag von Bit 6 in Bit 7 bzw. Bit 14 in Bit 15 an und stellt somit, wie bei vorzeichenbehafteter Addition bzw. Subtraktion einen Überlauf in das Vorzeichen fest.

Das Carry-Flag wird von INC/DEC nicht verändert.

ACHTUNG

Die Inkrementation/Dekrementation beeinflußt das Carry-Flag in keiner Weise. Falls Sie also diese Befehle in Schleifen verwenden, dürfen Sie niemals mit JC oder bedingten Sprungbefehlen arbeiten, die das Carry-Flag prüfen oder berücksichtigen! Andernfalls erfolgt keine Programmverzweigung. Die Prüfung des Zustands des Carry-Flags bei INC/DEC führt zu schwer auffindbaren Programmierfehlern.

TIP

Benutzen Sie daher INC/DEC nur in Verbindung mit einer Prüfung, ob der Zähler, der inkrementiert bzw. dekrementiert wird, 0 wird oder nicht. Das Zero-Flag ist das einzig zuverlässige und sinnvolle Flag bei diesen Befehlen, wenn auch manchmal das Sign- und/oder das Overflow-Flag zum Einsatz kommen können!

NEG

Bei der Besprechung der arithmetischen Befehle fehlt nur noch eine Operation: die arithmetische Negation NEG. *Negate* arbeitet grundsätzlich anders als NOT! Während NOT das sogenannte 1er-Komplement einer Zahl bildet, also die Bits, aus denen die Zahl besteht, lediglich umkehrt, so bildet NEG das 2er-Komplement, in dem sich die Bits gegenseitig beeinflussen. Das heißt, daß je nach Stellung der Bits das jeweils folgende (obere, höherwertige) Bit gesetzt oder gelöscht wird.

Das war eine komplizierte Erklärung eines für den Programmierer einfachen Sachverhaltes: NEG bildet eine Zahl, die den gleichen Betrag, aber ein entgegengesetztes Vorzeichen hat. Aus 2 wird -2, aus -4711 wird 4711. Daß der Sachverhalt für den Prozessor aber in Wirklichkeit nicht so einfach ist, können Sie im Anhang A bei der Besprechung der Zahlensysteme nachlesen!

Für uns ist die Bildung des negativen Werts einfach eine Subtraktion nach der Art AX = Const – AX. Im Falle von Bytes ist Const = 256, im Falle von Words 65536. Diese beiden Konstanten sind aber nicht mehr als Byte bzw. Word darstellbar; tatsächlich werden sie im Register als 0 gehalten.

Das fehlende Bit wird wiederum, wie bei jeder Subtraktion, »ausgeborgt«. Daher muß hier wie dort das Carry-Flag gesetzt werden, um dieses Ausborgen einer nicht vorhandenen Stelle anzuzeigen.

Alle anderen Flagveränderungen ergeben sich aus einer normalen Subtraktion:

- Das Overflow-Flag spielt keine Rolle, da NEG lediglich das Vorzeichen der Zahl ändert, nicht aber ihren Betrag, so daß kein Überlauf stattfinden kann.
- Das Carry-Flag ist, wie eben begründet, immer gesetzt.
- Das Zero-Flag signalisiert, daß versucht wurde, das Vorzeichen von 0 zu ändern, was selbstverständlich nicht möglich ist.
- Das Sign-Flag enthält die Stellung des Vorzeichenbits 7 (bei Bytes) bzw. 15 (bei Words).
- Das Parity und das Auxiliary-Flag gehen ihrer üblichen Tätigkeit nach.

2.8 Stringoperationen

Für den Prozessor sind alles Daten! Ob nun Bytes oder Worte – Inhalte von Registern und Speicherstellen sind Daten und somit entweder in einem Byte mit 8 Bit oder in einem Wort mit 16 Bit darstellbar.

Dieses Prinzip gilt immer. Der Buchstabe »A« ist für den Prozessor nichts anderes als die Bitfolge 0100 0001. Diese Bitfolge ist aber auch die Repräsentation des Bytes 65. Somit besteht für den Prozessor kein Unterschied zwischen 65 und »A«!

Warum diese Einführung? Weil der Name *Stringoperationen* für die Befehle, die im folgenden beschrieben werden sollen, eigentlich falsch ist. Als Hochsprachenprogrammierer stellt man sich unter einem String eine Folge von Zeichen vor. So ist in Pascal eine Variable *S : String* ein Speicherbereich, in dem die Buchstaben eines Wortes, Satzes oder gar Abschnittes gespeichert sein können.

Tatsächlich aber interpretiert nur der Hochsprachencompiler die an der betreffenden Stelle stehenden Bitfolgen als Buchstaben oder genauer gesagt als ASCII-Zeichen. Die Maschinenbefehle, in die ein Hochsprachenquelltext übersetzt wird, tun dies ebensowenig wie der Assembler.

Lösen wir uns also von dem Gedanken an Buchstabenfolgen, wenn wir über Strings reden. Für den Assembler ist ein String einfach eine Folge von Bytes oder Worten. Die Befehle, die mit Strings umgehen, also die Stringoperationen, verwalten und bearbeiten daher in Wirklichkeit genau diese Folgen von Bytes oder Worten.

Doch dies tun sie zum Teil sehr effektiv. So gibt es Stringbefehle, mit denen in kürzester Zeit ganze Segmente nach einem bestimmten Zeichen durchsucht oder Bereiche kopiert werden können. Aber sehen Sie selbst!

Im einzelnen stehen dem Programmierer folgende Möglichkeiten zur Verfügung:

- Das Laden eines Wertes aus einem String (LODS, LODSB, LODSW)
- das Speichern eines Wertes in einen String (STOS, STOSB, STOSW)
- der Vergleich eines Wertes mit einem Stringinhalt (SCAS, SCASB, SCASW)
- ein Datenaustausch zwischen zwei Strings (MOVS, MOVSB, MOVSW)
- ein Vergleich zweier Strings (CMPS, CMPSB, CMPSW)
- eine Wiederholung von Stringoperationen (REP, REPcc)

LODS, STOS

Wie Sie anhand der *Mnemonics* sehen können, verfügt der Prozessor bei allen Stringbefehlen außer den Wiederholungsbefehlen über jeweils drei Ausführungen des jeweiligen Befehls: xxx, xxxB und xxxW. In den Namen steht »B« für Byte und »W« für Word.

Um mit Strings zu arbeiten, muß dem Prozessor mitgeteilt werden, wo im Speicher sich dieser String nun eigentlich befindet. Bei Zugriffen auf Speicherstellen wurde den Befehlen, die wir bisher verwendet haben, immer die Adresse der Variablen als Operand übergeben, z.B. in *CMP AX, WordVar*.

Bei Strings ist das anders. Hier scheint der Prozessor die Adresse des Strings irgendwie zu kennen, denn die Stringbefehle verfügen über keinen Operanden. Woher also weiß der Prozessor z.B. beim Befehl LODSW, woher und wohin er ein Wort laden soll? Die Antwort ist für einen der Operanden einfach zu klären: Stringoperationen, die einen Wert laden oder speichern, also LODSx und STOSx, verwenden immer den Akkumulator! Das heißt, daß nach LODSB ein Byte aus dem String in AL geladen wird. Durch STOSW wird der Inhalt von AX in den String kopiert. Auch hier haben wir also ein Beispiel für die Spezialisierung der vier Rechenregister.

Doch wo liegt der String? Bei der Besprechung der Register haben wir noch zwei Indexregister kennengelernt: das *Source-Indexregister* SI und das *Destination-Index*register DI. Beide geben einen Offset auf den Bereich im Segment an, an dem der String steht. Aber woher kennen sie das Segment? Das klärt das DS-Register, also das Register, in dem die Adresse des Segments steht, das die Daten beinhaltet. Also: die Adresse des Strings kann der Prozessor eindeutig feststellen. Im Falle des Ladens eines Bytes oder Worts aus dem String in den Akkumulator ist es DS:SI, im Falle des Speicherns eines Bytes oder Worts aus dem Akkumulator in den String ist es DS:DI.

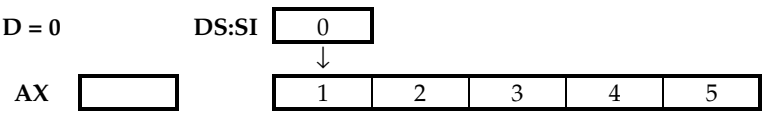
Warum diese Umstände? Wäre es nicht einfacher, die Adresse wie bei CMP oder den anderen Befehlen explizit zu übergeben, etwa nach *LODSW StringVar* oder *STOSB StringVar*? Antwort: Nein! Denn STOSx und LODSx machen außer dem eigentlichen Kopieren eines Wertes in den String bzw. aus dem String noch etwas anderes: sie inkrementieren den Inhalt von SI bzw. DI. Das heißt, daß nach einem LODSx-Befehl in SI der Offset auf den folgenden Wert zeigt, so daß mit einem weiteren LODSx-Befehl dieser gleich geladen werden kann, ohne daß irgend jemand sich um die Adressierung kümmern müßte.

Analog dazu inkrementiert STOSx das Register DI nach jedem Speicherbefehl. Beide Befehle »wissen« dabei, ob sie das jeweilige Register um 1 erhöhen müssen, weil nur ein Byte übertragen wurde, oder ob nach einer Wortübertragung der Inhalt der Indexregister um 2 erhöht werden muß. Auch dies erfolgt, ohne daß der Programmierer etwas dafür tun müßte.

Direction-Flag

Wenn ich nun die ganze Zeit von Inkrementieren und Erhöhen der Adressen in den Indexregistern rede, ist das nicht ganz korrekt! Denn man könnte ja auch auf den Gedanken kommen, Strings »von hinten« abarbeiten zu müssen oder wollen. Dann sollte die Adresse aber nicht inkrementiert, sondern eher dekrementiert werden. Der Prozessor unterstützt tatsächlich beide Möglichkeiten. Dabei hilft uns das *Direction-Flag*, das mit den Befehlen STD gesetzt und mit CLD gelöscht werden kann.

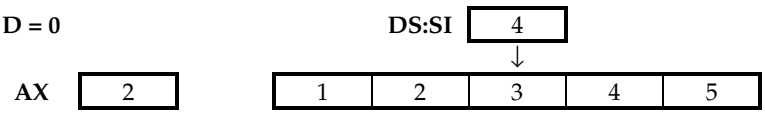
Ist das Direction-Flag gelöscht, so inkrementiert der Prozessor die Adressen in SI und DI, ist es gesetzt, dekrementiert er sie. Schauen wir uns dies einmal schematisch an:



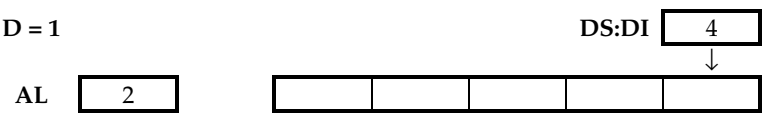
Das Direction-Flag ist 0, so daß die Befehle die in DS:SI stehende Adresse inkrementieren. Wir werden LODSW benutzen, so daß ein Wort aus dem String in AX kopiert wird. Anfänglich zeigt DS:SI auf eine Adresse, an der der String beginnt, hier die Adresse 0! Nach einem LODSW sieht die Situation wie auf der folgenden Seite gezeigt aus.

Das Wort an der Adresse in DS:SI wurde in AX kopiert und die Adresse korrekt um zwei Bytes (= ein Wort) erhöht. DS:SI zeigt somit auf das nächste Wort.

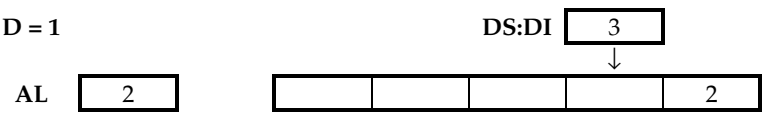
Der nächste LODSW-Befehl wirkt nun auf diese Speicherstelle:



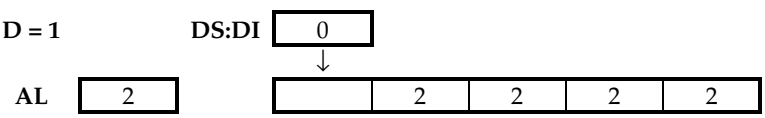
Man kann also mit konsekutiven LODSW-Befehlen ohne jegliche Adressen- oder Operandenangabe den String wortweise auslesen. Auch STOSx arbeitet so. Schauen wir uns allerdings hier einmal an, was beim byteweisen Abspeichern mit gesetztem Direction-Flag passiert.



Beim Abspeichern wird das Destination-Indexregister verwendet. Dort muß nun die Adresse des Stringendes stehen, da ja durch Direction-Flag = 1 bewirkt wird, daß nach dem Speichern die Adresse dekrementiert wird.



Beachten Sie bitte, daß die Adresse nur byteweise, also um 1, dekrementiert wird, weil auch nur Bytes übertragen werden. Nach drei weiteren STOSB-Befehlen haben wir dann folgendes Bild:



Alle Stringpositionen wurden mit dem Wert 2 belegt.

MOVS

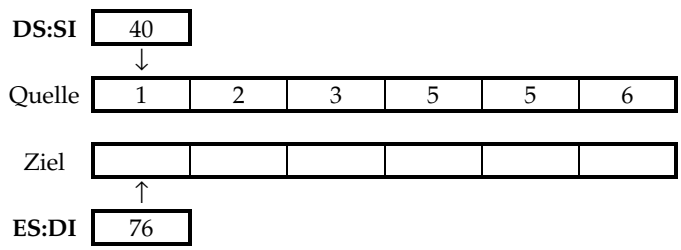
Mit LODSx holen wir uns einen Wert aus einem String, mit STOSx speichern wir einen in einem String ab. Das kann man ja ganz gut verwenden, um Stringinhalte zu kopieren, etwa mit:

```
lodsw
stosw
```

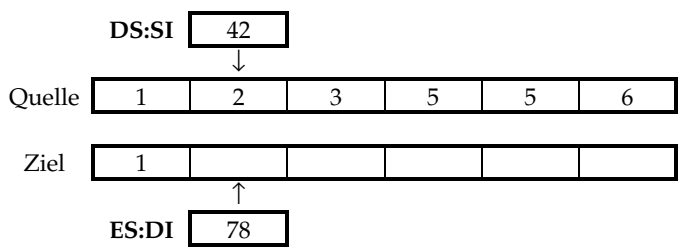
Da LODSW ein anderes Indexregister verwendet als STOSW, dürfte das keine Probleme bereiten. Richtig! Aber es gibt da noch etwas besseres: MOVSx.

Die MOVSx-Befehle machen genau das, ohne jedoch über das AL- bzw. AX-Register gehen zu müssen! MOVSW kopiert ein Wort aus einem String, in-/dekrementiert anhand des Direction-Flags das Source-Indexregister, schreibt den Wert in den Zielstring und in-/dekrementiert auch das für diesen String zuständige Destination-Indexregister. Es kommt noch besser: Während STOSx und LODSx als Segmentregister DS verwenden, kann man bei MOVSx ein jeweils für Quell- und Zielstring unterschiedliches Segmentregister angeben. Möglich wird dies durch die Einbeziehung von ES, einem Segmentregister, das üblicherweise nicht verwendet wird.

MOVSx kopiert also ein Byte oder Wort aus DS:SI in ES:DI. Auch hier ein Beispiel: Das Direction-Flag sei gelöscht, so daß inkrementiert wird. Beachten Sie bitte, daß die Adressen in DS:SI und ES:DI unterschiedlich sind (und es auch sinnvollerweise sein sollten).



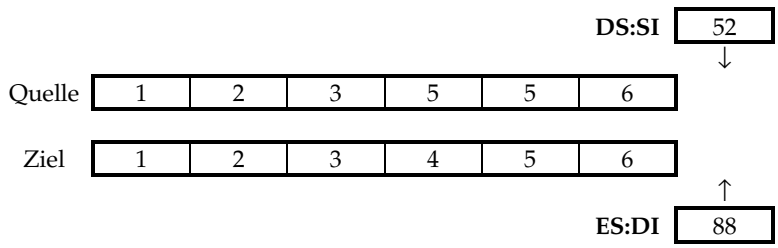
Der erste MOVSW-Befehl führt zu folgendem Bild:



Nach der Sequenz

```
movsw
movsw
movsw
movsw
movsw
```

wurde der Quellstring vollständig kopiert:



REP, REPZ, REPNZ, REPE, REPNE

Doch auch dies läßt sich noch vereinfachen. So stellt der 8086 drei sogenannte Repetierbefehle zur Verfügung, mit denen die Stringbefehle automatisch wiederholt werden können. Es handelt sich hierbei um:

- REP, *repeat until cx=0*, den allgemeinen Wiederholungsbefehl. Er wiederholt den ihm folgenden Stringbefehl genau so oft, wie im *Count-Register* CX steht.
- REPZ, *repeat until cx=0 or zero flag set*. Verglichen mit dem REP-Befehl prüft REPZ also zwei Bedingungen: ob CX = 0 oder das Zero-Flag gesetzt ist. In beiden Fällen wird die Wiederholung abgebrochen.
- REPNZ, *repeat until cx=0 or zero flag not set*. Dieser Befehl stellt das Gegenstück zu REPNZ dar und bricht die Wiederholung ab, falls das Zero-Flag gelöscht wird oder CX = 0 ist.

REPE und REPNE stellen wieder Redundanzen des Assemblers dar. Sie entsprechen vollständig REPZ und REPNZ.

ACHTUNG

Die Repetierbefehle arbeiten nur mit den Stringbefehlen zusammen. Die Verwendung z.B. mit MOV oder anderen Operationen ist nicht möglich. Dies ist auch ganz logisch, denn nur bei den Stringbefehlen wird keine explizite Adresse angegeben; nur diese Befehle benutzen SI und DI und in-/dekrementieren automatisch deren Inhalt. Was also zunächst etwas umständlich aussah, hat durchaus seine wohlüberlegten Gründe!

TIP

Das Kopieren der beiden Strings im obigen Beispiel erreichen Sie also am einfachsten und effektivsten mit der Sequenz:

```
mov     cx,6
rep     movsw
```

ACHTUNG REP prüft nur das CX-Register und dekrementiert dessen Inhalt. Die anderen vier (in Wirklichkeit nur zwei) Befehle prüfen auch das Zero-Flag. Daher muß die Operation, die durch REPx wiederholt wird, in irgendeiner Weise Einfluß auf dieses Flag nehmen. Weder LODSx noch STOSx oder MOVSx tun dies. Daher hat die Verwendung von REPZ, REPNZ, REPE und REPNE mit MOVSx, STOSx und LODSx keinen Sinn. Im Gegenteil: Ein zufällig gesetztes Zero-Flag bewirkt den sofortigen Abbruch der Wiederholung im Falle von REPZ MOVSW. Es ist vor allem für den Anfänger nicht leicht, den Grund für das dadurch unterbleibende Kopieren festzustellen! Aus diesem Grunde unterbinden auch die meisten Assembler solche Kombinationen.

Diese Befehle machen dagegen in Verbindung mit den folgenden Stringbefehlen Sinn:

SCAS, SCASB, SCASW SCAS führt einen Vergleich des Akkumulators mit einem Stringinhalt durch. Dieser Befehl arbeitet also analog zu CMP, nur daß eben ein Wort oder Byte aus einem String mit AX bzw. AL verglichen wird. Analog zu CMP werden dabei jedoch weder der Inhalt von AX/AL noch des Strings verändert, sondern lediglich Flags manipuliert.

Die Adresse des zu durchsuchenden Strings muß sich in ES:DI befinden. Auch hier steuert das Direction-Flag, in welcher Richtung der String durchsucht wird: bei gelöschtem Flag von vorn nach hinten, also zu steigenden Adressen, andernfalls genau andersherum.

TIP SCAS subtrahiert genau wie *CMP AX, WortVar* oder *CMP AL, ByteVar* den Wert des Strings an der in ES:DI stehenden Stelle vom Akkumulator. Entsprechend werden die Flags gesetzt – so auch das Zero-Flag, das somit in eine Prüfung mittels REPZ oder REPNZ eingebunden werden kann.

Auf diese Weise durchsucht *REPZ SCASW* den durch ES:DI adressierten String so lange wortweise, bis der Inhalt mit dem Wert in AX übereinstimmt, maximal jedoch CX-mal. Das Abbruchkriterium ist somit, daß das »Suchwort« in AX im String gefunden wurde.

REPNZ SCASB dagegen durchsucht den String byteweise, solange AL einen anderen Wert als den Inhalt der aktuellen Stringposition enthält, bricht also genau dann ab, wenn das »Suchbyte« im String nicht mehr gefunden wird.

CMPS, CMPSB, CMPSW Wer CMP und SCAS sagt, der muß auch CMPS sagen können! Denn mit CMP können Werte verglichen und mit SCAS Strings durchsucht werden. Glücklicherweise haben die Intel-Konstrukteure dem 8086 den sehr mächtigen Befehl CMPS mit in die Wiege gelegt, der eine logische Lücke füllt.

Diese müßte, falls es CMPS nicht gäbe, wie folgt geschlossen werden:

```
lodsw
cmp       ax,es:[di]
je        Weiter
```

CMPS vergleicht also einen String mit einem anderen. Das heißt, daß CMPS so wie SCAS arbeitet, jedoch nicht mit einem fixen Wert in AL/AX vergleicht, sondern mit einer variierenden Position in einem zweiten String. Rufen Sie sich bitte das Schaubild von MOVS in Erinnerung. Dort wurde der Inhalt eines Strings in einen anderen kopiert. Dies konnte geschehen, da der eine String über DS:SI, der andere über ES:DI adressiert wurde. Genauso ist es hier: CMPS vergleicht zwei Strings, indem vom Inhalt aus DS:SI der Inhalt aus ES:DI abgezogen wird.

Analog zu CMP wird dabei allerdings das Ergebnis verworfen, es werden nur, wie bei CMP auch, die Flags aktualisiert. Beachten Sie bitte auch, daß der Inhalt des Zielstrings vom Quellstring abgezogen wird, also genau konträr zu CMP! In Kombination mit REPE oder REPNE läßt sich dieser Befehl ebenfalls sehr flexibel einsetzen: REPE CMPSB vergleicht zwei Strings so lange, wie deren Inhalt gleich ist, maximal jedoch CX-mal. Hat also nach der Wiederholung CX den Inhalt 0, so sind beide Strings identisch, andernfalls zeigt CX (bzw. das Subtraktionsergebnis von CX vom Startwert!) an, an welcher Stelle die Strings sich zum erstenmal unterscheiden. REPNE CMPSW wiederholt dagegen den Vergleich nur so lange, bis eine Übereinstimmung gefunden wird.

2.9 Korrekturoperationen

Korrekturoperationen werden nur in wenigen, sehr speziellen Situationen benötigt. Man wird sie sehr selten benutzen, aber wenn sie notwendig werden, sind die Befehle, die der 8086 kennt, mächtige Werkzeuge.

Zum Einsatz kommen die Korrekturbefehle immer dann, wenn der Prozessor mit sogenannten BCDs (*Binary Coded Decimals*) umgehen soll. Diese Zahlen sind anders codiert als die Werte, mit denen der Prozessor üblicherweise rechnet, so daß bei ihrer Manipulation bestimmte Korrekturen notwendig werden.

Im einzelnen sind dies:

- Korrektur nach einer Addition von zwei BCDs (AAA, DAA)
- Korrektur nach einer Subtraktion zweier BCDs (AAS, DAS)

- Korrektur vor einer Multiplikation zweier BCDs (AAM)
- Korrektur nach einer Division zweier BCDs (AAD)

Wozu solche Befehle? Kann denn der Prozessor nicht mit BCDs rechnen? Er kennt doch diesen Datentyp. Was also muß noch korrigiert werden? Was sind BCDs überhaupt?

Nun – BCDs sind, ich habe es an anderer Stelle einmal so formuliert, Krücken[2]! Sie sollen uns glaubhaft machen, daß ein Prozessor auch mit dem von uns so geschätzten dezimalen Zahlensystem zurechtkommt. Leider tut er das eben nicht.

Ein Beispiel: Addieren Sie einmal – binär natürlich, alles andere kann der Prozessor ja nicht – 9 und 5. Was erhalten wir?

9	1	0	0	1
5	0	1	0	1
?	1	1	1	0

Die Bitfolge signalisiert uns das Hexadezimalzeichen $E, was für die dezimale Zahl 14 steht. Soweit ist also alles korrekt! Aber – BCDs sollen ja Dezimalzahlen darstellen, so daß die Addition der Ziffer 9 und der Ziffer 5 die Ziffer 4 sowie einen Überlauf in die nächste Position ergeben müßte! Daher sollte eigentlich die Bitfolge 0100 sowie, wo auch immer, ein Überlauf resultieren.

Übertragen wir das Problem nun auf eine zweizifferige Zahl und berücksichtigen den Überlauf, so müßte das (für uns korrekte) Ergebnis also so aussehen:

09	0	0	0	0	1	0	0	1
05	0	0	0	0	0	1	0	1
14	0	0	0	1	0	1	0	0

Sie müssen zugeben, daß dieses Ergebnis sehr weit weg ist von dem, was wir nach einer einfachen Addition von 9 + 5 erhalten! Was ist also zu tun? Nun – 9 ist offensichtlich die größte Ziffer im Dezimalsystem, $F die größte Ziffer im Hexadezimalsystem.

---

[2] Ganz gerecht ist diese Bemerkung allerdings nicht! Denn es gibt durchaus Fälle, in denen die Benutzung von BCDs nicht nur sinnvoll, sondern sogar erforderlich ist, wie wir auf den nächsten Seiten noch sehen werden. Beispielsweise kann man bei der Verwendung von BCDs und einer speziellen BCD-Arithmetik zu sehr viel genaueren Berechnungen kommen, als dies binär möglich ist – der diversen Rundungsfehler wegen! Daher arbeiten die meisten Taschenrechner im kaufmännischen und wissenschaftlichen Bereich mit BCDs und deren Arithmetik. Aber das sind Spezialfälle!

Das heißt, daß bei Additionen sechs Ziffern entstehen können, die größer als 9 sind und daher korrigiert werden müssen: die Hexadezimalzeichen $A, $B, $C, $D, $E und $F. Doch wie geschieht dies?

Wenn wir nach einer Addition eine Ziffer erhalten, die kleiner oder gleich 9 ist, brauchen wir nichts zu tun – das Ergebnis ist eine korrekte BCD-Ziffer. Andernfalls müssen wir korrigieren: $A entspricht ja »10«, $B »11« usw. Aber Achtung! Diese Zahlen, also »10«, »11«, »12« etc., sind keine dezimalen Zahlen, denn die kennt der Prozessor nicht! Es sind in Wirklichkeit Hexadezimalzahlen, also $10, $11, $12 usw. Das heißt aber, daß es die wirklichen (dezimalen) Werte 16, 17, 18 usw. sind.

ACHTUNG

Die Addition zweier als BCD definierter Ziffern führt also eine mathematisch falsche Berechnung durch. Damit der Prozessor Dezimalzahlen überhaupt darstellen kann, muß z.B. die Addition von 9 und 5 zum mathematisch falschen Ergebnis 20 führen, weil die hexadezimale Darstellung von »20« $14 ist und somit die Ziffernfolge, die wir nach (dezimaler) Addition erwarten!

HINWEIS

BCDs sind artifizielle Ziffern! Sie sind die binäre Darstellung einer Dezimalzahl und heißen deshalb auch so: **b***inary* **c***oded* **d***ecimal*. Berechnungen mit BCDs sind daher nur mittels besonderer Operationen möglich und richtig! Das Ergebnis solcher Berechnungen muß immer als BCD interpretiert werden!

Zurück zum Beispiel! Wenn also die Addition von 5 und 9 zum (mathematisch) falschen Ergebnis 20 führen muß, damit die BCD 14 resultiert, heißt das für unsere Korrekturvorschrift: Falls nach einer Manipulation eine Ziffer größer als 9 herauskommt, ist unverzüglich 6 zu addieren!

AAA

Genau das tut der Befehl AAA. AAA heißt *ASCII Adjust After Addition* und hat unmittelbar im Anschluß an eine Addition zu erfolgen. AAA prüft also, ob die Ziffern alle kleiner oder gleich 9 sind und führt nichts aus, wenn dem so ist. Andernfalls wird zu jeder Ziffer 6 addiert und ein Übertrag in die nächste Ziffer durchgeführt.

Was heißt das aber »in die nächste Ziffer«? Wieso heißt der Befehl eigentlich »*ASCII Adjust After Addition* «? Wir rechnen doch mit BCDs. Wieso dann »*ASCII*«? Ganz einfach: Eine der wenigen sinnvollen Aufgaben für BCDs ist, daß sie sich sehr leicht in ASCII-Ziffern umwandeln lassen. Falls Sie schon einmal in Hochsprachen programmiert haben: Haben Sie sich eigentlich einmal gefragt, was Anweisungen wie *WriteLn* oder *Printf* leisten müssen, wenn sie Ziffern darstellen sollen?

Darstellbar, egal ob auf dem Bildschirm oder über den Drucker, sind nämlich nur die darstellbaren (druckbaren) Zeichen. Diese bilden den so-

genannten Zeichensatz. Es gibt standardisierte Zeichensätze, einer davon ist *ASCII* (meistens »aszi« ausgesprochen – das zweite »i« wird verschluckt!), was für *American Standard Code for Information Interchange*, also amerikanischer Standardcode für den Informationsaustausch, steht.

Nun lassen sich nicht alle Zahlen dieser Welt in einen Zeichensatz aufnehmen – es gibt ja unendlich viele, und das ist ein bißchen zuviel. Also nahm man nur die Ziffern in diesen Zeichensatz auf und vereinbarte, daß Zahlen als Folge solcher Ziffern darzustellen sind. Genau hier beginnt die Leistung solcher Ausgabebefehle wie WriteLn. Ihnen kommt nämlich die Aufgabe zu, im Falle der Darstellung einer Zahl diese in Ziffern zu zerlegen, sie in ASCII-Zeichen umzuwandeln und diese dann auszugeben.

Dies also ist zum einen eine Berechtigung der BCDs, zum anderen der Grund für den Namen der Operation. Addiert man nämlich zu der BCD-Ziffer den Wert 48, so hat man den ASCII-Code dieser Ziffer: 48 für 0, 57 für 9 – und weil es eben im Dezimalsystem keine hexadezimalen Zeichen gibt, muß ein hexadezimales Zeichen in eine BCD umgewandelt werden, bevor es darstellbar wird.

Weil AAA genau das tut, heißt es auch so. Schließlich ist es dem Befehl ganz egal, ob vor seiner Ausführung tatsächlich eine Addition erfolgte oder nicht! Natürlich können wir auch lediglich ein Hexadezimalzeichen in das Register schreiben und dann AAA aufrufen. Das Ergebnis ist das gleiche wie nach einer erfolgten Addition, bei der ein Hexadezimalzeichen erzeugt wurde.

Das Ganze hat aber Konsequenzen. Denn es bedeutet, daß in einem Byte nur noch die Ziffern 0 bis 9 darstellbar sind! AAA dient ja (unter anderem) der Vorbereitung einer Ziffer zur ASCII-Darstellung. Da ASCII-Zeichen Bytes sind, ist also jede Ziffer im ASCII-Code ein Byte. Es läßt sich somit in jedem Byte-Register nur eine Ziffer darstellen, im Wortregister nur zwei. Dies ist auch der Grund, warum BCDs nicht sonderlich effektiv mit dem Speicherplatz umgehen. Anstelle des maximalen Bereichs von 0 – 255 bzw. 0 – 65535 für Bytes bzw. Worte kann unter ASCII nur noch ein Bereich von 0 bis 9 bzw. 0 – 99 verwendet werden.

Dies zeigt uns auch gleich, wohin ein Überlauf stattfinden muß. Nachdem in AL nur eine Ziffer stehen kann, muß ein eventueller Überlauf in AH erfolgen, in dem ja eine zweite Ziffer Platz hat. Genau das passiert. Halten wir daher noch einmal kurz fest, wie AAA tatsächlich arbeitet:

- AAA betrachtet zunächst nur das AL-Register. Der Befehl geht davon aus, daß er nach einer Addition zweier BCD-Ziffern aufgerufen wird, etwa nach *ADD AL, BL*. Daher kann in AL nur eine

Ziffer stehen, die entweder ein korrektes Dezimalzeichen mit dem Wert von 0 bis 9 ist oder eben eine Ziffer, die korrigiert werden muß.

- Muß die Ziffer korrigiert werden, so findet ein Überlauf statt. Dazu wird zunächst 6 zu der in AL stehenden Ziffer addiert und dann die oberen vier Bits in AL definitiv gelöscht, da ja der maximal mögliche Wert von 9 sich mit den unteren vier Bits darstellen läßt. Abschließend werden das Carry- und das Auxiliary-Flag gesetzt, und AH wird um 1 inkrementiert. Findet dagegen kein Überlauf statt, so werden beide Flags gelöscht!

HINWEIS

In diesem Fall gibt es also endlich eine schlüssige Erklärung für das Auxiliary-Flag. Es wird hier gesetzt, wenn ein sogenannter »Dezimalüberlauf« stattfindet, also eine Korrektur eines Hexadezimal- in ein Dezimalzeichen. Aber hieraus ergibt sich auch, daß AAA zwar das AH-Register um 1 inkrementiert, daß aber die eigentliche Korrektur nur im AL-Register stattfindet. Gegebenenfalls muß eben über einen (zeitweisen) Austausch des AL- und des AH-Registers und die erneute Nutzung von AAA auch AH korrigiert werden.

ACHTUNG

Daraus ergibt sich auch, daß unerwartete Effekte eintreten können, falls AAA nicht nach einer Addition zweier BCD-Zahlen aufgerufen wird! Da ja AAA nicht »wissen« kann, ob tatsächlich eine Addition, vor allem auch mit BCD-Ziffern, voranging, arbeitet es in der beschriebenen Weise auch dann, wenn man den Befehl auf andere Werte in AL »losläßt«, und berücksichtigt auch nicht, daß in AH eventuell etwas steht, das mit BCDs nicht konform ist. Verwenden Sie daher alle Korrekturbefehle nur dann, wenn Sie sicher sind, daß mit BCDs manipuliert wird.

AAS

Ein ähnlicher Fall liegt natürlich auch nach einer Subtraktion vor, allerdings im wahrsten Sinne des Wortes mit umgekehrtem Vorzeichen. Daher gibt es auch für diesen Fall einen Korrekturbefehl: AAS, *ASCII Adjust After Subtraction*:

- Der Befehl überprüft, ob die Ziffer in AL im Bereich 0 bis 9 liegt. Wenn ja, werden das Carry- und das Auxiliary-Flag gelöscht.
- Muß korrigiert werden, so wird von der Ziffer in AL 6 abgezogen. Wieder werden die oberen vier Bits in AL definitiv gelöscht, da auch hier das maximal mögliche Ergebnis mit den Bits 0 bis 3 darstellbar ist, die Bits 4 bis 7 also immer 0 sein müssen. Das Carry- und das Auxiliary-Flag werden gesetzt, und AH wird um 1 dekrementiert, ohne selbst korrigiert zu werden.

AAM

Auch bei Multiplikationen und Divisionen muß korrigiert werden. Ganz analog gilt daher alles eben Gesagte auch für AAM, *ASCII Adjust After Multiplication*. Allerdings liegt hier der Fall ein wenig komplizierter, wie Sie wohl vermuten werden. Denn wenn zwei BCD-Zahlen multipliziert

werden, muß dazu der MUL-Befehl verwendet werden. (Da BCDs per definitionem immer positiv sind, darf IMUL nicht verwendet werden.) Das bedeutet aber, da in einem 8-Bit-Register wie AL nur eine Ziffer steht, daß das Ergebnis der Multiplikation in AX korrigiert werden muß.

Wenn Sie z.B. 9 mit 9 über *MUL BL* multiplizieren, was die größte mögliche Multiplikation mit BCDs darstellt, so erhalten Sie 81 – hexadezimal in AX, also die hexadezimale Darstellung von 81 in AL und 0 in AX. Ich hoffe, das ist klar – denn woher soll der Prozessor wissen, daß mit dem *MUL BL*-Befehl von eben BCD-Ziffern multipliziert werden? Es könnte sich ja durchaus auch um die Multiplikation der beiden Hexadezimalzahlen 9 und 9 handeln!

Daß das Ergebnis als BCD aufgefaßt werden muß, weiß nur der Programmierer.Z Er trägt diesem Umstand durch die folgende Korrektur Rechnung. Die 81 muß durch AAM in eine Ziffer 8 in AH und eine Ziffer 1 in AL zerlegt werden. Das heißt aber, daß AAM folgende Operationen ausführt:

- Division des Inhalts von AL durch 10. Dies ergibt (da intern ein DIV-Befehl verwendet wird, der das Ergebnis nur in andere Zielregister ablegt) ein Ergebnis von 8 und einen Rest von 1.
- Ablegen des Ergebnisses der Division in AH als erste Ziffer, des Restes in AL als zweite Ziffer.

TIP

In Pascal-Schreibweise macht AAM also folgendes:

```
Var AH,
    AL : Byte
Begin
  AH := AL DIV 10
  AL := AL MOD 10
End;
```

Im zweiten Teil dieses Buches werden wir genau dieses Verhalten von AAM benutzen, um einige trickreiche Utilities zu programmieren.

AAD

Fehlt noch die Korrektur bei Divisionen. Die funktioniert nun ganz anders! Warum? Dividiert (weil BCDs immer positiv sein müssen!) wird mit DIV. Dieser Befehl erwartet einen Dividenden in DX:AX bzw. AX und erzeugt ein Ergebnis in AX/AL und einen Rest in DX/AH. Gehen wir, wie bei allen anderen Korrekturbefehlen auch, davon aus, daß nur maximal zwei BCD-Ziffern Verwendung finden, so läßt sich das Ganze mit einer Division eines Wortes durch ein Byte erledigen. Das heißt, die BCD-Zahl (aus maximal zwei Ziffern) steht in AX, dividiert wird durch ein Byte, das Ergebnis wird in AL und der Rest in AH dargestellt.

Das aber setzt voraus, daß die BCD-Zahl hexadezimal in AX stehen muß, bevor die Division ausgeführt wird! Daher muß in diesem Fall auch die Korrektur vor der Operation erfolgen! Sie muß zwei BCD-Ziffern, die in AH und AL stehen, in eine Hexadezimalzahl in AX umwandeln. Genau das leistet AAD, *ASCII Adjust before Division*.

TIP

AAD macht daher genau das Gegenteil vom AAM: Es multipliziert den Wert in AH mit 10 und addiert AL dazu. Das Ergebnis wird in AL abgelegt, da es niemals größer als 99 werden kann. AH wird somit auf 0 gesetzt. In AX steht deshalb nach AAD ein Byte mit dem maximalen Wert 99, hexadezimal codiert.

In Pascal-Schreibweise sieht das wie folgt aus:

```
Var AX : Word;
    AH,
    AL : Byte;
Begin
  AX := 10 · AH + AL;
End;
```

Auch diese Beziehung werden wir im zweiten Teil des Buches für einige Tricks benutzen.

DAA, DAS

DAA und DAS schließlich sind zwei Operationen, die vollständig analog zu AAA und AAS ablaufen – mit einer Ausnahme! AAA und AAS sind darauf ausgelegt, mit BCDs zu arbeiten, die pro Byte nur eine BCD-Ziffer codieren. Diese BCDs nennt man *ungepackte* BCDs. In allen Byte-Registern und -Speicherstellen steht bei diesen BCDs nur eine Ziffer – mit einem Betrag von maximal 9. Sie müssen zugeben, daß dies nicht sehr effizient ist!

Bytes können ja Werte bis zu 255 annehmen und eigentlich schon aus diesem Grunde zwei BCD-Ziffern aufnehmen. Deren maximaler Betrag wäre ja »nur« 99. Dies führte dazu, daß man neben *ungepackten* BCDs auch *gepackte* BCDs kennt, nämlich solche BCDs, bei denen ein Byte zwei BCD-Ziffern codiert.

Nun dürfte aber klar sein, daß bei Berechnungen mit solchen *gepackten* BCDs die weiter oben beschriebenen Korrekturoperationen AAA und AAS überfordert sind – sie berücksichtigen ja jeweils nur eine BCD-Ziffer. Dies ist schließlich die Existenzberechtigung für DAA und DAS.

2.10 Sonstige Operationen

In dieser Kategorie tummeln sich verschiedene Befehle, die zwar sehr nützlich, zum Teil sogar äußerst wichtig sind, die sich jedoch nicht so ohne weiteres in einen der oben besprochenen Abschnitte einteilen lassen. Hier befinden sich:

- Befehle zum Aufruf von Unterprogrammen (CALL, INT, INTO)
- Befehle zur Rückkehr aus Unterprogrammen (RET, IRET)
- Befehle zum Umwandeln von Integerwerten (CBW, CWD)
- ein Befehl zum Ausruhen (NOP)
- Befehle zur Synchronisation mit anderen Prozessoren oder Komponenten (HLT, WAIT, LOCK)

Die Befehle CALL, INT, INT0, RET und IRET werden wir im zweiten Teil des Buches detailliert beschreiben.

CBW

CBW, *Convert Byte to Word*, wandelt ein Byte in ein Wort um. In Hochsprachen kennt man diesen Vorgang als *Type-Casting*, also als Typumwandlung. Nun werden Sie fragen, was hierbei so entsetzlich kompliziert ist, daß dafür ein eigener Befehl notwendig wird. Immerhin ist das obere Byte eines Wortes immer 0, wenn nur ein Bytewert repräsentiert werden muß. Man könnte doch ganz einfach z.B. durch *MOV AH, 0* das Byte in AL in ein Wort in AX verwandeln.

Bei vorzeichenlosen Zahlen ist dies richtig, bei positiven, vorzeichenbehafteten Zahlen auch. Aber es gibt ja auch negative Zahlen. Diese werden im Zweierkomplement dargestellt – und dann ist das nicht mehr so einfach.

CBW nimmt uns die Arbeit ab, die z.B. in Form von Fallunterscheidungen positiv/negativ notwendig würden. Falls die in AL stehende Zahl positiv ist, so wird AH mit 0 belegt, falls negativ, so erhält AH den Wert $FF. CWB arbeitet ohne Operanden nur mit dem AL/AX-Register und verändert keine Flags!

CWD

Das Ganze funktioniert auch mit Worten, die in Doppelworte umgewandelt werden sollen. Hierzu dient CWD, *Convert Word to Wouble Word*. Es sollte klar sein, daß nach dieser Anweisung in DX 0 steht, wenn das Wort in AX positiv war, andernfalls $FFFF.

NOP

NOP ist ein ganz toller Befehl: *No Operation*. Er veranlaßt den Prozessor, nichts zu tun! Hin und wieder, zugegebenermaßen recht selten, ist es notwendig, den Prozessor eine Weile lang gar nichts tun zu lassen. Aber das sind Spezialfälle, auf die wir dann ausgiebig eingehen werden, wenn wir NOP benötigen.

NOP wirkt nur für »einen Takt«, das heißt, daß die Aktivität des Prozessors nicht eingestellt wird. Stellen Sie sich einfach vor, daß NOP ein Befehl ist, der nichts bewirkt, aber abgearbeitet werden kann. Nach Ausführung dieses Befehls ist der nächste Befehl an der Reihe.

Es folgen noch drei Befehle, die ebenfalls die Aktivität des Prozessors beeinflussen, wenn auch etwas drastischer:

HLT

HLT oder *Halt* führt den Prozessor in den sogenannten Halt-Zustand. In diesem Zustand führt der Prozessor keinerlei Befehle mehr aus, er tut nichts! Im Unterschied zu NOP tut er tatsächlich gar nichts mehr, sondern wartet.

Doch worauf? Wenn keine Befehle mehr abgearbeitet werden, kann doch nichts mehr passieren. Es gibt allerdings noch die Interrupts, genauer die Hardware-Interrupts (wir klären den Begriff und die Unterschiede an entsprechender Stelle im zweiten Teil des Buches). Tritt ein solcher Interrupt, ein MNI oder ein Reset auf, so erwacht der Prozessor und macht an dem Punkt weiter, der auf den HLT-Befehl folgt.

WAIT

WAIT ist ebenfalls ein Befehl, der eine Aktionslosigkeit des Prozessors bewirkt. Auch nach diesem Befehl wartet der Prozessor. Allerdings nicht auf einen Interrupt oder NMI, sondern darauf, daß sein Coprozessor ihm signalisiert, daß er mit der Aktivität fortfahren kann. Aber das wird im nächsten Kapitel erörtert, das sich mit dem Coprozessor auseinandersetzt.

LOCK

LOCK ist ein Befehl, der das sogenannte LOCK-Signal für die Dauer des nächsten Befehls setzt. Das heißt, daß LOCK eigentlich ein Vorsatz für die möglichen Befehle ist und nur in Kombination mit anderen Befehlen sinnvoll ist und wirkt.

Das LOCK-Signal verhindert, daß andere Komponenten auf den Speicherbereich zugreifen können, solange es gesetzt ist. Interessant ist dieser Befehlsvorsatz also eigentlich nur, falls es außer dem Prozessor noch andere Komponenten gibt, die auf Speicher zugreifen können. Mit LOCK kann sich der Prozessor somit das alleinige Zugriffsrecht auf den Speicher während des nächsten Befehls sichern.

Sinn macht dieser Befehl also nur in sogenannten Multiprozessorumgebungen, in denen mehrere Prozessoren Verwendung finden. Diese Voraussetzung ist in der Regel selten gegeben, läßt man den Coprozessor einmal außer Betracht (dort funktioniert die Synchronisation, wie wir gleich sehen werden, recht gut, so daß bei diesem Gespann kein LOCK erforderlich wird).

Sie kennen nun alle Befehle, die den 8086 veranlassen können, etwas Sinnvolles zu tun. Sie haben auch etwas über die Prozessorarchitektur

gehört und kennen seine Register und den Adreßraum des 8086. Wenden wir uns daher, bevor wir mit der Programmierung beginnen, einem sehr wichtigen Thema zu: dem Coprozessor.

3 Die Register des 8087

Noch vor einigen Jahren waren Coprozessoren sehr teuer und für sehr viele Computeranwender unerschwinglich. Ferner existierten nur sehr wenige Programme, die explizit einen solchen Coprozessor voraussetzten, ja ihn überhaupt nutzen konnten.

So griff weder das Betriebssystem DOS noch so hochkarätige Standardsoftware wie WORD, dBASE oder PAINTBRUSH auf ihn zurück. Selbst Tabellenkalkulationen wie LOTUS oder EXCEL, in denen mathematische Berechnungen durchgeführt wurden, nutzen ihn nicht. Auch die Programmiersprachen entdeckten erst relativ spät, daß es ihn gibt. Der Hintergrund war, daß diese Programme eben auch auf der überwiegenden Anzahl coprozessorloser Rechner laufen sollten.

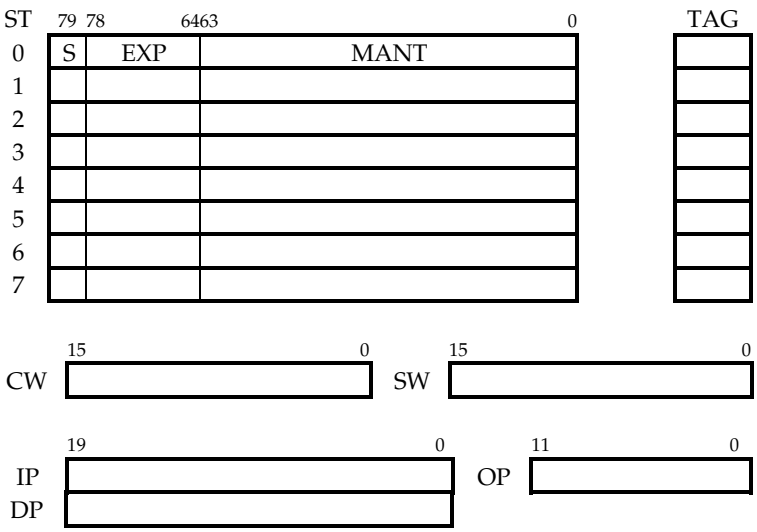
Coprozessoren fristeten ihr Dasein daher praktisch nur in Rechnern, die zu ganz bestimmten Zwecken dienten. Diese benutzten Spezialsoftware, die ausdrücklich auf die Nutzung von Coprozessoren ausgelegt war – ohne solche Mathe-Knechte lief gar nichts. Die Anwendungen dieser Programme, von denen wohl ein bekanntes Beispiel AUTOCAD sein dürfte, ließen sich auf Bereiche einschränken, bei denen die Kosten für einen Coprozessor auch nicht mehr ins Gewicht fielen – z.B. verglichen mit dem Preis der Software.

Doch das hat sich geändert. Alle Preise rutschten ins Bodenlose, so auch die Preise für Coprozessoren. Wenn man heute einen 80387 anschaffen möchte, so ist dieser billiger als manches Spielprogramm. Auch nutzt die moderne Software von heute recht ausgiebig den Coprozessor. Schlimmer noch: Viele der heutigen Programme setzen seine Existenz einfach voraus, weshalb manchmal seine Emulation notwendig wird. Es ist also an der Zeit, sich mit diesem Teil des Rechners auseinanderzusetzen!

Bis heute gibt es sehr wenige gute Bücher, die den Coprozessor erklären. Auch wenn Sie z.B. einmal in die Handbücher von Turbo Pascal, Visual C++, Turbo C++, MASM oder TASM schauen, werden Sie sich wundern! So werden Sie vor allem im Handbuch zu letzterem zwar jede Menge Informationen zum 8086, 80286 und 80386 finden. Auch Tips und Tricks und Programmieranweisungen finden Sie dort zuhauf. Aber der 8087?

Beim *Turbo Assembler* wird lediglich auf 20 Seiten des 374 Seiten starken Referenzhandbuchs der Coprozessor beschrieben – d.h. es gibt keine ausreichende Darstellung, Beispiele fehlen. Im 496 Seiten umfassenden Benutzerhandbuch finden sich auf zwei (!) Seiten die Ergänzungen, die der 80287 und der 80387 gegenüber dem 8087 erfahren haben. Das war's. Microsoft ist mit seinem Handbuch zu MASM auch nicht ausführlicher.

Aber auch andere Bücher scheinen den Coprozessor zu verschmähen. Dabei ist seine Programmierung auch nicht schwerer als die Programmierung des 8086, und die Software von heute kommt, wie gesagt, nicht um ihn herum. Packen wir das heiße Eisen also an. Wir werden dazu analog zum 8086 vorgehen und zunächst die Register und dann die Befehle des 8087 besprechen. Abgeschlossen wird das Kapitel mit der Beschreibung, wie 8086 und 8087 zusammenarbeiten.



Der Coprozessor besitzt acht Rechenregister, die als sogenannter Stack oder Stapel angeordnet sind (verwechseln Sie diesen Stack bitte nicht mit dem des Prozessors! Das ist etwas ganz anderes). Jedes dieser Register ist 80 (!) Bit breit. Bit 79 eines solchen Registers codiert das Vorzeichen des Wertes, Bit 63 bis 0 die Mantisse der Realzahl und Bit 78 bis 64 den Exponenten (mit Vorzeichen). Ferner verfügt jedes Register über ein zwei Bit breites *Tag-Feld*, in dem die Gültigkeit dessen verzeichnet ist, was im korrespondierenden Register eingetragen ist. Dieses *Tag-Feld* kann folgende Werte annehmen:

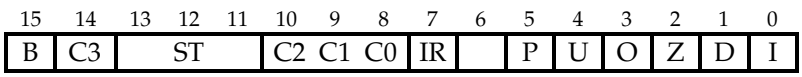
- 00 (=0): gültiger Wert
- 01 (=1): Null

- 10 (=2): spezieller Wert, wie z.B. *NaN*, unendlich, denormalisiert
- 11 (=3): leeres Register

Bitte gedulden Sie sich noch bis zur Besprechung der Operationen, um zu erfahren, was es mit den Begriffen *NaN* und denormalisiert auf sich hat. Zunächst soll genügen, daß das *Tag-Feld* entscheidend dabei mitwirkt, festzustellen, was im entsprechenden Stackregister steht.

Um ganz ehrlich zu sein: die *Tag-Felder* gibt es eigentlich gar nicht. Genauer gesagt: es gibt sie nicht so, wie eben dargestellt. Es gibt vielmehr ein *Tag-Register*, in dem die *Tag-Felder* zusammengefaßt sind. Dennoch ist die Funktion die gleiche wie beschrieben. Ob die *Tag-Felder* nun rein physikalisch etwas anders organisiert sind oder nicht, spielt schließlich für die Wirkungsweise keine Rolle.

Wie Sie in der Abbildung sehen, besitzt auch der Coprozessor etwas Ähnliches wie ein Flagregister. Es heißt hier allerdings Statuswortregister (*Statuswortregister)* und ist folgendermaßen belegt:



Bit 15 signalisiert, daß der Coprozessor gerade eine Berechnung durchführt, also *busy* ist. Es wird jedoch, wie wir später noch sehen werden, selten vorkommen, daß dieses Bit von Bedeutung ist. Wichtiger sind Bit 14 sowie die Bits 10 bis 8, die zusammen den sogenannten *Condition Code* bilden. Das ist ein künstlicher Wert, den man nicht irgendwie auslesen kann; man muß ihn aus der Flagstellung von C3 bis C0 selbst berechnen. Dies erfolgt normalerweise, wie wir im zweiten Teil des Buches noch sehen werden, nicht! Dennoch ist es zum Verständnis nicht ungeschickt, ihn im Rahmen des Buches zu berechnen und darzustellen, da einige Debugger, wie z.B. TD es tun! Man kann auf diese Weise etwas besser dem Geschehen folgen.

Der *Condition Code* erfüllt beim 8087 ähnliche Funktionen wie die Flags beim 8086. Allerdings sind die Zusammenhänge hier etwas komplexer: Der Wert wird in unterschiedlichen Situationen unterschiedlich interpretiert.

Beginnen wir bei Vergleichen. Wie wir bei der Besprechung des 8086 schon gesehen haben, ist es interessant, festzustellen, ob zwei Zahlen größer, gleich oder kleiner sind. Der 8086 signalisiert durch Verändern des Carry-Flags bei vorzeichenlosen Zahlen, des Overflow-Flags bei vorzeichenbehafteten Zahlen und des Auxiliary-Flags bei BCDs, ob der zweite Operand größer oder kleiner als der erste war. Waren beide gleich groß, so setzt der 8086 das Zero-Flag.

Ähnlich arbeitet auch der 8087. Doch weil dieser grundsätzlich nur mit vorzeichenbehafteten Zahlen arbeitet, entfällt die Notwendigkeit für ein Carry-Flag. Auch die BCDs behandelt der 8087 nicht grundsätzlich anders als vorzeichenbehaftete Zahlen, weshalb auch kein Auxiliary-Flag erforderlich ist. Bleiben also das Overflow- und das Zero-Flag. Diese beiden Flags werden bei Vergleichen vom *Condition Code* nachgebildet: C0 ist das Overflow-Flag des 8087, C3 das Zero-Flag. C1 und C2 haben bei Vergleichen keine Bedeutung, werden also nicht verändert. Eine etwas andere Bedeutung hat der Condition Code bei Operationen, die einen Registerinhalt untersuchen. Wir kommen jedoch bei den entsprechenden Operationen darauf zurück.

TIP

Halten wir zunächst fest, daß der Condition Code aufgrund der eben geschilderten Zusammenhänge praktisch ein kleines Flagregister des 8087 ist, bei dem lediglich ein Zero-Flag und ein Overflow-Flag existeren. Das heißt aber, daß der Condition Code (und somit die beiden Flags) nur von Operationen verändert wird, die auch beim 8086 Veränderungen an Flags hervorrufen würden, also z.B. von Vergleichen und arithmetischen Operationen. Logische Operationen gibt es beim 8087 definitionsgemäß nicht – wenn man es genauer bedenkt, ist das sogar ganz richtig so.

Die Bits 13 bis 11 beinhalten die Nummer des derzeit aktuellen »obersten« Registers des Stapels, des sogenannten TOS (*Top Of Stack*). Gültige Werte sind hier 0 bis 7, also die Nummern der tatsächlich verfügbaren physikalischen Register. Warum dieser TOS gespeichert werden muß, sehen wir weiter unten.

Bit 7 ist das *Interrupt-Request-Flag*. Es signalisiert, daß ein Interrupt des Prozessors (nicht des Coprozessors!) notwendig wird, da eine Ausnahmebedingung vorliegt. Diese Ausnahmebedingungen werden in den folgenden Bits genauer spezifiziert:

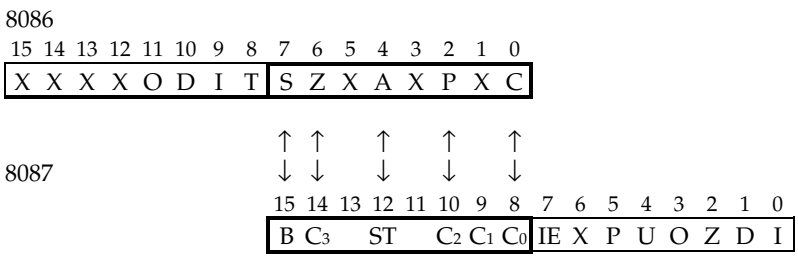
- *Precision* (Bit 5); die Genauigkeit der Berechnung ist unzureichend
- *Underflow* (Bit 4); der minimal darstellbare Wert wurde durch die vorangehende Operation unterschritten
- *Overflow* (Bit 3); der maximal darstellbare Wert wurde durch die vorangehende Operation überschritten
- *Zero Divide* (Bit 2); es wurde versucht, durch 0 zu dividieren
- *Denormalized Operand* (Bit 1); es wurde versucht, mit einer denormalisierten Zahl zu manipulieren
- *Invalid Operation* (Bit 0); es wurde eine ungültige Operation ausgeführt

TIP

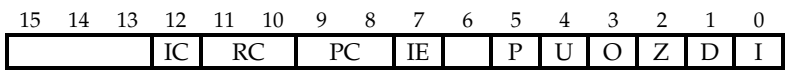
Obwohl wir hiermit schon ein wenig vorgreifen, sollte vielleicht doch ein kleiner Einschub gemacht werden. Ich glaube, er dient dem Verständnis und hilft ein wenig. Erinnern Sie sich an die Flags beim 8086.

Aufgrund bestimmter gesetzter oder gelöschter Flags war mit Hilfe der bedingten Sprünge ein Reagieren auf verschiedene Bedingungen möglich. Um es gleich vorwegzunehmen: Dies ist mit den Flags des 8087 nicht möglich! Es gibt keinen Befehl, mit dem der 8086 auf gesetzte oder gelöschte 8087-Flags reagieren kann.

Es gibt jedoch eine trickreiche Möglichkeit, dies dennoch zu erreichen! Wir wollen erst weiter unten klären, wie dies realisiert werden kann. Hier wollen wir nur versuchen, die Flags des 8087 denen des 8086 gegenüberzustellen. Vergleichen wir also einmal beide Register:



Falls es uns gelingen sollte, die Bits 15 bis 7 aus dem Statuswort des 8087 irgendwie in die Bits 7 bis 0 des Flagregisters des 8086 zu kopieren, so können wir den Condition Code des 8087 wenigstens zum Teil über die Flags des 8086 und somit über bedingte Sprunganweisungen auswerten. Die Zuordnung wäre dann: C3 ≡ Zero-Flag, C2 ≡ Parity-Flag und C0 ≡ Carry-Flag. C2 läßt sich leider nicht testen, da es keinem definierten Flag des 8086 entspricht. Der *Stack-Pointer* ST ist uninteressant, das Busy-Flag B, das über das *Sign-Flag* getestet werden kann, eigentlich auch. Neben dem *Statuswortregister* gibt es auch ein Kontrollwortregister (*Control Word Register)*:



Die niedrigen 8 Bits haben viel Ähnlichkeit mit denen aus dem *Statuswort*. Der Verdacht liegt nahe, daß sie irgend etwas miteinander zu tun haben. So ist es auch. Im Kontrollwortregister kann durch die Bits 0 bis 7 festgelegt werden, ob und wenn ja, bei welcher Ausnahmebedingung, die durch das *Statuswort* definiert wird, ein Interrupt ausgelöst werden soll. Bit 7 legt fest, daß dies überhaupt zu geschehen hat, die Bits 5 bis 0 regeln die Ausnahme.

Wichtig sind noch die Bits 8 und 9, mit denen die »Arbeitsgenauigkeit« des Coprozessors festgelegt wird, und Bit 10 und 11, in denen die Rundungsarten bei Berechnungen gewählt werden können.

Für die *Precision Control* gilt hierbei:

- 00 (=0): 24 Bits Genauigkeit
- 01 (=1): reserviert
- 10 (=2): 53 Bits Genauigkeit
- 11 (=3): 64 Bits

Gemeint ist in allen Fällen die Genauigkeit der Mantisse. *Round Control* läßt mit

- 00 (=0) eine Rundung zur nächsten oder geraden Zahl zu,
- 01 (=1) eine Rundung in Richtung »Minus unendlich«,
- 10 (=2) eine Rundung zu »Plus unendlich« und
- 11 (=3) schneidet den Nachkommateil vollkommen ab.

Bit 12 schließlich, *Infinity Control*, gibt an, wie Unendlichkeiten gehandhabt werden sollen. Es gibt hierzu zwei Möglichkeiten: das affine und das projektive Modell.

Im affinen Modell, das gewählt wird, wenn Bit 12 gesetzt wird, liegen alle Zahlen auf einem Zahlenstrahl, der von -∞ bis +∞ reicht. Somit sind die beiden Unendlichkeiten -∞ und +∞ rein wertmäßig voneinander verschieden und können als unterschiedliche »Zahlen« behandelt werden. Das projektive Modell dagegen schließt den Zahlenstrahl zu einem Kreis, der sich bei den beiden Unendlichkeiten schließt. Somit liegen -∞ und +∞ auf dem gleichen Kreispunkt und werden nicht unterschieden.

Wir sind mit der Besprechung des Registersatzes des 8087 fast fertig. Es fehlen nur noch drei Register, die aber nur der Vollständigkeit halber erwähnt werden, weil sie für den Programmierer weder interessant noch manipulierbar sind.

IP ist der *Instruction-Pointer*. Er hat die gleiche Funktion wie CS:IP beim Prozessor. Während dort jedoch die maximale 20-Bit-Adresse durch die Segmentierung in zwei 16-Bit-Register geschrieben werden muß, kann der 8087 sie vollständig in ein 20-Bit-Register laden und dort halten.

DP, *Data Pointer*, macht das gleiche, nur für die Daten. Man kann ihn daher mit DS:SI beim Prozessor vergleichen. Fehlt noch OP, ein »Zwischenspeicher« für den abzuarbeitenden Befehl, der wenig interessant ist.

Soweit die Besprechung der Befehle, mit denen der Coprozessor arbeitet. Wichtig ist nun jedoch noch zu erfahren, wie er dies tut.

Die Rechenregister des Coprozessors bilden einen sogenannten Stack, also einen Stapel. Der eine oder andere von Ihnen wird Stacks vielleicht von seinem Taschenrechner her kennen. So besitzen alle Rechner, die mit der

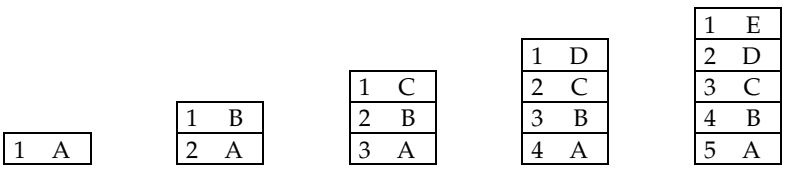
Umgekehrten Polnischen Notation (UPN) arbeiten, einen solchen Rechenstack. Andere wiederum werden an den Stack des 8086 denken.

Im Prinzip ist der Stack also nichts Ungewöhnliches, dennoch haben viele Leute Schwierigkeiten, seine Funktion zu verstehen. Dabei ist das eigentlich gar nicht so schwer! Denken Sie einfach an einen Stapel Kisten.

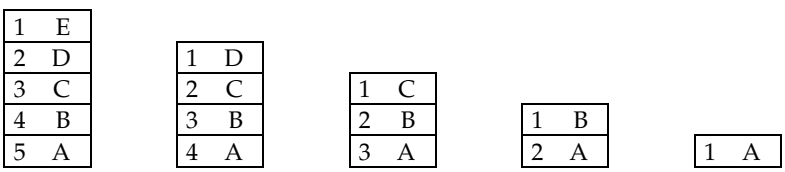
Zunächst füllen Sie eine Kiste mit irgend etwas, das Sie aufbewahren wollen. Sobald die Kiste voll ist, brauchen Sie eine neue. Sie holen sich also eine weitere leere Kiste und, weil Sie wenig Stauraum haben, stapeln Sie diese auf der ersten Kiste, der vollen. Sie machen dann auch die zweite Kiste voll, holen sich eine weitere, leere, füllen diese usw., bis Sie nichts mehr aufzubewahren oder keine Kisten mehr haben.

Natürlich sind Sie bei dieser Aktion nicht ganz planlos vorgegangen! In weiser Voraussicht haben Sie die Dinge, die Sie wahrscheinlich als nächstes wieder benötigen, ganz zum Schluß in Kisten verpackt. Denn unsere Kisten sind sehr groß und schwer, und wir können an den Inhalt einer »weiter unten« liegenden Kiste nur dadurch in einfacher Weise herankommen, daß wir die Kisten darüber leeren und wieder abräumen – oder aufwendig umstapeln!

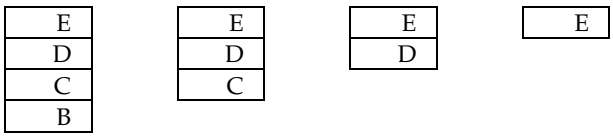
Was wir auf diese Weise erzeugt haben, ist ein Stapel Kisten, in dem wir die Dinge gemäß ihrer Wichtigkeit von oben nach unten geordnet haben. Wir können also nach jedem Schritt die Kisten von oben nach unten durchnumerieren: Oben liegt Kiste 1, unten Kiste X.



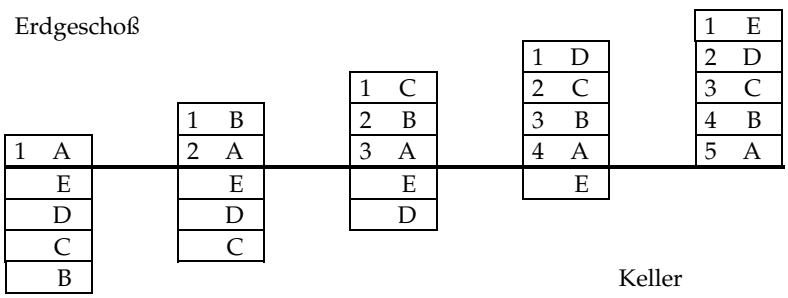
Wir sollten uns dabei merken, daß die Nummer der Kiste immer anzeigt, wieviel Bedeutung wir ihr zumessen. Die jeweils oberste Kiste ist Kiste Nummer 1, also die Kiste mit dem Inhalt, den wir als nächstes brauchen werden, selbst wenn es die eigentlich fünfte Kiste, also Kiste E, ist. Wie schon angedeutet, geht das Ganze auch umgekehrt:



Doch woher beziehen wir nun die Kisten, und wo stecken wir nicht mehr benutzte hin? Aus dem Keller und in den Keller! Dort sind sie in einem Regal gestapelt. Und weil Faulheit im allgemeinen obsiegt, verwenden wir immer diejenige, die uns am nächsten ist, also die jeweils untere im Regal, an die wir leichter herankommen:



Schauen wir uns das Beladen der Kisten nun einmal von außen mit Röntgenaugen an, mit denen wir Keller und Erdgeschoß gleichzeitig durch die Wand betrachten können:



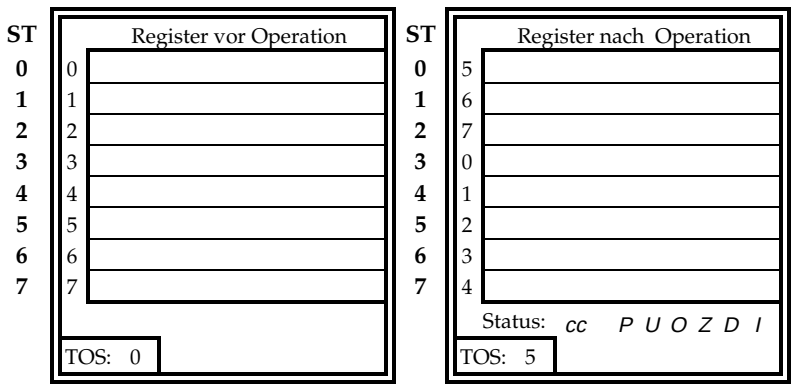
Während im Erdgeschoß der Stapel mit den vollen Kisten wächst, schrumpft die im Kelleregal vorhandene Anzahl freier Kisten in gleichem Maß. Beachten Sie bitte, daß die oberste Kiste immer die Nummer 1 hat, egal welchen Buchstaben die Kiste tatsächlich hat. Ich glaube, wir müssen uns den umgekehrten Weg nicht auch noch anschauen. Jede Kiste, die oben weggeräumt wird, kommt an die oberste freie Stelle im Regal.

4 Der Befehlssatz des 8087

Es ist auch für denjenigen, der sich in der Programmierung des Prozessors auskennt, nicht einfach nachzuvollziehen, was sich in den Registern des 8087 abspielt, wenn seine Befehle abgearbeitet werden. Das Rechnen mit Stacks ist nicht jedermanns Sache, und Intel hat es mit der zwar ungeheuer flexibel einsetzbaren, aber schwer zu durchschauenden Art der Coprozessorprogrammierung nicht gerade erleichtert.

Vielleicht ist bisher auch aus diesem Grund so wenig zum 8087 veröffentlicht worden! Wie dem auch sei, im folgenden wird versucht, anhand eines »Vorher-Nachher«-Schaubildes die Vorgänge bei der Abarbeitung eines Coprozessorbefehls etwas zu verdeutlichen. Sie sehen in diesem Schema die acht Rechenregister, das Statuswort sowie die Nummer des Bezugsregisters vor und nach der Operation eingezeichnet.

Schauen Sie sich noch einmal das letzte Schaubild auf Seite 87 an. Wenn Sie alle acht Register darstellen und nicht unterscheiden, dann erhalten Sie, immer in gleicher Höhe gezeichnet, die folgende identische Anordnung der Register:



Mit ST wird hierbei die Nummer der »Kisten« bezeichnet, mit dem Wert rechts daneben die physikalische (also »echte«) Registernummer.

HINWEIS Lassen Sie sich, wie zum Ende des letzten Kapitels demonstriert, nicht durch die Durchnumerierung der Register bluffen und auch nicht dadurch, daß oben eine Numerierung »von oben nach unten« erfolgt, im letzten Kapitel die Buchstaben »von unten nach oben« vergeben wurden. Namen sind Schall und Rauch! Alle mathematischen Operationen beziehen sich grundsätzlich auf das aktuelle »Bezugsregister«, das auch *Top Of the Stack* oder treffend abgekürzt TOS heißt, also das Register, das immer »oben« ist. Ein Befehl wie z.B. *FCHS*, der zum Wechseln des Vorzeichens einer Zahl dient, wirkt nur auf ein einziges Register: den TOS! Falls dieser TOS das Register 7 ist, so wird das Vorzeichen der Zahl im Register 7 gewechselt. Ist TOS = 4, so wird die Zahl im Register 4 verändert.

In den folgenden Schaubildern wird immer der TOS oben auf dem Stapel gezeigt, ST(1) direkt darunter, ST(2) wiederum darunter usw., bis mit ST(7) der *Bottom Of Stack* (BOS) erreicht ist. Die Stapelnummern liegen also ein für allemal fest und ändern sich nicht, weshalb sie im folgenden auch nur noch bei Bedarf zur Erinnerung angefügt

werden. Die zusätzliche Darstellung der (physikalischen) Registernummern dient lediglich dazu, Ihnen die Arbeitsweise des Coprozessorstacks zu demonstrieren.

HINWEIS

Die von mir gewählte Darstellung wird auch vom Turbo Debugger von Borland verwendet. Sie werden sich dort also sehr schnell zurechtfinden, wenn Sie sich einmal an mein Schema gewöhnt haben.

Das Statuswort wird nur gezeigt, wenn es für die Ergebnisse der Operationen wichtig ist. Es sind die sechs unteren Bits sowie Bit 14 und die Bits 10 bis 8 dieses Wortes, die wir bei der Besprechung der Register des 8087 schon kennengelernt haben. Ein » « als Eintrag bedeutet hierbei, daß dieses Bit von der Operation nicht beeinflußt wird, ein »*«, daß es in Abhängigkeit vom Ergebnis der Operation gesetzt oder gelöscht wird. Wird es zwangsweise gesetzt, so wird hier eine »1« verzeichnet, sollte es immer gelöscht werden, so erscheint hier eine »0«. Der Condition Code wird als Zahl dargestellt.

4.1 Arithmetische Operationen mit Realzahlen

Der 8087 kennt verschiedene Arten von Realzahlen. Je nach geforderter Genauigkeit der Mantisse, also der Anzahl an Nachkommastellen, und der Größe des Exponenten besteht die Möglichkeit, Realzahlen mit 4, 8 und 10 Bytes zu codieren. In Hochsprachen werden diese Möglichkeiten wie folgt genutzt:

Größe	8086	8087	ASM[2]	Pascal[3]	C[4]
32 Bits	-	SHORTREAL	DWORD	single	float
48 Bits	-	-	FWORD	real[1]	-
64 Bits	-	LONGREAL	QWORD	double	double
80 Bits	-	TEMPREAL	TBYTE	extended	long double

1 Der Datentyp *real* in *TurboPascal* und *QuickPascal* ist eine Softwarerealisation, die mit Befehlen des 8086 arbeitet und somit von einem Coprozessor oder einer Coprozessoremulation unabhängig ist.

2 Einige Assembler kennen noch weitere Datentypen, die jedoch nicht zum Standardumfang der Assembler-Daten gehören und daher nur dann benutzt werden sollten, wenn es nicht auf Assembler-Kompatibilität ankommt. So kennt z.B. MASM 6.0 auch die Datentypen REAL4 für DWORD, REAL8 für QWORD und REAL10 für TBYTE.

3 Die verschiedenen Pascal-Dialekte unterscheiden sich hier. So kennt der Microsoft Professional Pascal Compiler 4.0 nicht die Datentypen *single*, *double* und *extended*, wie sie in TurboPascal und QuickPascal bekannt sind. Statt dessen heißt hier *single real* oder *real4* und *double real* oder *real8*. Ein Pendant zu *extended* kennt dieser Compiler nicht.

4 Schön, daß wenigstens C-Dialekte hier nicht aus der Reihe tanzen: Diese Datentypen kennt jeder C-Compiler!

ACHTUNG

Intern arbeitet der 8087 immer mit 80 Bits, also mit Realzahlen vom Typ TEMPREAL. Deshalb erfolgt auch intern eine »Umrechnung« in und aus diesem Datentyp, falls SHORTREAL- oder LONGREAL-Zahlen verwendet werden. Dies geschieht aber im Hintergrund und für uns nicht sichtbar, so daß wir hierauf auch nicht achten müssen.

HINWEIS Dieser Satz könnte uns nun dazu veranlassen, immer mit TEMPREAL-Zahlen zu arbeiten; schließlich macht es kaum einen Unterschied, ob nun 8 oder 10 Bytes für die Darstellung der Zahlen benötigt werden. Das mag auch in den meisten Fällen richtig sein.

Dennoch gibt es Gründe, dies nicht zu tun. So können nur sehr wenige 8087-Operationen TEMPREAL-Zahlen direkt verarbeiten. Es handelt sich hierbei um die Operationen, die Realzahlen in die Stackregister laden oder aus ihnen in den Speicher ablegen.

Alle anderen Operationen dagegen können TEMPREAL-Zahlen nur verarbeiten, wenn sich diese schon im Stack befinden, also mit einem Ladebefehl geholt worden sind. Ebenso kann als Ziel einer TEMPREAL-Operation nur wieder der Stack gewählt werden, aus dem dann gegebenenfalls in den Speicher zurückgeladen wird.

Während also bei Verwendung von LONGREAL-Zahlen ein Konstrukt wie

```
...
fadd      st,LongRealVar1
fsub      LongRealVar2,st
...
```

möglich ist, muß beim Einsatz von TEMPREAL-Zahlen der Umweg über den Stack genommen werden:

```
...
fld       TempRealVar1
fadd      st,st(1)
fld       TempRealVar2
fsub      st,st(1)
fst       TempVar2,st
...
```

TIP Beide Verfahren unterscheiden sich in Wirklichkeit nicht sehr! Bei Verwendung von SHORTREAL- oder LONGREAL-Zahlen kann direkt in/aus Speicherstellen operiert werden. Dafür müssen sie jedoch intern zunächst in TEMPREAL-Werte umgebildet und nach der Operation wieder zurückverwandelt werden, was natürlich Zeit kostet. TEMPREAL-Zahlen dagegen brauchen nicht umgeformt zu werden. Sie müssen dafür jedoch zunächst in den Stack geholt werden, bevor mit ihnen gearbeitet werden kann, was auch zusätzliche Zeit kostet. Welches Verfahren das richtige ist, hängt unter anderem davon ab, ob die 25% mehr an Speicherplatz (10 statt 8 Bytes) kritisch sind oder wieviel Zeit im jeweiligen Fall aufgebracht werden muß. Dies ist unter anderem von Hardware-Komponenten und dem Systemtakt abhängig. Daher kann ein genereller Tip nicht gegeben werden. Hier heißt es: probieren, wenn man wirklich

das Allerletzte herausholen will. Aber meistens lohnt sich das nicht. Benutzen Sie daher immer den Datentyp, der Ihnen am geeignetsten erscheint, um die Zahlen darzustellen.

Doch nun zu den Operationen. Im folgenden werden wir, wie bei der Besprechung des 8086 auch, folgende Bezeichnungen für Operanden wählen:

- *temp*, wenn eine TEMPREAL verarbeitet werden kann. Dies beinhaltet automatisch auch die anderen Real-Typen, also SHORTREAL und LONGREAL.
- *real*, wenn lediglich die Verwendung von SHORTREAL- und LONGREAL-Zahlen möglich ist.

In Beispielen finden Sie Namen für die Variablen, die den Datentypen aus Turbo Pascal entsprechen: *SingleVar* für SHORTREAL, *DoubleVar* für LONGREAL und *ExtendedVar* für TEMPREAL.

Der 8087 kennt folgende Befehle zum Umgang mit Realzahlen:

- FADD, FADDP
- FCOM, FCOMP, FCOMPP
- FDIV, FDIVP, FDIVR, FDIVRP
- FLD
- FMUL, FMULP
- FSUB, FSUBP, FSUBR, FSUBRP
- FST, FSTP

FLD

Betrachten wir zunächst FLD. Mit FLD kann eine Realzahl aus dem Speicher geholt werden. Dies geschieht über die Anweisung

```
fld     Temp
```

wobei *temp* eine Realzahl vom Typ SHORTREAL oder LONGREAL, aber auch TEMPREAL sein kann. Bevor die Realzahl tatsächlich geladen wird, wird der Stack-Pointer dekrementiert, so daß Platz für den neuen Wert geschaffen werden kann. Die Dekrementierung des Stack-Pointers um 1 bedeutet, daß das »unterste« Register »nach oben« kommt und alle vorangehenden um eine Position »nach unten« schiebt (vgl. Abbildung auf der folgenden Seite).

Falls die Aktion erfolgreich verlief, so werden die Flags nicht verändert.

ACHTUNG

Wichtig ist hierbei, daß sich im »untersten« Register, nämlich dem, das nun den zu ladenden Wert aufnehmen soll, nichts befindet, daß dieses Register also als *empty* markiert ist. Nur dann werden die Flags nicht verändert.

	Register vor Operation	
0	+1.000000000	E+0000
1	+2.000000000	E+0000
2	+3.000000000	E+0000
3	+4.000000000	E+0000
4	+5.000000000	E+0000
5	+6.000000000	E+0000
6	+7.000000000	E+0000
7	EMPTY	

TOS: 0

	Register nach Operation	
7	+4.711000000	E+0003
0	+1.000000000	E+0000
1	+2.000000000	E+0000
2	+3.000000000	E+0000
3	+4.000000000	E+0000
4	+5.000000000	E+0000
5	+6.000000000	E+0000
6	+7.000000000	E+0000

Status: cc P U O Z D I

TOS: 7 * *

Befindet sich dagegen irgendein anderer Wert in diesem Register, so wird es nicht etwa mit dem zu ladenden Wert belegt, sondern mit einer Codezahl, die dem Prozessor mitteilt, daß der aktuelle Inhalt nicht stimmt.

	Register vor Operation	
0	+1.000000000	E+0000
1	+2.000000000	E+0000
2	+3.000000000	E+0000
3	+4.000000000	E+0000
4	+5.000000000	E+0000
5	+6.000000000	E+0000
6	+7.000000000	E+0000
7	+8.000000000	E+0000

TOS: 0

	Register nach Operation	
7	-NaN	
0	+1.000000000	E+0000
1	+2.000000000	E+0000
2	+3.000000000	E+0000
3	+4.000000000	E+0000
4	+5.000000000	E+0000
5	+6.000000000	E+0000
6	+7.000000000	E+0000

Status: cc P U O Z D I

TOS: 7 2 1

NaN heißt *Not a Number* und zeigt dem Coprozessor an, daß die enthaltene Zahl den gültigen Wertebereich aus irgendeinem Grund verlassen hat. Wie wir aus dem letzten Kapitel wissen, ist *NaN* eigentlich gar keine Zahl, sondern ein Code im Tag-Feld des Registers, der die Ungültigkeit des Inhalts anzeigt. Belassen wir trotzdem der Einfachheit halber die Formulierung, daß das Register eine *NaN* als Inhalt hat, denn ganz so falsch ist es auch wiederum nicht (siehe Anhang)!

Außerdem wird das *Invalid-Operation-Flag* und der *Condition Code 2* gesetzt, was bedeutet, daß die letzte Operation eine gültige, negative, aber nicht normalisierte Zahl hervorgerufen hat (zu den unterschiedlichen Condition Codes siehe Teil 3).

Aber FLD kann noch mehr. So kann mit diesem Befehl nicht nur ein Wert aus einer bestimmten Speicherstelle ausgelesen werden, sondern auch aus einem anderen Register.

Die Anweisung lautet dann

```
fld     st(i)
```

wobei *i* eine Zahl zwischen 0 und 7 sein kann. 0 bezeichnet den TOS selbst, wodurch ganz einfach der Wert im TOS kopiert wird. Für diesen Fall kann auch kurz

```
fld     st
```

geschrieben werden.

ACHTUNG

Die Zahl *i* bezeichnet nicht die physikalische Nummer des Registers, sondern die Position auf dem Stack! So ist immer der TOS = 0, das direkt darunter liegende Register 1 und das letzte Register 7, egal, welche physikalischen Nummern diese Register haben.

Der Ablauf ist dann der gleiche: Zunächst wird das betreffende Register gelesen und der enthaltene Wert zwischengespeichert. Dann wird der Stack-Pointer dekrementiert, also Platz auf dem TOS geschaffen. An diese Stelle wird der gemerkte Registerinhalt kopiert. Auch für den Fall, daß das zu verwendende Register nicht *empty* ist, ist die Reaktion gleich: Setzen des Invalid-Operation-Flags und des Condition Code 2, wobei *-NaN* in den TOS geschrieben wird.

Im folgenden Beispiel wird mit *FLD ST(5)* der Inhalt aus Position 5 (**nicht Register 5!**) in den TOS kopiert:

ST		Register vor Operation		Register nach Operation
0	1	+1.000000000 E+0000	0	+6.000000000 E+0000
1	2	+2.000000000 E+0000	1	+1.000000000 E+0000
2	3	+3.000000000 E+0000	2	+2.000000000 E+0000
3	4	+4.000000000 E+0000	3	+3.000000000 E+0000
4	5	+5.000000000 E+0000	4	+4.000000000 E+0000
➜	6	+6.000000000 E+0000	5	+5.000000000 E+0000
6	7	+7.000000000 E+0000	6	+6.000000000 E+0000
7	0	EMPTY	7	+7.000000000 E+0000
		TOS: 1		Status: cc P U O Z D I TOS: 0 * *

FST, FSTP

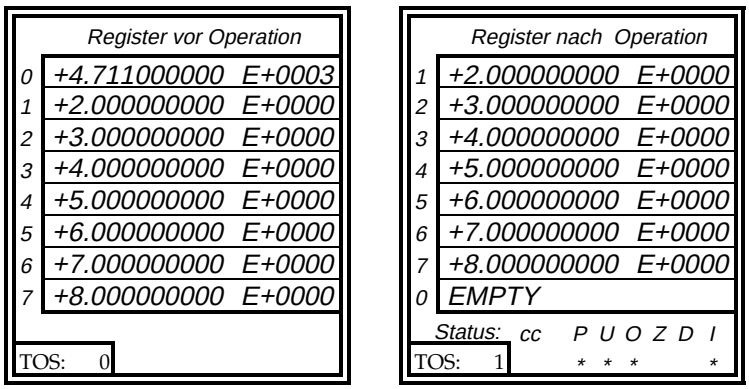
Das Pendant zu FLD ist FST. Mit FST kann der Wert im TOS an eine bestimmte Speicherstelle geschrieben werden.

```
fst     Temp
```

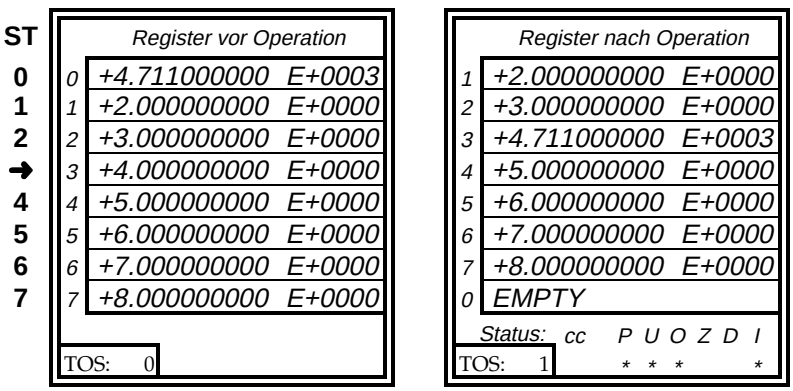
Im Unterschied zu FLD wird der Stack dabei jedoch nicht verändert. Es erfolgt lediglich das Abspeichern einer Kopie des TOS.

TIP

Häufig genug benötigt man den Wert nicht mehr, der mit FST abgespeichert wurde. Vielmehr möchte man entweder einen neuen Wert laden oder mit anderen Registerinhalten weiterrechnen. Um dies zu erreichen, gibt es den Befehl FSTP (das »P« steht für Pop), der ebenso wie FST den Inhalt des TOS in die Speicherstelle kopiert, dann jedoch den Stack-Pointer wieder inkrementiert, so daß der Stack wieder nach oben wandert. Gleichzeitig wird das soeben »gespeicherte« Register »freigegeben«, indem es mit dem »Wert« *empty* belegt wird, so daß sofort mit einem Ladebefehl wie FLD weitergearbeitet werden könnte. Schauen wir uns dies einmal am Beispiel *FSTP ExtendedVar* an:



Natürlich funktionieren FST und FSTP auch mit Registern statt mit Speicherstellen. Durch *FST ST(3)* z.B. wird der Inhalt vom TOS in das Register an Stackposition 3 kopiert, während ein *FSTP ST(3)* anschließend noch den Stack poppt, so daß der kopierte Wert an Position 2 wandert:



Sollte sich im TOS eine *NaN* befinden oder ist der TOS *empty*, so wird natürlich eine fehlerhafte Handlung durchgeführt, was sich in einem gesetzten Invalid-Operation-Flag manifestiert. Nichtsdestotrotz wird die Operation ausgeführt, so daß tatsächlich eine *NaN* abgespeichert und, falls FSTP verwendet wurde, der Stack auch gepoppt wird.

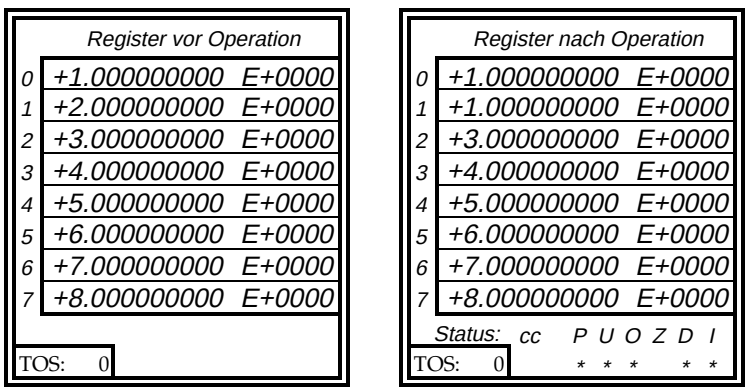
FADD, FSUB, FMUL, FDIV

Doch nun zu den artihmetischen Operationen mit Realzahlen. Wie einleitend gesagt, stehen die vier Grundrechenarten zur Verfügung: Addition (FADD), Subtraktion (FSUB), Multiplikation (FMUL) und Division (FDIV). Bei allen diesen Rechenbefehlen erwartet der Coprozessor zwei Operanden.

Als Operanden können wie üblich Realzahlen, aber auch Stackregister in jeder Kombination angegeben werden. Allerdings ist eine Bedingung, daß mindestens ein Operand der TOS ist. Wie bei den arithmetischen Prozessorbefehlen ist auch bei den Coprozessorbefehlen der erste Operand nach dem Befehl gleichzeitig Operand und Ziel der Operation, so daß sich das Ergebnis immer im ersten Operanden wiederfindet.

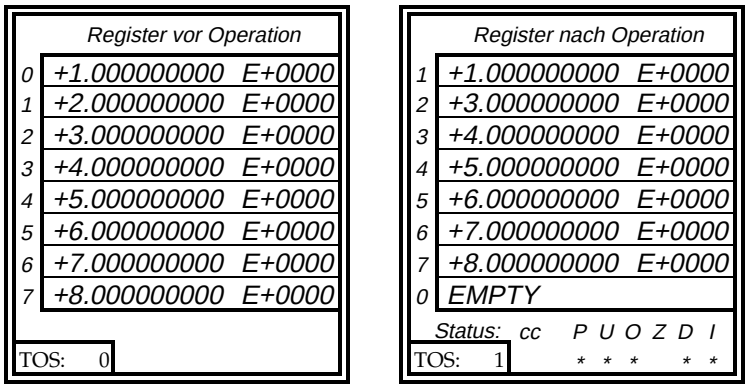
TIP

Ganz im Gegensatz zu der Situation in Taschenrechnern mit Rechenstacks wird beim Coprozessor nach einer arithmetischen Operation kein automatisches Poppen des Stacks durchgeführt. Alle Register bleiben erhalten, lediglich das Zielregister wird mit dem Ergebnis der Operation überschrieben. So erzeugt z.B. der Befehl *FSUB ST(1), ST* folgendes Bild:



FADDP, FSUBP, FMULP, FDIVP

Soll dagegen der Stack wie bei den Taschenrechnern automatisch gepoppt werden, muß der jeweilige »P«-Befehl verwendet werden, also FADDP, FSUBP, FMULP und FDIVP. Im Beispiel von eben heißt das: *FSUBP ST(1), ST*.



ACHTUNG Die »P«-Grundrechenarten arbeiten nicht mit Speicherstellen zusammen. Das heißt, daß FADDP und die anderen nur dann funktionieren, wenn beide Operanden Stackregister sind.

ACHTUNG Beachten Sie bitte die Reihenfolge in der Angabe der Operanden. So ist der Befehl *FSUB ST, ST(1)* nicht mit *FSUB ST(1), ST* identisch. Im ersten Fall wird vom Wert im TOS der Wert des Stackregisters 1 abgezogen und in den TOS geschrieben, während im zweiten Fall vom Wert im Stackregister 1 der Wert im TOS abgezogen wird; das Ergebnis wandert in das Stackregister 1. Abgesehen von der Tatsache, daß im zweiten Fall das negative Resultat vom ersten Fall entsteht und an andere Positionen im Stack geschrieben wird, hat eine falsche Wahl der Reihenfolge der Operanden manchmal fatale Folgen: Den Befehl *FSUBP ST, ST(1)* können Sie nicht verwenden – die Assembler verbieten ihn sogar! Überlegen Sie einmal, warum.

FSUBR, FDIVR Das Problem der Reihenfolge der Operanden ist bei einer Addition und Multiplikation kein Problem: *FADD ST, ST(1)* ergibt das gleiche Resultat wie *FADD ST(1), ST*; nur wird es in unterschiedliche Stackregister geschrieben. Bei Subtraktion und Division dagegen ist auch das Resultat, wie wir gesehen haben, unterschiedlich. Diese Berechnungen sind, wie man so schön sagt, nicht *kommutativ*. Um dem Rechnung zu tragen, gibt es zwei Befehle, die den eigentlichen Berechnungsverlauf umkehren: FSUBR und FDIVR. Während z.B. *FSUB ST(1), ST* etwas leger gesprochen ST von ST(1) abzieht und in ST(1) speichert, zieht *FSUBR ST(1), ST* den Wert aus ST(1) von ST ab und speichert ihn in ST(1). Das heißt also, daß das Ziel das gleiche bleibt, die Operation jedoch umgekehrt abläuft. Dies ist manchmal sehr nützlich.

FSUBRP, FDIVRP Selbstverständlich gibt es auch FSUBR und FDIVR in einer »P«-Version, nämlich FSUBRP und FDIVRP. Sie unterscheiden sich von

FSUBR und FDIVR nur darin, daß anschließend auch hier der Stack gepoppt wird. Aber auch hier macht *FSUBRP ST, ST(1)* wenig Sinn!

ACHTUNG

Bitte denken Sie auch bei den *revers* arbeitenden Grundrechenarten DIV und SUB daran, daß die poppbaren Versionen keinen Sinn machen, wenn mit Speicherstellen gearbeitet wird. Daher gibt es nur die Möglichkeit, diese Befehle in Verbindung mit Stackregistern für beide Operanden zu verwenden.

FCOM, FCOMP

Fehlt noch der arithmetische Vergleich FCOM. Für den arithmetischen Vergleich gilt das, was auch schon bei der Besprechung des analogen Befehls CMP des 8086 gesagt wurde. FCOM ist eigentlich eine Subtraktion, bei der lediglich Flags gesetzt, die Registerinhalte also nicht verändert werden. Auch hier steht das »P« im Befehl FCOMP für Pop, so daß nach dem Vergleich ein Poppen des Stacks durchgeführt wird.

Dennoch besteht ein Unterschied zu den anderen arithmetischen Operationen: FCOM und FCOMP vergleichen immer den TOS mit einem Operanden, so daß nur ein Operand als Parameter gültig ist: FCOM *real* bzw. FCOMP *real*.

Das Ergebnis des Vergleichs wird in *CC*, also als Condition Code, zurückgegeben.

FCOMPP

FCOMPP ist ein weiterer Vergleichsbefehl, der jedoch überhaupt keinen Parameter mehr zuläßt. Er vergleicht ST mit ST(1), bietet dafür aber einen anderen Vorteil: Die beiden Register, die eben verglichen wurden, werden nach dem Vergleich gepoppt.

ACHTUNG

Erinnern Sie sich daran, was wir bei der Besprechung der Register des 8087 festgestellt haben: Der Condition Code spiegelt bei arithmetischen Operationen, wie FCOMxx es sind, eigentlich Flagstellungen eines imaginären Overflow- und Zero-Flags des 8087 wider. Das heißt, nach Vergleichen können folgende Flagkombinationen bestehen, die sich dann in einem bestimmten Wert des Condition Code äußern:

C3	C2	C1	C0	CC	Bedeutung
0	0	0	0	0	Operand 1 > Operand 2
0	0	0	1	1	Operand 1 < Operand 2
1	0	0	0	8	Operand 1 = Operand 2
1	x	x	1	≥9	Operanden nicht vergleichbar

HINWEIS

Denken Sie bitte daran, daß C2 und C1 bei arithmetischen Operationen bedeutungslos sind, also immer auf 0 gesetzt werden. C3 spiegelt das Zero-Flag wieder, C0 das Overflow-Flag. Die Kombination von gesetztem Zero-Flag und Overflow-Flag signalisiert, daß etwas mit den Operanden nicht stimmt. Das könnte z.B. der Vergleich einer Zahl mit einer NaN sein.

In diesem Fall kann man sich nicht mehr unbedingt darauf verlassen, daß C2 und C1 gelöscht sind, weshalb CC hier Werte zwischen 9 und 15 annehmen kann. Aber, wie schon gesagt, es werden in Programmen sowieso nur die Flagstellungen ausgewertet, nicht der Zahlenwert CC, so daß dieses Faktum eigentlich unwichtig ist.

TIP Denken Sie an dieser Stelle daran, daß es ja die Möglichkeit gibt, den Condition Code auch mit den Flags des 8086 auszuwerten. Wenn Sie noch im Hinterkopf haben, daß C3 dem Zero-Flag entspricht und C0 dem Carry-Flag, so läßt sich, vorausgesetzt, wir können das *Statuswort* des 8087 in das Flagregister des 8086 kopieren, auf die FCOMxx-Befehle über bedingte Sprünge reagieren. Mit JB kann dann gesprungen werden, wenn der Operand 1 kleiner als Operand 2 ist, mit JA im umgekehrten Fall. JE läßt eine Verzweigung zu, wenn beide Operanden gleich sind, JNE, wenn nicht. Natürlich funktionieren auch die anderen bedingten Sprünge: JAE, JBE, JNA, JNB, JNAE, JNBE.

ACHTUNG So genial sich die Entwickler der Chips dies auch ausgedacht haben, eine kleine Unstimmigkeit bleibt dennoch bestehen. Wie Sie sich erinnern, ist das Carry-Flag, das die oben genannten bedingten Sprungbefehle prüft, beim Vergleich zweier vorzeichenloser Zahlen durch den 8086 involviert, bei vorzeichenbehafteten ist es das Overflow-Flag. Da der 8087 grundsätzlich mit vorzeichenbehafteten Zahlen arbeitet, besteht hier keine Notwendigkeit, zwischen Overflow- und Carry-Flag zu unterscheiden, und es wird daher auch nicht getan: Es gibt nur C0. Allerdings wird es nun nicht konsequenterweise mit dem Overflow-Flag des 8086 behaftet, sondern etwas unglücklich mit dem Carry-Flag.

4.2 Weitere arithmetische Operationen mit Realzahlen

Zusätzlich zu den eben beschriebenen Grundrechenarten verfügt der 8087 natürlich noch über weitere Operationen mit Realzahlen. Es sind dies:

- FPTAN, FPATAN
- FSQRT
- FLDLG2, FLDLN2, FLDL2E, FLDL2T, FLDPI, FLDZ, FLD1
- FXTRACT
- FSCALE
- FYL2X
- FYL2XP1
- F2XM1
- FRNDINT
- FPREM

FPTAN

FPTAN berechnet den *Tangens* einer Zahl. Aber wofür steht das »P« im Namen? Antwort: Für »*partial*«, also für partiell! Denn es wäre richtig schlimm, wenn FPTAN nur den *Tangens* berechnen könnte. Da es keine weiteren trigonometrischen Funktionen gibt, könnte man andernfalls nicht den *Sinus*, *Cosinus* oder *Cotangens* berechnen.

FPTAN berechnet aus dem Wert in ST(0), also im TOS, zwei Zahlen, die es in ST(1) und ST(0) ablegt. Diese beiden Zahlen können nun für die Berechnung der trigonometrischen Funktionen benutzt werden: so führt eine einfache Division der beiden Werte zum *Tangens*, reziprok dividiert zum Cotangens. Nach einer Formel, die wir im zweiten Teil des Buches kennenlernen werden, können auch der Sinus und der Cosinus aus diesen beiden Zahlen ermittelt werden.

Das »P« bedeutet also, daß wir noch Rechenarbeit leisten müssen, wenn wir wirklich den Tangens benötigen; der Befehl selbst liefert ihn nicht zurück. Wenn ein Befehl zwei Ergebnisse liefert, wird der Stack gepusht, um Platz für die beiden Zahlen zu machen. Der ursprüngliche Wert in ST(0) wird überschrieben. Beachten Sie bitte auch, daß gültige Wertebereiche für die Operation lediglich Zahlen zwischen 0 und $\pi/4$ sind; d.h. die Winkel müssen im Bogenmaß eingegeben werden und können nur im Bereich 0 bis 45° berechnet werden. Das schadet aber nicht, da die trigonometrischen Funktionen ja symmetrisch sind. Alle anderen Winkel können daher ebenfalls berechnet werden (siehe Teil 2). FPTAN auf $\pi/8$, also 22,5°, angewandt, liefert demnach:

Register vor Operation	
1	+3.926990817 E-0001
2	+2.000000000 E+0000
3	+3.000000000 E+0000
4	+4.000000000 E+0000
5	+5.000000000 E+0000
6	+6.000000000 E+0000
7	+7.000000000 E+0000
0	EMPTY
TOS: 1	

Register nach Operation	
0	+1.000000000 E+0000
1	+4.142135624 E-0001
2	+2.000000000 E+0000
3	+3.000000000 E+0000
4	+4.000000000 E+0000
5	+5.000000000 E+0000
6	+6.000000000 E+0000
7	+7.000000000 E+0000
Status: cc	P U O Z D I
TOS: 0	* *

Wie Sie sehen, führt eine einfache Division nach *FDIVP ST(1), ST* zu 0,4142135624, dem korrekten Ergebnis für TAN(22,5°). *FDIVRP ST(1), ST* würde den COT(22,5°) liefern.

FPATAN

FPATAN ist das Gegenstück zu FPTAN und bildet aus vorgegebenen (partiellen) Werten in ST(0) und ST(1) den *Arcus Tangens*. FPATAN direkt nach FPTAN ausgeführt ergibt den ursprünglichen Wert! Wozu das benutzt werden kann, sehen wir in Teil 2.

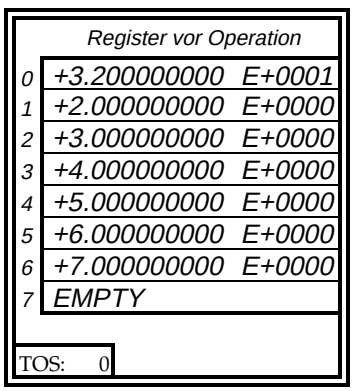
FSQRT FSQRT ersetzt den Wert im TOS durch seine Quadratwurzel, so daß sich hinsichtlich des Stackzustandes nichts ändert.

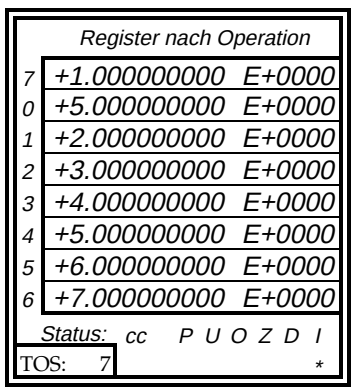
FLDLG2, FLDLN2, FLDL2E, FLDL2T, FLDPI, FLDZ, FLD1. Diese Operationen laden lediglich einen vorgegebenen Wert in das Register, auf das ST zeigt. So lädt FLDLG2 den dekadischen Logarithmus von 2, also Log(2) in den TOS, FLDLN2 den natürlichen Logarithmus von 2, also Ln(2). Wie wir im zweiten Teil des Buches noch sehen werden, spielen diese Konstanten bei den verschiedensten Berechnungen eine bedeutende Rolle.

FLDL2E und FLDL2T laden den Logartihmus Dualis von *e*, der *Eulerschen Konstanten*, und von 10 in den TOS. Auch diese beiden Konstanten werden uns noch wertvolle Dienste leisten.

FLDPI, FLDZ und FLD1 laden die Konstanten π, 0 und 1 in den TOS. Bitte beachten Sie bei diesen Ladebefehlen, daß sie wie der »normale« Ladebefehl FLD arbeiten, also auch den Stack-Pointer dekrementieren. Auch alle anderen Eigenschaften des FLD-Befehls, wie z.B. die Reaktion auf nicht leere Register, sind hiervon betroffen.

FXTRACT FXTRACT zerlegt die Zahl im TOS in ihre Mantisse und den Exponenten. Dies erfolgt, indem zunächst die Mantisse temporär zwischengespeichert wird und dann der Exponent der Zahl in den TOS geschrieben wird. Anschließend erfolgt ein Pushen des Stacks, also ein Dekrementieren des Stack-Pointers. In den auf diese Weise geleerten TOS wird dann die temporär gespeicherte Mantisse geschrieben.





Der Wert von 32 im TOS findet sich also nach FXTRACT in Form seiner Mantisse in ST und seines Exponenten in ST(1) wieder.

ACHTUNG

Hoppla! Sollte bei dieser Art der Zerlegung nicht 3,2 als Mantisse und 1 als Exponent herauskommen? Ist dann aber nicht das Schaubild oben falsch? Denn dort steht als Exponent 5, als Mantisse 1. Nein, es ist alles in Ordnung! Vergessen Sie bitte nicht, daß der Prozessor und auch der Coprozessor mit Binärzahlen arbeiten. Daher wird auch die Zahl binär zerlegt. Und 32 ist $1 \cdot 2^5$, womit wir zu einer Mantisse von 1 und einem Exponenten von 5 kommen.

FSCALE

Auch das Umgekehrte geht: Erheben des Wertes in ST(1) zur Basis 2 und Multiplikation mit dem Wert im TOS. Das Ganze nennt sich dann FSACLE:

Register vor Operation	
0	+1.000000000 E+0000
1	+5.000000000 E+0000
2	+2.000000000 E+0000
3	+3.000000000 E+0000
4	+4.000000000 E+0000
5	+5.000000000 E+0000
6	+6.000000000 E+0000
7	+7.000000000 E+0000

TOS: 0

Register nach Operation	
0	+3.200000000 E+0001
1	+5.000000000 E+0000
2	+2.000000000 E+0000
3	+3.000000000 E+0000
4	+4.000000000 E+0000
5	+5.000000000 E+0000
6	+6.000000000 E+0000
7	+7.000000000 E+0000

Status: cc P U O Z D I

TOS: 0 * * *

FSCALE beläßt den Zustand des Stacks so, wie er ist; es wird lediglich der Wert im TOS mit dem Ergebnis von $ST(0) \cdot 2^{ST(1)}$ überschrieben.

HINWEIS

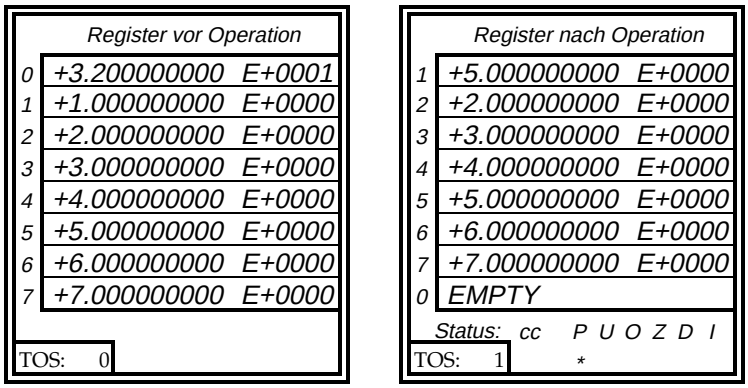
Die Werte in ST(1), also die Exponenten, sind sowohl bei FXTRACT als auch bei FSCALE Integer. Sie liegen zwar immer als Realzahl vor; gebrochene Exponenten aber liefert FXTRACT niemals, und sie werden, falls übergeben, von FSCALE als Integer ohne Nachkommateil interpretiert.

FYL2X

FYL2X berechnet einen Wert nach der Formel $Z = Y \cdot Ld(X)$, bildet also von X den Logarithmus Dualis und multipliziert das Ergebnis mit Y. Dies erfolgt, indem zunächst der Logarithmus Dualis vom Wert im TOS gebildet wird und mit dem Wert in ST(1) multipliziert wird. Das Ergebnis wird in ST(1) gespeichert und anschließend gepoppt (siehe Abbildung auf der nächsten Seite).

FYL2XP1

FYL2XP1 ist eine leichte Abwandlung des eben besprochenen FYL2X. Er unterscheidet sich von diesem nur dadurch, daß zunächst 1 zu dem Wert in ST addiert wird, bevor dann der Logarithmus Dualis gebildet und mit ST(1) multipliziert wird.



Der Grund dafür ist, daß das Resultat komplexer Berechnungen durch Logarithmierung – Antilogarithmierung mit Werten sehr nahe an 0 dadurch sehr viel genauer wird. Aber Achtung: gültige Werte sind nur diejenigen zwischen 0 und 0,5 – einschließlich 0, ausschließlich 0,5, also in mathematischer Schreibweise: [0; 0,5[.

F2XM1 F2XM1 ist die Umkehrfunktion zu FYL2XP1. Sie potenziert zunächst den Wert in ST zur Basis 2 und zieht dann vom Ergebnis 1 ab. Es wird somit nur der Inhalt im TOS verändert, nicht aber der Inhalt anderer Stackregister.

FRNDINT FRNDINT rundet eine Realzahl zu einer Integerzahl. Das Ergebnis liegt zwar noch immer als Realzahl vor, doch ist der Nachkommateil 0. Somit kann durch einfaches Abspeichern dieser »Realzahl« im Integerformat eine »echte« Integerzahl erzeugt werden. Auf welche Weise gerundet wird, bestimmen die Flags *Round Control* im *Kontrollwortregister*.

FPREM Wann immer ein »P« mitten im Wort steht: es steht für *partial*! Während »P«s am Ende eines *Mnemonics* für Poppen steht, zeigt das andere »P« an, daß etwas unvollständig durchgeführt wurde! Bei FPTAN war das die Bildung des Tangens, bei *Partial Remainder* (FPREM) ist es eine Division. FPREM dividiert ST(0) durch ST(1). Der Rest wird in ST(0), dem TOS, abgelegt. Der entstandene Quotient wird verworfen, zumindest zum größten Teil. Die niederwertigen drei Bits werden im CC, dem Condition Code, gespeichert, und zwar als C3, C1 und C0. C2 ist ein Flag und zeigt an, ob die Reduktion vollständig durchgeführt werden konnte oder nicht.

FPREM führt in Wirklichkeit keine Division aus, sondern eine x-fach wiederholte Subtraktion. Der Grund ist klar: Das Ergebnis wird dadurch exakt! Aber einen Nachteil hat das ganze natürlich auch: Stellen Sie sich vor, Sie wollen den Rest von 1,23456789E+4096 dividiert durch 3 wissen. Dies bedeutete, daß FPREM so lange von dieser Zahl 3 abziehen müßte, bis ein Rest entsteht, also ca. 4E+4095mal. Ein Co-

prozessor mit 100 MHz Taktfrequenz würde, hätte man ihn zum Anbeginn der Welt mit dieser Aufgabe betraut, auch heute noch munter weiter subtrahieren und in einer halben Ewigkeit auch noch! Also entschloß man sich, eine Grenze zu setzen, jenseits derer die Bildung des *Partial Remainders* (daher FPREM) abzubrechen ist. Diese Information gibt C2 wieder: Ist dieses Flag gesetzt, können Sie den Wert in ST(0) getrost vergessen. Er war für FPREM viel zu groß. C2 = 0 dagegen besagt, daß im TOS der absolut exakte Wert für den Divisionsrest liegt.

Wo nun liegt die Grenze? Das hängt davon ab, welchen Divisor Sie vorgeben. Wenn Sie z.B. 1.0E+20 durch 8 dividieren, so erhalten Sie einen gültigen Rest von 0. Dividieren Sie die gleiche Zahl dagegen durch 7, so ist die Grenze der maximal durchführbaren Subtraktionen überschritten, und C2 wird gesetzt. Der dann in ST(0) stehende Wert ist der Rest, der übrig geblieben ist, nachdem die maximale Anzahl an Subtraktionen erfolgt ist. Wie gesagt, diesen Wert können Sie vergessen – oder zumindest fast! Denn Sie haben nun weder ein Divisionsergebnis noch einen korrekten Divisionsrest. Ich sage deshalb »fast«, weil es Ihnen ja niemand nimmt, auch diesen Rest wieder mit FPREM zu reduzieren – der Rest als solches ist zwar nicht korrekt, aber exakt! Vielleicht klappt es ja dieses Mal. Ich überlasse Ihnen, was Sie tun, wenn nicht! Wie oft hintereinander Sie FPREM mit dem jeweils unvollständig reduzierten Rest benutzen, sollen Sie selbst entscheiden – mir jedenfalls ist die halbe Ewigkeit ein bißchen zu lang. Allgemein kann gesagt werden, daß mit einem Durchlauf von FPREM eine Reduktion um den Faktor 2^{64} (= $1.844 \cdot 10^{19}$) erreicht werden kann.

Die niederwertigen drei Bits des »Divisionsergebnisses« finden sich wie folgt in CC wieder: C3 enthält Bit 1 des Divisionsergebnisses, C1 Bit 0 und C2 Bit 2. Demonstrieren wir dies an einem Beispiel:

Register vor Operation	
0	+1.490000000 E+0002
1	+1.600000000 E+0001
2	+2.000000000 E+0000
3	+3.000000000 E+0000
4	+4.000000000 E+0000
5	+5.000000000 E+0000
6	+6.000000000 E+0000
7	+7.000000000 E+0000

TOS: 0

Register nach Operation	
1	+5.000000000 E+0000
2	+1.600000000 E+0001
3	+2.000000000 E+0000
4	+3.000000000 E+0000
5	+4.000000000 E+0000
6	+5.000000000 E+0000
7	+7.000000000 E+0000
0	+7.000000000 E+0000

Status:	cc	P	U	O	Z	D	I
TOS: 1	2		*			*	*

149 dividiert durch 16 ergibt 9, Rest 5. Folgerichtig steht in ST(0) der Rest, hier also 5. Der *Control-Code* ist $2, hat also die Bitfolge 0010. Hieraus sieht man, daß C1 gesetzt ist und C3 und C0 gelöscht sind. Nach der oben genannten Beziehung ist also die Bitfolge der letzten drei Bits des Divisionsergebnisses 001. Diese Bits sind auch im (verworfenen) Ergebnis, 9, gesetzt: 1001. C2 = 0 signalisiert, daß das Ergebnis vollständig reduziert ist.

ACHTUNG Wenn bisher der Eindruck erweckt wurde, daß mit FPREM nur Integerzahlen manipuliert werden können, so ist dies nicht beabsichtigt, sondern sollte nur das Beispiel vereinfachen! Zwar fällt einem bei einer Restbildung nach Division unweigerlich die Integerdivision DIV/IDIV des Prozessors ein, die eben nur Integer behandeln kann und daher als Ergebnis einer Division nur einen Quotienten und einen Divisionsrest im Integerformat liefern kann. Dennoch ist dies bei FPREM nicht der Fall.

Vielmehr dürfte sich sogar das Hauptanwendungsgebiet von FPREM in der Bearbeitung von Realzahlen befinden – sogar in einem sehr speziellen Fall, nämlich bei den trigonometrischen Funktionen wie Sinus, Cosinus usw. Dies scheint zunächst sehr befremdlich, da man gerade bei Fließkommazahlen in Verbindung mit dem Coprozessor ja genaue Ergebnisse nach Divisionen erwartet. Das ist zwar in der Regel auch richtig und die Rundungsfehler nach/bei Divisonen werden selten ins Gewicht fallen.

Dennoch schaukeln sie sich in manchen Fällen hoch. Wollen Sie z.B. *sin(12345.6789)* berechnen, so müssen Sie zunächst das Argument 12345.6789 durch $\pi/4$ dividieren. Denn der Befehl FPTAN, den Sie zur Berechnung des Sinus heranziehen müssen, kann nur Werte verwenden, deren Betrag zwischen 0 und $\pi/4$ liegt. Der Rest, der bei einer »echten« Division entsteht, ist aber naturgemäß ungenauer als der nach einer »exakten« Subtraktion.

Hierin ist auch der Grund zu sehen, warum in den Bits des Condition Codes ausgerechnet die niedrigsten drei Bits des Ergebnisses stehen. Zur Berechnung der trigonometrischen Funktionen ist es nämlich sehr interessant, festzustellen, in welchem Oktanden des Einheitskreises das Argument anzusiedeln ist. Drei Bits codieren Zahlen zwischen 0 und 7 – genau die acht möglichen Oktanden. FPREM liefert somit nicht nur das richtig auf die Fähigkeiten von FPTAN abgebildete Argument, sondern auch den korrekten Oktanden dazu. Er kann wie folgt berechnet werden: $O = 4 \cdot C2 + 2 \cdot C3 + C1$.

TIP Wie schon mehrfach erwähnt, können wir mit dem Statuswort auch das Flagregister laden und den Condition Code auf diese Weise auswerten. Hierbei zeigt sich, daß für die Frage »gültige – ungültige Reduktion« das Parity-Flag zuständig ist, das mit C2 kommuniziert. Al-

somüssen Sie das Statuswort ins Flagregister kopieren und prüfen, ob das Parity-Flag gesetzt ist. Wenn ja, ist was faul. Übrigens kristallisiert sich hier vielleicht doch noch eine Existenzberechtigung für das Parity-Flag heraus.

4.3 Arithmetische Operationen mit Integerzahlen

Unter Integerzahlen versteht ein 8087 drei Arten von Zahlen: Ganzzahlen mit 16, 32 oder 64 Bit Breite. Dies sind also Integerzahlen mit Größen von 2 (*Word Integer*), 4 (*Short Integer*) oder 8 Bytes (*Long Integer*). Der Prozessor selbst kann außer mit den ersteren mit den anderen Zahlen nicht mehr rechnen, zumindest nicht mehr so einfach, da für ihn eine Integer entweder 8 Bit (1 Byte) oder 16 Bit (2 Byte) breit ist.

ACHTUNG

An dieser Stelle kann es leicht zu Verständnisschwierigkeiten kommen. Jeder benennt hier Datentypen so, wie er es gern möchte:

PASCAL

In Quick Pascal und Turbo Pascal gibt es neben den Datentypen *Byte* und *Word*, die mit den Integerbefehlen des 8086 manipuliert werden können, auch die Datentypen *ShortInt*, *Integer* und *LongInt*. Bei den ShortInts handelt es sich um vorzeichenbehaftete Bytes, d.h. um Zahlen, deren Bit 7 das Vorzeichen repräsentiert und die damit einen Wertebereich von -128 bis 127 haben können. Dieser Datentyp wird weder vom 8086 noch vom 8087 explizit unterstützt. Zwar ist mit den mathematischen Opcodes des 8086 analog zu den Bytes einiges machbar, allerdings muß dann sorgsam auf das Vorzeichen geachtet werden.

Der Datentyp Integer aus Turbo Pascal ist mit dem Datentyp *Word Integer* des 8087 identisch. Ebenso läßt sich der Turbo-Pascal-Datentyp LongInt mit dem *Short Integer* des Coprozessors realisieren. *Long Integer* dagegen finden sich im Datentyp *Comp* von Turbo Pascal wieder.

C

Doch auch C macht sich die Sache hier sehr leicht. Was in C eine *Short Int* ist, kennt der Coprozessor überhaupt nicht, egal ob *signed* oder *unsigned*! Auch *signed* oder *unsigned Int*s sind ihm vollkommen fremd – dies sind sowieso die gleichen Datentypen wie die *Short Ints*. Das, was der Coprozessor unter *ShortInt* versteht, ist in C eine *signed Long Int*, während eine *Long Int* in C als *double Long Int* bezeichnet wird!

Zur Übersicht finden Sie auf der nächsten Seite eine Tabelle, die den Zusammenhang klarstellt.

HINWEIS

Intern rechnet der 8087 immer mit 80 Bit breiten TEMPREALS. Auch bei Integerzahlen ist dies nicht anders. Die Interpretation dieser TEMPREAL erfolgt also durch die Lade- und Speicherbefehle FILD und FIST (FISTP).

Größe	8086[1]	8087[2]	ASM[3]	Pascal[4]	C[5]
8 Bits	BYTE	-	BYTE	byte[1]	unsigned char
8 Bits	BYTE	-	BYTE	short int[2]	signed char
16 Bits	WORD	-	WORD	word[1]	unsigned int
16 Bits	WORD	WORDINT	WORD	integer[2]	signed int
32 Bits	2 WORDs[6]	-	DWORD	-	unsigned long int
32 Bits	2 WORDs[6]	SHORTINT	DWORD	longint[2]	signed long int
64 Bits	4 WORDs[6]	LONGINT	QWORD	comp	double long int

1 vorzeichenlos
2 vorzeichenbehaftet
3 Unterschiedliche Assembler kennen eventuell noch andere Datentypen, die aber nicht unbedingt zum Standardumfang gehören und daher nur verwendet werden sollten, wenn Assembler-Kompatibilität nicht erforderlich ist. So kennt z.B. MASM 6.0 auch die Datentypen SBYTE, SWORD und SDWORD für vorzeichenbehaftete BYTEs, WORDs und DWORDs.
4 Microsoft Professional Pascal 4.0 geht auch hier eigene Wege: *integer2* ist das, was in Turbo Pascal und Quick-Pascal eine *integer* ist, während eine *integer4* ihr Pendant in der beliebten *longint* hat. Dafür ist *short int* in MPP 4.0 unbekannt, wie auch *comp*.
5 Auch hier brilliert *C* durch einheitliche Namensgebung in den unterschiedlichen Dialekten.
6 Durch spezielle Befehle des 8086 sind auch Berechnungen mit mehr als 2 Bytes möglich.

Das heißt aber, daß FILD eine Integerzahl des eben besprochenen Datentyps liest, sie in eine TEMPREAL umwandelt und in ein Register schreibt. Umgekehrt wandelt FIST (FISTP) zunächst die TEMPREAL im TOS in eine Integerzahl des entsprechenden Typs um, bevor sie an die entsprechende Stelle geschrieben wird.

ACHTUNG Das heißt aber, daß alle Berechnungen des 8087 mit Integerzahlen in Wirklichkeit Realzahlberechnungen sind. Es können also alle mathematischen Operationen, die wir bisher kennengelernt haben, auch mit Integerzahlen durchgeführt werden.

Diese Aussage ist richtig, wenn auch noch unvollständig. Wie wir bei den Realzahlen gesehen haben, können einige Befehle direkt mit Realzahlen aus dem Speicher ausgeführt werden, also z.B. *FADD ST, ExtendedVar*. Nach dem eben Gesagten könnte man nun versucht sein, mittels FADD auch Integer direkt zu benutzen, also etwa nach *FADD ST, LongIntVar*. Das funktioniert nicht! FADD, auf Speicherstellen angewandt, erwartet dort Realzahlen mit Mantisse und Exponenten. Integer haben keine Exponenten, so daß zunächst eine Umwandlung erfolgen muß, die z.B. FILD durchführt.

Um solche Berechnungen zu ermöglichen, müssen Befehle her, die dies können. Diese Befehle sind:

- FIADD
- FICOM, FICOMP
- FIDIV, FIDIVR
- FILD

- FIMUL
- FIST, FISTP
- FISUB, FISUBR

HINWEIS

FILD, FIST und FISTP sind nichts anderes als FLD, FST und FSTP, nur wandeln sie vor/nach der Datenübertragung die Integer von/in TEMPREALS um. Nach dem bisher Gesagten sollten dann die anderen arithmetischen Befehle für Integerzahlen auch nur die Realzahlbefehle mit automatischer Umwandlung sein. So ist es auch! Aus diesem Grunde sind die Befehle auch ziemlich unflexibel. Da lediglich eine Integerzahl direkt mit dem Inhalt eines Registers über eine Grundrechenart verknüpft werden soll, gibt es nur die Möglichkeit, als Operand eine Quelle, also eine Speicherstelle anzugeben. Verknüpft werden kann auch jeweils nur der TOS. Typische Befehle sind also *FIADD LongIntVar* oder *FISUBR CompVar*.

Die Wirkungsweise dieser Grundrechenarten sowie der Vergleichsbefehle ist in jedem Fall die gleiche wie bei den Realzahlen. Da bei den Integerbefehlen immer ein Speicheroperand involviert ist, gibt es mit FICOMP – logischerweise – auch nur den einfachen »P«-Befehl für FICOM. Und weil analog zu der Konstruktion *FADDP ST, ST(1)* bei Realzahlen ein Poppen nach FIADD keinen Sinn macht, gibt es für die Grundrechenarten auch keine »P«-Befehle!

ACHTUNG

Glauben Sie nicht, FIDIV wäre ein DIV-Befehl des Prozessors für den Coprozessor, um damit Ganzzahlen zu bearbeiten, die länger als 2 Byte sind! Während DIV mit Integer eine sogenannte Ganzzahldivision durchführt, tut der 8087 mit FIDIV dies beileibe nicht. DIV dividert zwei Zahlen durcheinander und schneidet dann den Nachkommateil ab, also z.B. 3 DIV 2 = (1,5) = 1. FIDIV dagegen führt eine »richtige« Division aus: 3 FIDIV 2 = 1,5.

Dies ist zwar befremdlich, aber durchaus konsequent und richtig! Denn intern wird das Ergebnis der Division als Realzahl dargestellt, ja mehr noch: Nach dem Laden der Integer sind es gar keine Integer mehr. Beachten Sie daher bitte, daß die Sequenz

```
fild     LongIntVar1            { enthält 3 }
fidiv    LongIntVar2            { enthält 2 }
fist     LongIntVar3
```

tatsächlich zu einer »1« in *LongIntVar3* führt. Aber nicht etwa, weil FIDIV dies so getan hätte, sondern wegen FIST, das die Umwandlung der TEMPREAL in eine SHORTINTEGER vornimmt, indem es einfach den Nachkommateil abschneidet. Beachtet man dies nicht, können unvorhersehbare Ergebnisse entstehen:

```
fild     LongIntVar1            { enthält 3 }
fidiv    LongIntVar2            { enthält 2 }
fimul    LongIntVar1
fist     LongIntVar1
```

Die Ganzzahldivision von 3 durch 2 führt, wie wir gesehen haben, zu 1,5. Die anschließende Multiplikation mit 3 ergibt daher 4,5, so daß ein nun folgender FIST-Befehl die Ganzzahl 4 in den Speicher schreibt, obwohl 3 der für Ganzzahloperationen korrekte Wert wäre.

TIP

Die Moral von der Geschicht': Integer sind für den 8087 lediglich besondere Realzahlen. Nur einige wenige Befehle für den Umgang mit Integer behandeln diese auch entsprechend. Daher mein Tip: Vermeiden Sie, vor allem als Anfänger, die Verwendung von Integer beim 8087. Die möglichen (versteckten) Fehler sind nach dem *Prinzip von Murphy* schon so gut wie gemacht und das (offensichtlich) konstant falsche Rechenergebnis frustriert und demoralisiert ungemein. Wenn Sie aber dennoch so große Integerwerte benutzen wollen oder müssen, daß Sie mit den Befehlen des 8086 nicht mehr zurechtkommen, so denken Sie immer daran: für den 8087 sind Integer Realzahlen ohne Exponenten, aber mit Mantisse und somit mit Nachkommateil! Einem FIDIV oder auch FDIV, so die Daten schon in den Registern sind, sollte dann immer ein FRNDINT folgen, wobei »in Richtung 0 gerundet wird« (dazu müssen die Bits im *Kontrollwort* korrekt gesetzt werden!).

4.4 Weitere Operationen mit Zahlen

Es gibt aber noch weitere Operationen, die sich auf alle Zahlen beziehen können, die der 8087 kennt:

- FABS
- FCHS

FABS, FCHS

FABS bildet den Absolutwert der Zahl in ST(0), FCHS multipliziert ST(0) mit -1. Beide Funktionen verändern darüber hinaus den Stack nicht.

4.5 Sonderfall: BCDs

Neben Integer und Reals kennt auch der 8087, wie der 8086, BCDs. Aber was er macht, macht der 8087 richtig! Das heißt, daß er nicht nur eine BCD-Ziffer, maximal zwei, bearbeiten kann, wie der 8086. Er rechnet auch mit »richtigen« BCDs mit einer Länge bis zu 18 Ziffern, auch mit Vorzeichen.

Aber er realisiert dies wie mit den Integer auch: intern stellt er die Zahl als TEMPREAL dar, so daß mit den normalen Rechenbefehlen gearbeitet werden kann. Die Umwandlung von BCD in das interne Format erfolgt, wie bei den Integer auch, durch den Ladebefehl, die Umwandlung vom internen Format in die BCD durch den Speicherbefehl.

Aus diesem Grunde gibt es auch nur diese beiden Befehle für BCDs:

- FBLD
- FBSTP

FBLD, FBSTP

FBLD lädt eine gepackte BCD in ST(0), so daß als Parameter nur die Speicherstelle angegeben werden muß. Andererseits speichert FBSTP den Wert im TOS mit anschließendem Poppen wieder an eine Speicherstelle im BCD-Format. Auch hier wird daher nur das Ziel, die Speicherstelle, angegeben.

HINWEIS

Wie wenig Interesse man den BCDs üblicherweise schenkt, drückt sich auch darin aus, daß Operationen unter direkter Beteiligung von Speicherstellen, wie sie bei Realzahlen ausführlich, bei Integer nur noch eingeschränkt möglich sind, bei BCDs nicht mehr durchgeführt werden können.

4.6 Rechenstackmanipulationen

Nun können wir rechnen! Was uns noch fehlt, sind ein paar Werkzeuge, die uns das Leben erleichtern sollen. Was z.B. machen wir mit Inkonsistenzen im Speicher, wenn wir nicht immer korrekt gepoppt haben, vielleicht weil es nicht möglich war? Also mit Werten, die zwar noch im Register stehen, aber nicht mehr gebraucht werden? Wie wir wissen, kann dies zu Problemen führen, z.B. wenn im untersten Register – also ST(7) – noch ein Wert steht, wir aber dringend FLD ausführen müssen. Das Ergebnis wäre eine NaN mit gesetztem Invalid-Operation-Flag. Oder was machen wir, wenn wir dringend am Wert in ST(3) etwas verändern müssen, ohne den Stack insgesamt verändern zu dürfen. Oder wenn wir einfach feststellen wollen, was für eine Zahl sich im TOS befindet. Dazu gibt es folgende Befehle:

- FFREE
- FDECSTP
- FINCSTP
- FXAM
- FXCH
- FTST

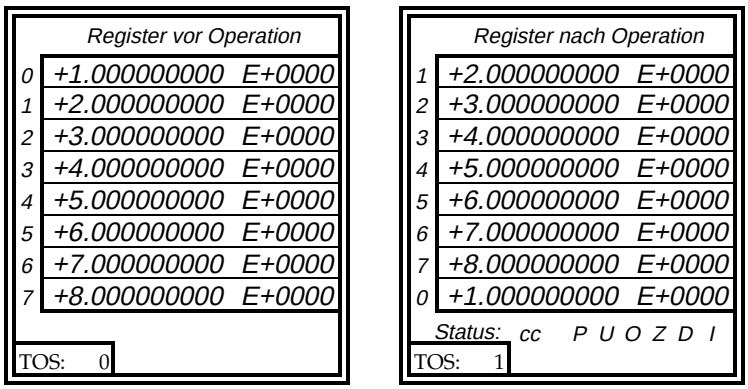
FFREE

FFREE entfernt Inkonsistenzen. Durch die Angabe, welches Register betroffen ist, kann dieses geleert werden. So erklärt FFREE ST(7) z.B. das unterste Register im Stack als leer. Probleme wie oben geschildert kann es also ab jetzt nicht mehr geben! Aber Vorsicht: FFREE ist rigoros! Ein mit FFREE freigegebenes Register ist wirklich leer – enthaltene Daten sind rettungslos verloren.

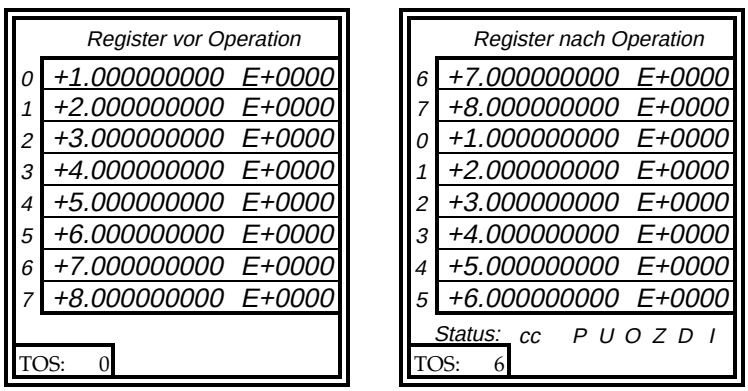
FDECSTP, FINCSTP

Keine Regel ohne Ausnahme! Mit FINCSTP und FDECSTP gibt es zwei »P«-Befehle, die nichts poppen. Hier ist das »P« Bestandteil von »STP« und bedeutet Stack-Pointer. FINCSTP inkrementiert, FDECSTP dekrementiert den Stack-Pointer.

Das heißt aber, daß die Register des Stacks nur rotieren: bei FINCSTP »nach oben«, bei FDECSTP »nach unten«. Inhalte werden hierbei nicht verändert: FINCSTP führt zu:

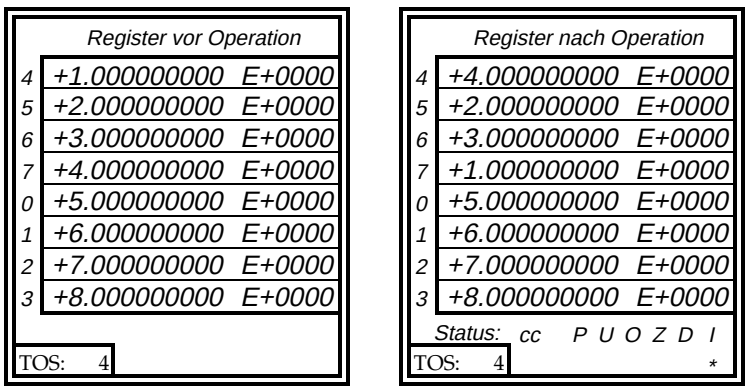


Zwei FDECSTP hintereinander führen dagegen zu:



FXAM untersucht den Inhalt vom TOS. Als Ergebnis wird der Condition Code manipuliert. Eine Tabelle, welche Werte von C3 bis C0 welches Ergebnis codieren, finden Sie im dritten Teil dieses Buches. An dieser Stelle nur so viel: Sie können über FXAM feststellen, ob die »Zahl« im TOS normalisiert, denormalisiert, unnormal, unendlich oder eine NaN ist, ob die Exponenten gültig sind oder nicht und ob die Zahl positiv oder negativ ist. **FXAM**

FXCHG tauscht den Wert im TOS mit einem Wert in einem anzugebenden Register aus. So führt FXCHG ST(3) z.B. zu: **FXCH**



FTST ist ein Befehl, der eigentlich besser in die Rubrik »Arithmetische Operationen« als Vergleichsbefehl gepaßt hätte. Denn FTST testet den TOS gegen 0.0. Praktisch macht FTST das gleiche wie ein FCOM ST(1), wenn man in ST(1) eine Null stehen hätte. Analog werden auch die Bits C3 und C0 im Condition Code gesetzt. **FTST**

Denken Sie daran: C3 kommuniziert mit dem Zero-Flag des 8086, wenn wir den Condition Code in das Flagregister laden, und C_0 mit dem Cary-Flag! **TIP**

4.7 Allgemeine Coprozessorbefehle

Wie bei jedem Problemkreis gibt es auch bei den Coprozessorbefehlen einige »Ausreißer«, die sich nicht einem bestimmten Thema zuordnen lassen.

- FCLEX, FNCLEX
- FENI, FNENI, FDISI, FNDIS
- FINIT, FNINIT
- FSTCW, FNSTCW, FLDCW

- FSTENV, FNSTENV, FLDENV
- FNOP
- FSAVE, FNSAVE, FRSTOR
- FSTSW, FNSTSW
- FWAIT

FCLEX, FNCLEX

FCLEX (*Clear Exceptions*) löscht alle Ausnahmebedingungen. Das heißt, hat ein Befehl zum Setzen eines oder mehrerer der Ausnahmeflags *Precision*, *Underflow*, *Overflow*, *Zero Divide*, *Denormalized* oder *Invalid Operation* geführt, so möchte man, nachdem man dies ausreichend gewürdigt hat, das Programm definiert fortsetzen. Da die meisten Befehle des 8087 diese Flags nur verändern, wenn tatsächlich eine Ausnahmebedingung stattgefunden hat, sonst aber nicht, bleiben die Zustände, einen weiteren korrekten Ablauf vorausgesetzt, bis in alle Ewigkeit so erhalten. Das soll nicht sein. Deswegen gibt es FCLEX. Dieser Befehl setzt alle *Exception-Flags* auf 0 zurück. Aber Achtung: Das betrifft nur die Exception-Flags, nicht den Condition Code!

Das »N« in FNCLEX steht für *No Wait*. FNCLEX arbeitet genau wie FCLEX, mit einer Ausnahme. Vor dem eigentlichen Befehl wird kein WAIT-Befehl eingefügt, der die beiden Prozessoren synchronisiert. Wozu dies gut ist und wie die Synchronisation überhaupt erfolgt, klären wir im nächsten Kapitel.

FENI, FNENI, FDISI, FNDISI

Wie wir bei der Besprechung der Register des 8087 gesehen haben, kann dieser auch Interrupts auslösen, falls eine Ausnahmebedingung eintritt. Üblicherweise erfolgt kein Interrupt – die Ausnahmebedingungen werden von der Berechnungsroutine, die mit den 8087-Befehlen arbeitet, durch Auslesen der entsprechenden Flags bearbeitet.

Dennoch kann es interessant sein, auch Interrupts zuzulassen. Es müssen dann die entsprechenden Flags im Kontrollwort gesetzt und ein Interrupt zugelassen werden. Letzteres erfolgt mit FENI. FDISI dagegen sperrt die Interrupt-Auslösung wieder.

Auch hier bedeutet das »N« in FNENI und FNDISI, daß keine WAIT-Befehle eingestreut werden.

FINIT FNINIT

FINIT (sowie FNINIT) initialisiert den Coprozessor. Dies bedeutet, daß alle Register einen bestimmten Anfangsinhalt bekommen. So erklärt FINIT alle Rechenregister als *empty*. Das Kontrollwortregister (*Control Word register)* erhält den Wert 037Fh:

			IC	RC	PC	IE		P	U	O	Z	D	I
0	0	0	0	0 0	1 1	0	1	1	1	1	1	1	1

Das bedeutet, daß *Infinity Control* 0, also das projektive Modell, gewählt wird, bei dem nicht zwischen -∞ und +∞ unterschieden wird. *Round control* ist 0, wodurch zur nächsten oder geraden Zahl gerundet wird. *PCecision control* = 3 stellt die größte Mantissengenauigkeit von 64 Bits ein (= TEMPREALS), Interrupts werden unterdrückt (*Interrupt Enable* = 0).

Durch Eintragen von 0 in das Statuswortregister werden der Condition Code sowie alle Exception-Flags gelöscht.

ACHTUNG

Bitte beachten Sie, daß FINIT nur dann benutzt werden sollte, wenn der Coprozessor tatsächlich initialisiert werden soll. So schreiben sehr viele Hochsprachen, unter anderem Turbo Pascal und Turbo C, nach der eigentlichen Initialisierung andere Werte in das Kontrollwortregister, um etwas andere Reaktionen zu erhalten. Diese Manipulationen machen Sie durch ein FINIT zunichte, falls Sie die Assemblerroutinen im Rahmen von OBJ-Modulen in Hochsprachen einsetzen.

FINIT muß auch nicht sein! So lassen sich die Register auch mit FFREE als leer markieren. Mit FCLEX können Siedie Flags löschen, und die Sequenz

```
fstcw    DefaultControl
fldcw    MyControl
```

stellt den Coprozessor auf die gewünschten, in einem Kontrollwort in *MyControl* stehenden Bedingungen ein, ohne sich der Freiheit zu berauben, den ursprünglichen Inhalt mittels *FLDCW DefaultControl* restaurieren zu können, wenn die eigenen Routinen ausgeführt wurden.

Einmal jedoch muß FINIT auf jeden Fall aufgerufen werden. Wenn Sie den Assembler also verwenden, um von Hochsprachen, die den Coprozessor nutzen können und ihn daher initialisieren, unabhängige Programme zu schreiben, so müssen Sie FINIT aufrufen.

FSTCW, FNSTCW, FLDCW

FSTCW und FLDCW dienen zum Laden bzw. Speichern des Inhalts des Kontrollwortregisters. FSTCW kopiert den Inhalt dieses Registers in eine als Parameter zu übergebende Wortvariable. FLDCW kopiert aus einer solchen Variablen ein Kontrollwort in das Kontrollwortregister. FLDCW ist die einzige Möglichkeit, dem 8087 beizubringen, bei welchen *Exceptions* (Ausnahmebedingungen), wenn überhaupt, Interrupts ausgelöst werden sollen. Ferner wird über diesen Befehl mit Hilfe des übergebenen Wortes die Präzision sowie die Art der Rundung eingegeben.

FSTENV, FNSTENV, FLDENV

Das *Environment* (die Umgebung) des 8087 ist der Inhalt seiner Nicht-Rechenregister, also des Kontrollworts, des *Statusworts*, des *Tag-Registers*, des *Instruction-Pointer-Registers* sowie des *Data-Pointer-Registers*. FSTENV speichert nun dieses Environment an eine Speicherstelle, die dem Befehl als Operand übergeben wird. Das Environment ist 7 Worte groß. Dem-

nach muß die Variable, die dem Befehl übergeben wird, mindestens 14 Bytes groß sein. FLDENV lädt aus der 14 Bytes großen, als Parameter übergebenen Variablen das Environment wieder, nachdem es vorher dort mit FSTENV abgelegt wurde.

Der Sinn dieses Befehlspaares ist die temporäre Zwischenspeicherung der wichtigen Coprozessorregister, falls die aktuell durchgeführten Coprozessoraktionen aus irgendeinem Grunde unterbrochen werden müssen.

Auch FSTENV gibt es in der »N«-Fasssung!

FNOP

FNOP ist das 8087-Gegenstück zum 8086-Befehl NOP und verlanlaßt hier wie dort Untätigkeit. Nach FNOP tut der Coprozessor während des ganzen Befehls nichts!

FSAVE, FNSAVE, FRSTOR

Sollen neben dem Environment auch die Inhalte der acht Rechenregister gesichert oder restauriert werden, muß FSAVE/FRSTOR verwendet werden.

Dies bläht aber naturgemäß den benötigten Platz der Speichervariablen auf: 8 Register à 80 Bit (= 10 Byte) plus 14 Bytes Environment macht zusammen 94 Bytes, die die Variable groß sein muß! Und auch hier: FSAVE im chicen »N«-Design.

FSTSW, FNSTSW

Mit FSTSW und seinem *NoWait*-Partner kann das Statuswortregister ausgelesen und in eine Wortvariable kopiert werden, die als Parameter übergeben wird.

Hier gibt es nun die Möglichkeit, den Condition Code des 8087 in das Flagregister des 8086 zu schreiben:

TIP

Nachdem man mit FSTSW den Inhalt des Statusworts an eine Speicherstelle schreiben kann, kann man diese Speicherstelle auch mit einem 8086-Befehl auslesen. Der 8086 kennt darüber hinaus den Befehl SAHF, der das obere Byte eines Wortes in AX, also den Inhalt von AH, in den niederwertigen Teil des Flagregisters kopiert, also an die Bitpositionen 7 bis 0. Das bedeutet, daß die Sequenz

```
fstsw    WordVar
mov      ax,WordVar
sahf
```

genau das bewirkt, was wir bisher als möglich in Aussicht gestellt haben: Die Zuordnung der Flags des Condition Codes zu den Flags des 8086. Nach der Befehlsfolge von eben läßt sich mit folgenden 8086-Flags der Zustand des 8087 feststellen:

Sign-Flag	Zero-Flag	Auxiliary-Flag	Parity-Flag	Carry-Flag
Busy-Flag	C3		C2	C0

C1 allerdings läßt sich nicht identifizieren! Wollen Sie C1 prüfen, so muß dies z.B. mit

```
fstsw    WordVar
mov      ax,WordVar
test     ah,002h
```

erfolgen. Ist C1 gesetzt, so ist das Zero-Flag gelöscht, andernfalls ist es gesetzt.

TIP

Die Exception-Flags des Statusworts lassen sich auch austesten, allerdings nicht so elegant wie der Condition Code. Nach der Sequenz oben ist nämlich in AL genau das untere Byte des Statusworts vorhanden. Die Anwendung von Bitoperationen wie TEST, AND usw. ermöglicht dann, auch diese Bits zu berücksichtigen.

FWAIT

FWAIT ist eigentlich kein Coprozessorbefehl, sondern ein 8086-Befehl. Dort heißt er WAIT und wurde schon besprochen.

Ein weiterer Meilenstein ist erreicht! Nachdem wir nun die Befehle kennen, mit denen wir Prozessor und Coprozessor programmieren können, haben wir den größten Teil der erforderlichen Arbeit geleistet. Die uns nun bekannten Befehle sind diejenigen, die Sie bei der Assemblerprogrammierung hauptsächlich benutzen werden – unabhängig vom tatsächlich vorhandenen Prozessortyp. Denn wie zu Beginn dieses Teils schon gesagt: Diese Befehle kennen alle Prozessoren x86/x87 von Intel und deren Clones. Die noch fehlenden Befehle der neueren Prozessoren, die wir im übernächsten Kapitel besprechen werden, sind so speziell, daß Sie wohl kaum in Versuchung kommen werden, sie zu nutzen!

5 Zusammenarbeit zwischen 8086 und 8087

Wir wissen nun, wie ein 8086/88 bzw. deren Clones arbeiten, welche Register sie haben und mit welchen Befehlen sie umgehen können. Wir kennen auch das Mathe-Genie 8087 mit seinen Riesenregistern, seine Fähigkeiten, seine Grenzen – und seine Schwächen. Interessant ist nun die Frage: Wie arbeiten die beiden zusammen?

Tonangebend ist der 8086. Er liest die Befehle aus dem Speicher, er adressiert über die Adreßleitungen die Speicherstellen, er koordiniert alles. Der 8087 verfolgt ebenfalls die aktuell abzuarbeitenden Befehle, beginnt aber erst dann mit seiner Arbeit, wenn er einen an ihn adressierten Befehl erkennt.

Woran erkennt er das? Nun – es kommt eigentlich schon in den *Mnemonics* der Befehle, die der 8087 kennt, zum Ausdruck: Sie fangen alle mit »F« für *Float* an. Im Opcode macht sich dies durch bestimmte gesetzte Bits bemerkbar. So beginnt jeder mathematische Befehl mit 11011, was im allgemeinen *ESC* (für *Escape*) genannt wird. Erkennt ein 8086 diese Bitfolge am Beginn eines Opcodes, so weiß er, daß dieser Befehl ihn nichts angeht, er tut dann auch nichts. Da aber der 8087 parallel zum 8086 die aktuell abzuarbeitenden Befehle verfolgt, erkennt auch er den magischen Code des Mathebefehls und führt ihn aus.

Nun gibt es aber mathematische Berechnungen, die für den 8086 eine Ewigkeit dauern. Während dieser Zeit müßte er tatenlos abwarten, bis der 8087 seinen Befehl abgearbeitet hat. Die Entwickler der Chips bei Intel gaben ihm daher die Möglichkeit, unabhängig vom 8087 weiterzuarbeiten, falls dieser beschäftigt ist. Also liest der 8086 nun weitere Befehle ein und führt sie aus, diesmal ohne daß ihm der 8087 zuschaut – der ist ja beschäftigt.

Die beiden Prozessoren sind nun »aus dem Takt«. Der eine erledigt die Arbeit mit den Fließkommazahlen, der andere setzt die Abarbeitung der Befehle fort, ohne daß der erste weiß, welche Befehle – er kann ja nicht zuschauen. Irgendwie müssen die beiden Partner wieder in Gleichklang gebracht werden!

Dies tut der Assembler. Schaut man sich nämlich das Assemblat an, so fällt einem Erstaunliches auf: Vor jedem mathematischen Befehl steht der Code $9B. Dies ist der Code für WAIT. WAIT aber ist die Anweisung für den Prozessor, zu warten, bis der Coprozessor fertig ist. Gut – aber die Anweisung steht *vor* dem ESC-Befehl, der jeden Mathecode einleitet! Dort aber nutzt er nichts, oder?

Im Prinzip darf der Prozessor alles tun, was er will, auch während der Coprozessor beschäftigt ist. Er darf nur eines nicht, sich einen Mathebefehl »anschauen«, wenn der Coprozessor nicht »zuguckt«. Denn den Coprozessor interessieren nur Coprozessorbefehle. Folgen also einem Mathebefehl nur noch 8086-Instruktionen, so braucht der Coprozessor nicht mehr berücksichtigt zu werden und der Prozessor kann unabhängig weiterarbeiten.

Folgt dagegen noch ein Coprozessorbefehl, so steht vor diesem wiederum ein WAIT. Trifft der Prozessor auf diesen Befehl, so gibt es zwei Möglichkeiten: Entweder der Coprozessor ist mittlerweile mit seiner Berechnung fertig geworden, dann kann er wieder beobachten, was der Prozessor liest. Dazu hat er dem Prozessor mittlerweile mitgeteilt, daß er fertig ist, so daß der 8086 den WAIT-Befehl nicht mehr berücksichtigen muß.

Ist der Coprozessor dagegen noch nicht fertig, so hat er dem Prozessor noch nicht mitgeteilt, daß er mitliest. Daher macht der 8086 am WAIT-Befehl **vor** dem nächsten Mathebefehl genau das Richtige: Er wartet auf den 8087.

Sie müssen zugeben, daß das eine geniale Organisation ist: Prozessor und Coprozessor können, obwohl sie voneinander sehr abhängig sind, in weiten Bereichen absolut unabhängig voneinander arbeiten. Auf gleichen Takt gebracht werden sie nur dann, wenn sie zusammenarbeiten müssen, was bei der Abarbeitung von Coprozessorbefehlen der Fall ist.

Wie genial diese Sache wirklich ist, sehen Sie an folgendem Beispiel:

Üblicherweise werden die Coprozessoren 8087 mit 2/3 der Taktfrequenz des Prozessors 8086 betrieben. Nun nehmen wir einen typischen Vertreter für einen Coprozessorbefehl und schauen uns an, wie viele Takte er benötigt (falls Ihnen Takt nichts sagt, schauen Sie bitte in den Anhang). FADD z.B. benötigt ca. 90 Takte, wenn zwei Realzahlen im Stack addiert werden, oder mindestens 150 Takte, wenn eine Zahl aus dem Speicher addiert wird.

Während also der Coprozessor innerhalb von 90 Takten zwei Zahlen addiert, sind, da der Prozessor einen schnelleren Takt besitzt, für diesen das anderthalbfache, also 135 Takte vergangen. Nun dauert aber ein typischer Prozessorbefehl weniger als 10 Takte; ein CMP z.B. schlägt mit 4 Takten zu Buche, ein Vergleich mit anschließendem bedingten Sprung dauert 20 Takte, ein Unterprogrammaufruf einschließlich Rücksprung keine 60 Takte! Das heißt, daß der Prozessor ganze Unterprogramme ausführen kann, während der Coprozessor zwei Realzahlen im Stack addiert.

Hier sehen Sie ein gutes Beispiel dafür, daß die richtige Verwendung des Assemblers tatsächlich eine drastische Reduktion der Bearbeitungszeit bewirken kann. Da ein Hochsprachencompiler nicht analysiert, ob der folgende Befehl ein Coprozessorbefehl ist oder nicht und ob dazwischen andere Prozessorbefehle ausgeführt werden können, reiht er einfach die durch die Hochsprache vorgegebenen Sequenzen aneinander. Dies kann dazu führen, daß der Prozessor bei drei aufeinanderfolgenden Mathebefehlen sehr lange Zeit nur wartet!

Ein Ladebefehl z.B. mit anschließender Multiplikation, Addition und Rückspeicherung dauert für den Prozessor eine Ewigkeit! Stellen Sie sich das in einer Schleife vor, die ein paar tausendmal hintereinander ausgeführt werden muß (z.B. beim Zeichnen einer Linie). Sie als Assemblerprogrammierer können das aber berücksichtigen und optimieren!

ACHTUNG

Ein Problem bei dieser Unabhängigkeit darf allerdings nicht verschwiegen werden: Stellen Sie sich vor, der Coprozessor soll einen

Funktionswert berechnen. Dieser Funktionswert wird dann vom Prozessor dem Video-Chip übermittelt, um an der berechneten Koordinate einen Punkt auf den Bildschirm auszugeben.

Nun passiert folgendes: Der Coprozessor liest den Mathebefehl und beginnt seine Berechnung. Der Prozessor jedoch macht weiter; er liest den nächsten Befehl. Dort jedoch steht, daß er den berechneten Funktionswert dem Video-Chip zur Verfügung stellen soll. Er liest also die entsprechende Speicherstelle aus – schließlich hat das ja nichts mit dem Coprozessor zu tun. Doch der ist mit der Berechnung des Funktionswertes noch gar nicht fertig! Resultat: der Prozessor verwendet Ergebnisse des Coprozessors, die noch gar nicht vorliegen.

Daß das natürlich nicht sein darf, ist klar. Leider kann der Prozessor aber nicht vorhersehen, daß die Speicherstelle, die er auslesen soll, erst vom Coprozessor verändert werden wird. Also sind auch in diesem Fall Sie gefordert! Sie müssen an den entsprechenden Stellen direkt nach den Coprozessorbefehlen, zumindest aber unmittelbar vor kritischen Prozessorbefehlen manuell WAIT-Befehle einfügen, falls die Gefahr besteht, daß der Prozessor Ergebnisse verwendet, die noch nicht berechnet wurden! Hiermit bewirken Sie, daß der Prozessor auf jeden Fall mit der Abarbeitung weiteren Codes wartet, bis der Coprozessor fertig ist.

Zugegeben, das wird selten passieren! Denn meistens können die Codes nicht so optimert werden, daß der Prozessor tatsächlich die ganze Zeit ausnutzen kann, die der Coprozessor benötigt. Aber man sollte an diesen Fall denken, wenn ganz offensichlich Resultate entstehen, die nicht richtig sein können. Vielleicht ist der Prozessor dann schneller mit der Verwertung der Ergebnisse des Coprozessors, als dieser sie liefern kann. Ein WAIT ist hier die Rettung!

Nun verstehen wir auch, was es mit den »N«-Befehlen aus dem letzten Kapitel auf sich hat! Bei diesen *NoWait*-Befehlen wird eben durch den Assembler vor der ESC-Sequenz kein WAIT erzeugt. An seiner Stelle steht einfach ein NOP. Der Hintergrund ist, daß der Coprozessor bei diesen Befehlen nicht auf den Prozessor angewiesen ist, ihn also weiterarbeiten lassen kann.

Doch zurück zum Zusammenspiel von 8086 und 8087. Wie wir nun wissen, liest der 8087 parallel zum 8086 die Befehle, wird aktiv, wenn ein Befehl für ihn bestimmt ist, und bleibt passiv, wenn nicht. Was aber macht der 8087, wenn er Daten lesen oder speichern will? So ganz passiv wie eben geschildert, bleibt der 8086 nicht, wenn ein 8087-Befehl für ihn sichtbar wird. Handelt es sich nämlich um einen Befehl, der den Speicher betrifft, also einen Lade- oder Speicherbefehl, so berechnet der 8086 die Speicheradresse und programmiert den Datenbus so, daß der 8087 das Datum nur noch lesen oder schreiben muß.

Aber nun tritt doch noch ein Problem auf: Der 8087 kann mit Daten bis zu 80 Bit Größe arbeiten, also 10 Byte. Selbst auf den 80386/80486-Rechnern ist aber der Datenbus lediglich 32 Bit breit, d.h. es können nur jeweils 32 Bits gleichzeitig übertragen werden. Beim 8086/80286 sind es nur 16, beim 8088 sogar nur 8 Bits! Also muß ein solches Datum, abhängig von der Breite des Datenbusses, durch bis zu zehnmaliges Adressieren und Lesen bzw. Beschreiben des Datenbusses übertragen werden.

Wo liegt das Problem? Der 8086 »kennt« nur maximal 16 Bit breite Worte. Diese kann er mit einem Mal über den Datenbus auslesen! Ein Befehl – ein Datenbuszugriff. Der 8087 aber braucht mehrere Zugriffe: ein Befehl entspricht hier drei, vier oder fünf Datenbuszugriffen. Wie vielen genau, weiß nur der 8087, es sind schließlich seine Daten, und nur er weiß, welchen Datentyp er gerade zu bearbeiten hat. Die Programmierung des Datenbus aber macht der 8086. Dilemma!?

Nein! Wie wir wissen, hat der 8087 ein eigenes Adreßregister. Er läßt zunächst vom 8086 die Adresse berechnen und den Datenbus vorbereiten. Die eingestellte Adresse liest er dann in sein eigenes Adreßregister, den *Data Pointer*. Für den 8086 ist damit die Sache erledigt. Der 8087 holt sich nun zunächst aus der eingestellten Adresse die ersten 16 (8; 32) Bits. Dann korrigiert er selbständig die Adresse der folgenden Bits in seinem *Data-Pointer-Register* und holt sich alle restlichen Bits durch so viele Datenbuszugriffe/-adressierungen, wie er braucht. Aber Achtung! Die eigentliche, erste Adreßberechnung bleibt auch bei dieser Methode beim 8086! Der 8087 kann nur eine schon berechnete Adresse inkrementieren.

ACHTUNG

Die Opcodes so gut wie aller Coprozessorbefehle bestehen ja, von den »N«-Zwillingen der Befehle abgesehen, aus der Sequenz *Wait – Esc – Operation – Operand(s)*. Das heißt, daß praktisch jeder dieser Befehle damit beginnt, auf die Freigabe durch den Coprozessor zu warten! Was aber macht der Prozessor, wenn gar kein Coprozessor vorhanden ist? Richtig: Er wartet – bis in alle Ewigkeit, da ja niemand dem Prozessor das entsprechende Signal gibt! Also dürfen Coprozessorbefehle nur dann programmiert werden, wenn ein Coprozessor vorhanden ist!

Wie aber kann man das feststellen? Eine Möglichkeit ist, das BIOS zu befragen. Eine andere besteht darin, es selbst festzustellen! Das ist nicht schwer:

TIP

Verwenden Sie einen Mathebefehl, der bestimmte Aktionen nachprüfbar ausführt, kein vorangestelltes WAIT besitzt und nicht zu Problemen führt, wenn kein Coprozessor vorhanden ist. Ein solcher Befehl ist FNINIT. FNINIT initialisiert den Prozessor und belädt ihn mit einem Kontrollwort. Falls also nach einem FNINIT das Kontrollwort einen Wert verschieden von 0 hat, ist ein Coprozessor vorhanden. Andernfalls gibt es niemanden,

der ein solches Kontrollwort initialisieren könnte! Der Befehl wird einfach nicht ausgeführt – der Prozessor überliest ja Coprozessorbefehle. Und da kein WAIT vorangeht, wartet er auch nicht! In Kombination mit FNSTCW, über das der Inhalt des Kontrollworts festgestellt werden kann und das analog zu FNINIT auf das Fehlen eines Coprozessors reagiert (nämlich gar nicht) ist ein einfacher Test auf das Vorhandensein eines 8087 möglich. Wir werden dies im zweiten Teil des Buches nutzen.

Doch noch ein Wort zu den Ausnahmesituationen, die der 8087 kennt. Was eigentlich passiert, wenn eine solche Situation eintritt? Der 8087 bietet mit den Befehlen FENI/FNENI und FDISI/FNDISI die Möglichkeit, das Interrupt-Enable-Flag im Kontrollwort zu verändern. Und bei gelöschtem Flag kann der 8087 einen Interrupt erzeugen, der den Prozessor dazu veranlaßt, seine Aktivität zu unterbrechen und sich der Behandlung der Ausnahme zu widmen. Ein solcher Interrupt gehört zu der Gattung der sogenannten *nicht maskierbaren Interrupts*, kurz NMIs, die nicht vom eigentlich dafür zuständigen *Interrupt-Controller*[3] verwaltet werden und daher nicht abgeschaltet werden können. Man kann sie daher nur verhindern, indem man das Interrupt-Enable-Flag setzt.

6 Änderungen beim 80186/80188

Im Vergleich mit dem 8086 hat sich beim 80186 sehr wenig geändert. Ich weiß nicht, ob Sie überhaupt wissen, daß es einen 80186 tatsächlich gab. Er hat jedoch im PC-Bereich keine große Verbreitung gefunden, da er keine umwerfend neuen Möglichkeiten aufwies und nur wenig mehr bot als sein Vorgänger. Das größte Hindernis war wohl, daß sich der Adreßraum des 80186 gegenüber seinem Vorgänger nicht geändert hatte, er also auch nur 1 MByte Speicher adressieren konnte.

Ich selbst habe auch niemals einen PC in den Händen gehabt, in dem ein 80186 seinen Dienst versah. Allerdings wurden einige Prozessoren dieses Typs in verschiedenen anderen Rechnern zur Emulation eines PCs eingesetzt. So wurde auf einigen Bürosystemen der Firma Xerox z.B. eine Steckkarte mit einem solchen Prozessor eingesetzt, um PC-Software benutzen zu können. Einige Steuerungsautomaten scheinen auch heute noch mit diesem Prozessor bestückt zu sein.

---

[3] Wir werden im zweiten Teil des Buches auf Interrupts zu sprechen kommen und dann auch kurz ansprechen, wie und durch wen sie ausgelöst werden. Bitte nehmen Sie daher diese Information einfach so hin!

Folgende Befehle des 8086 erfuhren eine Veränderung:

- IMUL: Während der 8086 lediglich Inhalte von Registern oder Speicherstellen mit dem AL/AX-Register multiplizieren konnte, wurde ab dem 80186 eine direkte Multiplikation mit einer Konstanten, etwa *IMUL DX, 4711*, zusätzlich eingeführt. Aber wohl gemerkt: nur IMUL, nicht etwa MUL!
- PUSH: Nachdem beim 8086 nur Prozessorregister oder Speicherinhalte auf den Stack gepusht werden konnten, kann man ab dem 80186 auch Konstanten auf den Stack schieben: *PUSH 0815*.
- RCL, RCR, ROL, ROR, SAL, SAR, SHL, SHR: Auch bei diesen Befehlen wurde die Angabe einer Konstanten eingeführt. Während der 8086 lediglich eine Verschiebung von Bits um entweder eine Position oder um die in CL stehende Anzahl durchführen konnte, konnte man ab dem 80186 auch direkt die Anzahl zu verschiebender Stellen angeben. So konnte man ab jetzt z.B. die Befehlssequenz

  ```
  mov   cl,4
  shl   ax,cl
  ```

  durch

  ```
  shl   ax,4
  ```

  ersetzen.

TIP

Mit dieser Veränderung hat sich auch das Verhalten der Befehle gewandelt. Der 80186 verändert den Wert in einem Register bzw. Speicher nicht, wenn z.B. SHR mit CL als Operanden und einem Wert >16 in CL durchgeführt wird. Der 8086/88 jedoch tut dies. Die Sequenz

```
mov       ax,0FFFFh
mov       cl,020h
shl       ax,cl
```

hinterläßt beim 8086/8088 sowie deren Clones V20/V30 eine 0 in AX, wohingegen beim 80186 und den folgenden Prozessoren nichts verändert wird. Auf diese Weise ist es recht einfach möglich, einen Nachfolger des 8086/8088 zu erkennen! Wir werden dies im zweiten Teil des Buches nutzen.

Es wurden jedoch auch neue Befehle eingeführt:

- BOUND: Mit diesem Befehl kann ein Index für ein Feld auf Korrektheit überprüft werden.
- ENTER: Dieser Befehl setzt einen sogenannten Stackrahmen, wie er in Hochsprachen praktisch bei jeder Routine zum Einsatz kommt.
- LEAVE: Mit diesem Befehl kann ein mit ENTER gesetzter Stackrahmen wieder gelöscht werden.

- PUSHA: Mit diesem Befehl werden alle Registerinhalte gerettet:

```
push ax
push cx
push dx
push bx
push sp
push bp
push di
push si
```

- POPA: Die mit PUSHA geretteten Registerinhalte können mit diesem Befehl wieder restauriert werden.
- INS, INSB, INSW: Dieser Befehlssatz ist praktisch eine Erweiterung der Stringbefehle des 8086. Durch diesen Satz kann ein String mit Werten geladen werden, die aus Ports analog zum Befehl IN gelesen werden. In Verbindung mit REP ist auf diese Weise z.B. das Einlesen von Daten aus einer seriellen Schnittstelle äußerst bequem und elegant.
- OUTS, OUTSB, OUTSW: Dies sind die Pendants zu INS, INSB und INSW. Mit ihnen können Daten aus einem String ausgelesen und auf Ports ausgegeben werden.

Wie Sie sehen, waren die Neuerungen zwar durchaus interessant und logisch, jedoch brachten sie lange Zeit nicht viel, denn die Verbreitung des 80186 war nicht sonderlich groß. Schlimmer noch: Bis in unsere Tage schleppt jedes Programm die Bürde der Abwärtskompatibilität mit sich herum. Das heißt nichts anderes, als daß das Programm auf jedem Rechner lauffähig sein muß. Vom 8086 angefangen bis zum Pentium. Und diese Forderung verbietet ganz offensichtlich den Einsatz von Befehlen, die der 8086 nicht kennt. Übrigens: Auch den 80186 gibt es in einer »8-Bit-Datenbus«-Version, die dann folgerichtig auch 80188 heißt. Doch gilt auch hier das, was schon beim 8086/8088-Gespann gesagt wurde: Die Unterschiede brauchen uns nicht zu interessieren; sie sind rein verwaltungstechnischer Art und computerintern!

7 Der 80187

Einen 80187 hat es nie gegeben. Wenn man die Prozessoren 8086 und 80186 vergleicht, so machte dies auch wenig Sinn. Der 80186 konnte mit keinen wesentlichen Neuerungen gegenüber seinem Vorgänger aufwarten. Da vermutlich auch bei Intel dieser Prozessor nur als Eintagsfliege mit bestimmten Einsatzgebieten gedacht war, verzichtete man auf die Herstellung eines korrespondierenden Coprozessors.

8 Änderungen beim 80286/80287

Die Änderungen beim 80286 lassen sich im Prinzip auf zwei gravierende Unterschiede dieses Prozessors zum 8086/80186 zurückführen: die Einführung eines neuen Betriebsmodus, des sogenannten *Protected-Mode*, und die Erhöhung der Adreßleitungen auf 24.

Fangen wir mit dem zweiten Punkt an. Während der 8086/88 und sein Nachfolger 80186/88 noch über 20 Leitungen verfügten, mit denen er seinen Adreßraum ansprechen konnte, so kamen beim 80286 also vier neue Leitungen hinzu. Somit konnten nun statt der ursprünglichen 2^{20} = 1.048.576 = 1.024 k = 1 M Speicherstellen 2^{24} = 16.777.216 = 16.384 k = 16 M adressiert werden, also genau das 2^4- oder 16fache.

Dies war zunächst ein Fortschritt. Doch wozu das Ganze? DOS hat eine Speichergrenze, die bei 640 kByte liegt und nicht überschritten werden kann. Denn DOS wurde entwickelt, als der 8086 und dessen technische Möglichkeiten aktuell waren. Dies bedeutet: maximal 1 MByte Speicher, den man ansprechen konnte. Somit war DOS auf diese Grenze hin ausgerichtet, aber auch festgelegt.

Falls man nun nicht den neuen zur Verfügung stehenden, größeren Adreßraum un- oder nur halbherzig genutzt lassen wollte, mußte der neue Prozessor über neue Arten der Speicherverwaltung verfügen. Und dies geschah mit einem neuen Betriebsmodus, dem Protected-Mode. In diesem Modus kann der 80286 seine vollen Möglichkeiten ausschöpfen. Natürlich waren hierzu neue Prozessorbefehle notwendig, die den Befehlssatz des 8086/88 ergänzen mußten. Klar auch, daß dann ein neues Betriebssystem hierfür notwendig wurde, das die Befehle – und den großen Adreßraum – nutzen konnte. Es wurde dann auch eingeführt – wenn auch erst lange nachdem der 80286 das Licht der Welt erblickt hat: OS/2 und *Windows*.

Es gibt noch einen weiteren Aspekt. Der Protected-Mode hat seinen Namen daher, daß er gewisse Speicherbereiche »schützt« – und zwar vor Zugriff von Programmteilen oder Programmen, die dort nichts zu suchen haben. Möglich wird auch dies durch *Deskriptoren*, in denen bestimmte Flags gesetzt werden können, die den Zugriff kontrollieren. Dies bedeutet, daß z.B. ein unsauber programmiertes Programm nicht mehr so einfach das ganze System zum Absturz bringen kann, nur weil es (beabsichtigt oder versehentlich) auf eine verbotene Speicherstelle zugreift.

Warum gibt es aber dann noch den Real-Mode beim 80286, wenn der Protected-Mode doch so viel leistungsfähiger war und ist? Auch hier gilt wiederum das Schlagwort: »Abwärtskompatibilität«! Protected-Mode

und Real-Mode sind miteinander unvereinbar! Das liegt einfach daran, wie der Speicher adressiert wird bzw. wie der Adreßraum genutzt werden kann. Beim 8086 haben wir es einfach: Durch die Segmentierung des Speichers kann der gesamte Adreßraum von 1 MByte adressiert werden. Die Segmentregister enthalten hierzu, wie Sie sich erinnern werden, die Segmentgrenze, an der ein Segment beginnt. Die Indexregister (»Zeigerregister«) hingegen können mit ihrem Inhalt auf ein Byte innerhalb (und, wie wir gesehen haben, gegebenenfalls sogar außerhalb) dieses Segments weisen.

Wollte man nun mit den zusätzlichen vier Adreßleitungen, über die der 80286 verfügt, ähnliches erreichen, so müßte man anders segmentieren. Denn es müßten nun die 16 MByte, die der 80286 adressieren kann, durch 65.536 dividiert werden, um die Segmentgrenzen zu bestimmen. Das hieße, daß die Segmentgrenzen nicht wie beim 8086 bei Vielfachen von 16 liegen, sondern bei Vielfachen von 16·16, also von 256 – sogenannten *Pages*! Dadurch ließe sich zwar der gesamte Adreßraum erreichen, aber die Segmente könnten nicht mehr »so fein« überlagert werden. Diese »gröbere Körnung« würde also nicht nur eine neue Art der Segmentbenutzung erfordern, sondern auch zu Inkompatibilitäten mit den Befehlen des 8086 bzw. bestehender 8086-Programme führen.

Also hieße dies in jedem Fall, daß neue Befehle und ein neuer Modus geschaffen werden müssen. Nun hatte man in der Zwischenzeit aus den gemachten Fehlern gelernt. Man sagte sich, daß es nicht ausgeschlossen sei, daß eine neue Prozessorgeneration über noch mehr Adreßleitungen verfügen könnte, was dann mit dem 80386 ja auch geschah. Also entschloß man sich, gleich ein vollständig neues Konzept der Speicheradressierung zu entwickeln.

Diese Konzept sieht folgendes vor: In den Segmentregistern, in denen bisher nur die Segmentadressen standen, stehen jetzt Adressen von sogenannten D*eskriptoren*. Diese »Beschreiber« sind nun einfach Datenstrukturen, in denen jede Menge an Informationen stehen, unter anderem auch eine Adresse, an der das zum jeweiligen *Deskriptor* gehörende »Segment« steht (Der Begriff scheint hier nicht mehr richtig zu sein, da ja nicht mehr segmentiert wird – zumindest nicht im Sinne des 8086! Dennoch hat er sich gehalten, um anzudeuten, daß in diesem Segment bestimmte zueinander gehörende Informationen stehen, wie z.B. der Code von »Units« oder Daten). Das aber heißt, vereinfacht ausgedrückt, daß der Prozessor nun, bevor er auf ein Segment zugreifen kann, erst die Adresse des betreffenden Segmentd*eskriptors* auslesen und im *Deskriptor* dann nach der erforderlichen Segmentadresse suchen muß.

Sie sehen: Durch die Neuerungen und Speichererweiterungen haben sich nicht nur Horizonte geöffnet! Es sind auch neue Probleme entstanden. Auch – und vielleicht sogar gerade – für Programmierer.

Beschränken wir die Vorstellung der neuen Befehle auf zwei wesentliche Aspekte: Änderungen und Ergänzungen der Befehle des Real-Mode, die wir ausführlich diskutieren werden, und ein kurzes Anreißen der Befehle des Protected-Mode, die wir im gesamten Buch nicht nutzen werden. Falls Sie nun enttäuscht sind, folgender Trost: Die Protected-Mode-Befehle sind im Prinzip nur dazu da, Betriebssysteme zu entwickeln, die den riesigen Adreßraum nutzen können und die ein Multitasking ermöglichen sollen. Natürlich sind auch im Protected-Mode die (meisten) Befehle des Real-Mode verwendbar, so daß Sie auch Programmteile entwickeln können, die in Programmen unter OS/2 oder Windows einsetzbar sind. Denn schließlich habe ich ja schon im Vorwort geschrieben: Kein ernsthafter Mensch käme heute auf die Idee, ein Programm für Windows, OS/2 oder andere moderne Betriebssysteme vollständig in Assembler zu schreiben! Dieses Buch dient dazu, Ihnen den Umgang mit Assembler näherzubringen, um Programme zu optimieren, die in Hochsprachen entwickelt werden –, oder ganz kleine Assemblerprogramme für den Real-Mode unter DOS! Haben Sie weitergehende Interessen, so sei auf weiterführende Literatur verwiesen.

Es folgt nun die Auflistung der neuen Befehle des 80286:

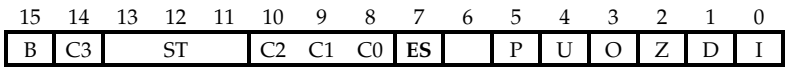
- ARPL; *Adjust RPL Field*. Dieser Befehl ist nur im Protected-Mode verwendbar und korrigiert das RPL-Feld im *Selector*.
- CLTS; *Clear Task Switch Flag*. Neben den Flags des 8086 verfügt der 80286 noch über weitere Flags, die im Protected-Mode genutzt werden. Das *Task-Switch-Flag* ist ein Beispiel dafür, das mit CLTS gelöscht werden kann.
- LAR; *Load AR*. Mit LAR wird das sogenannte AR-Byte geladen, das die Zugriffsrechte im Protected-Mode regelt.
- LGDT, LIDT, SGDT, SIDT; *Load Global Descriptor Table*, *Load Interrupt Desriptor Table*, *Store Global Descriptor Table* und *Store Interrupt Descriptor Table*. Diese Befehle laden oder speichern bestimmte Deskriptortabellen, sind also nur im Protected-Mode verfügbar und interessant.
- LLDT, SLDT; *Load Local Descriptor Table* und *Store Local Desriptor Table*. Wie bei LGDT etc. lädt und speichert dieses Befehlspaar eine weitere Deskriptortabelle.
- LMSW, SMSW; *Load Machine Word*, *Store Machine Word*. Diese Befehle laden und sichern das sogenannte Maschinenwort, ein Statuswort, das nur im Protected-Mode benötigt wird.
- LSL; *Load Segment Limit*. Im Protected-Mode kann mit diesem Befehl eine Segmentgrenze geladen werden. Vorsicht! Dies weicht etwas von den Segmentgrenzen im Real-Mode ab.

- LTR, STR; *Load Task Register*, *Store Task Register*. Das *Task-Register* ist ein Register, das bei der Verwaltung der verschiedenen *Tasks*, also der Programme eines Multitasking-Betriebssystems, benötigt wird. Es setzt den Protected-Mode voraus. Es wird ausschließlich von Betriebssystemen verwendet.
- VERR, VERW, *Verify Read*, *Verify Write*. Diese Befehle stellen fest, ob in einem bestimmten Segment gelesen oder geschrieben werden darf. Dies wird im Protected-Mode benötigt (gerade weil in diesem Modus bestimmte Zugriffsrechte für Segmente vergeben werden können, die ein Beschreiben bzw. Auslesen verhindern, heißt dieser Betriebsmodus ja Protected-Mode, also »geschützter Modus«).

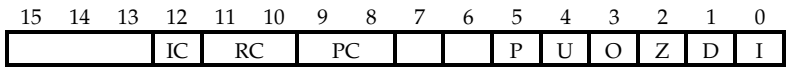
Beim 80287 sind nur zwei neue Befehle hinzugekommen:

- FSETPM, *Set* Protected-Mode. Da der 80286 im Protected-Mode arbeiten kann, muß sein Coprozessor dies auch können. Dieser Befehl schaltet einen 80287 in den Protected-Mode.
- FSTSW AX, FNSTSW AX, *Store Statusword In AX*. Dieser Befehl kopiert das Statuswort direkt in das AX-Register. Wie Sie sich erinnern werden, konnte mit der Sequenz *FSTSW WordVar – MOV AX,WortVar – SAHF* das obere Byte des Statuswortes des 8087 mit dem Condition Code in das Flagregister geladen werden. Man hat es nicht ganz geschafft, mit einem Befehl den Inhalt direkt aus dem Statuswort in das Flagregister zu kopieren. Immerhin ist es aber ganz schön, daß es nicht mehr über eine Variable im Speicher erfolgen muß und somit schneller geht! Auch hier gibt es eine »N«-Variante, die kein WAIT voranstellt.

Es sollte vielleicht noch erwähnt werden, daß sich auch im Kontrollwort und im Statuswort etwas geändert hat. Da sich die Art der Ausnahmebehandlung aufgrund der (wie wir noch sehen werden) anders gearteten Zusammenarbeit zwischen Prozessor und Coprozessor geändert hat, mußten hier Änderungen vorgenommen werden:



Das Bit 7 des Statuswort zeigt nun nicht mehr an, daß ein Interrupt aufgrund einer Ausnahmebedingung stattgefunden hat. Vielmehr spiegelt es jetzt wider, ob irgendeine Exception aufgetreten ist. Es heißt somit ab jetzt Exception-Summary-Flag ES. Praktisch stellt es die logische Oder-Verknüpfung der Bits 5 bis 0 dar: Ist eines oder mehrere dieser Flags gesetzt, so ist es ES auch, während es nur gelöscht ist, wenn alle anderen es auch sind.



Da keine Interrupts mehr freigeschaltet werden (können), gibt es auch für das *Interrupt-Enable-Mask*-Bit 7 (IE) im Kontrollwort keine Existenzberechtigung mehr: Es wurde entfernt!

9 Zusammenarbeit zwischen 80286 und 80287

Der 80286 hat verglichen mit dem 8086/8088, aber auch im Vergleich mit dem 80186 erheblich mehr Möglichkeiten aufzubieten. Der Protected-Mode ist eine davon, eine andere die Fähigkeit, virtuellen Speicher zu benutzen. Das bedeutet, der 80286 kann Speicher verwalten, der gar nicht existiert – genauer gesagt, Speicher, dessen Adresse real nicht existiert, sondern irgendwie »abgebildet« werden muß. Wie, das weiß nur der Prozessor (und einige sehr schlaue Programmierer!).

Dies muß aber Konsequenzen haben! Denn wir haben ja gesehen, daß der 8086 in gewissen Grenzen mit dem 8087 parallel zusammenarbeiten kann und letzterer bei Adressierungen lediglich die Startadresse z.B. einer Realzahl vom Prozessor berechnen lassen muß – alle weiteren Zugriffe erledigt der 8087 allein. Das geht aber nicht mehr, wenn der 80286 vituellen Speicher verwaltet!

Der 80287 hat aus diesem Grunde die Eigenschaften des 8087 zur Zusammenarbeit praktisch vollständig verloren. Er ist eigentlich kein »Co-«Prozessor mehr, sondern nur noch ein Peripheriegerät des Prozessors wie die Bausteine zur parallelen oder seriellen Datenübertragung auch, erschaffen, um nur noch auf die Anweisungen des Prozessors zu warten. Daher erfolgt hier auch die Zusammenarbeit zwischen dem 80286 und seinem Coprozessor anders. Der 80287 wird wie jeder andere Peripheriebaustein auch vom Prozessor über Portadressen angesprochen. Zusätzlich verfügt er über mehrere Steuerleitungen, über die er mit dem Prozessor kommunizieren kann.

Das bedeutet aber, daß der Prozessor den Coprozessor mit Daten versorgt und diese auch wieder abholt! Somit arbeiten beide nicht mehr unabhängig voneinander und eine Synchronisation über WAIT ist nicht mehr erforderlich. Der Assembler berücksichtigt dies, indem er die WAIT-Befehle nicht mehr voranstellt – falls man ihm die Anweisung dazu gibt. Wie, das klären wir im zweiten Teil des Buches. Der Code wird dadurch kompakter und schneller. Dennoch ist der 80287 vollständig abwärtskompatibel und kann daher auch 8087-Programme mit all den WAITs verarbeiten!

Was passiert nun in Ausnahmesituationen, also nach Operationen, die ein Exception-Flag gesetzt haben? Beim 8087 wurde noch ein NMI, ein nicht maskierbarer Interrupt, erzeugt. Beim 80287 jedoch erfolgt dies ein wenig anders. Der 80287 verfügt über eine Leitung, die direkt mit dem Prozessor verbunden sein sollte und die nur zu dem Zweck dient, dem Prozessor die Ausnahmesituation mitzuteilen. »Sein sollte« heißt aber, daß sie es nicht ist! Vielmehr liegt sie an einem Eingang eines *Interrupt-Controllers*.

Der Grund dafür ist, daß sich IBM bei der Entwicklung des ATs nicht an die Vorgaben des Prozessorherstellers Intel gehalten hat und eigene Wege gegangen ist, die nicht kompatibel sind. Und, so seltsam es klingen mag: aus Kompatibilitätsgründen haben sich alle anderen PC-Hersteller ebenfalls über Intels Vorgaben hinweggesetzt. Daher mußte nun mit viel Aufwand eine andere Lösung her – über den *Interrupt-Controller*!

Der Interrupt-Controller teilt dem Prozessor nun mit, daß einer seiner Peripheriebausteine, in diesem Fall der Coprozessor, Aufmerksamkeit benötigt. Der Prozessor reagiert darauf mit einem Interrupt, dem INT $75. Dieser Interrupt ist aber im Gegensatz zum NMI des 8087 durch den Interrupt-Controller maskierbar, so daß die Notwendigkeit, ihn im Coprozessor selbst zu unterbinden, entfiel. Halten wir daher fest, daß die Befehle FENI/FNENI und FDISI/FNDIS ab dem 80287 keine Bedeutung und somit keine Wirkung mehr haben, obwohl sie noch aufgrund der Kompatibilität verfügbar sind. Sie werden einfach ignoriert, funktionieren also wie ein WAIT-Befehl.

10 Änderungen beim 80386/80387

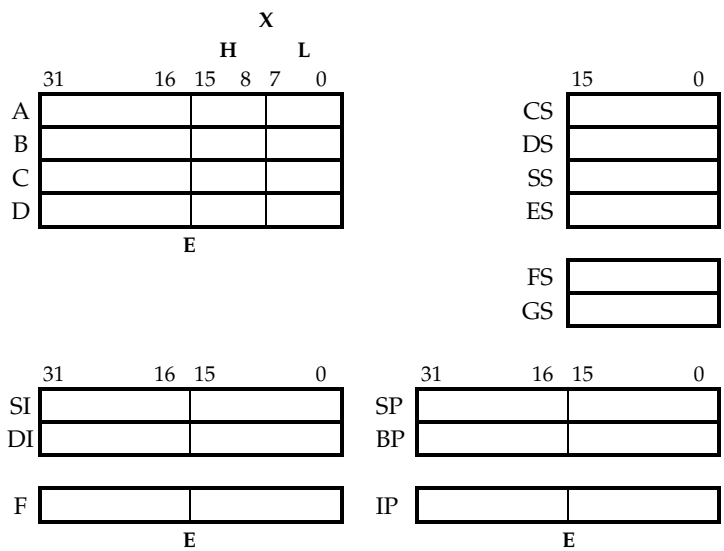
Die Änderungen beim 80386 lassen sich auf verschiedene Gründe zurückführen. Zum einen erhielt der 80386 einen 32-Bit-Adreßbus. Wenn das Ergänzen der 20 Adreßleitungen des 8086 um vier beim 80286 zu einer Vergrößerung des adressierbaren Speichers auf das 16fache führte, um wieviel wird dann der nutzbare Speicher größer, wenn noch einmal 8 Bits, sprich acht Adreßleitungen hinzukommen? Noch einmal um das 256fache! Das ist genau das 4.096fache des Speichers, den ein 8086 ansprechen kann! Mit anderen Worten: 4 GByte RAM!

Diese tatsächlich phänomenale Erweiterung des Adreßraums machte natürlich eine vollkommen neue Prozessorstruktur notwendig. Deshalb stellte man den 80386 auch intern gleich in einem zweiten Punkt um: Dieser Prozessor hat auch 32 Datenleitungen, also genau die doppelte Anzahl, die ein 8086, 80186 oder ein 80286 besitzt.

Das aber hat Konsequenzen! Denn wenn über 32 Datenleitungen parallel 32 Bits übertragen werden können, müssen auch die Register des 80386 32 Bit breit sein.

Aber nicht bei allen 386ern kann dies so ausnahmslos behauptet werden. So produzierte Intel als billige »Einsteigerversion« für den damals noch sehr teuren 80386 eine *Slim-line*-Version. Es handelte sich intern um einen vollständigen 80386er, der jedoch in seinen Kontakten zur Außenwelt kräftig beschnitten worden war: Statt 32 Datenleitungen erhielt dieser 80386**SX**-Prozessor nur 16 und statt 32 Adreßleitungen nur 24. Damit kann man ihn in Hinblick auf seine Adressierbarkeit als 80286er auffassen, der jedoch über alle erweiterten Befehle und Modi des »richtigen« 80386**DX** verfügt.

Ansonsten hat sich im Real-Mode des Prozessors nicht viel geändert:



Verglichen mit dem Registersatz des 8086/80186/80286 fällt uns zunächst auf, daß zwei neue Register hinzugekommen sind. Sie heißen FS und GS und sind, wie ES auch, global verfügbare Segmentregister, die keine besondere Bedeutung haben, die man aber zur Angabe von Adressen nutzen kann. Aus diesem Grunde wurden sie auch wenig einfallsreich mit *F* und *G* durchnumeriert. Sie sind, wie die anderen Segmentregister, auch weiterhin nur 16 Bits breit.

Ferner fällt auf, daß alle anderen Register 32 Bit breit sind. Dennoch lassen sie sich alle wie die ganz normalen 16-Bit-Register ansprechen. Das von diesen Prozessoren gewohnte Ansprechen ist sogar Standard:

MOV AX, CX kopiert die Bits 15 bis 0 aus dem *C*-Register in die Bits 15 bis 0 des *A*-Registers. Die Bits 31 bis 16 bleiben jeweils unberührt. Auch wenn nur die 8-Bit-Anteile verwendet werden sollen, bleibt alles beim alten: *CMP DL, BH* vergleicht die Bits 7 bis 0 des *D*-Registers mit den Bits 15 bis 8 des *B*-Registers.

Soll das ganze 32-Bit-Register angesprochen werden, so muß wiederum ein Buchstabe bemüht werden, wie wir dies ja von den »Rechenregistern« her schon gewohnt sind. Ein nachgestelltes *X* bedeutete das ganze 16-Bit-Register, *H* stand für die Bits 15 bis 8 und *L* für die Bits 7 bis 0. Zur Benennung des 32-Bit-Registers dient der Buchstabe *E*, der diesmal jedoch vorangestellt wird: **E**AX ist das *Extended*-AX-Register, also die Bits 31 bis 0 des *A*-Registers. Analoges gilt für **E**BX, **E**CX und **E**DX. Die Buchstabenkombinationen EAH, EBL etc. sind verboten; es gibt sie nicht. Das wäre aber auch recht sinnlos, da damit ja die Bits 31 bis 16 und 15 bis 8 bzw. 7 bis 0 angesprochen würden, also 24 Bits.

Auch die *Indexregister*, das *Flagregister* und die *Pointerregister*, also SI, DI, F, BP, SP und IP haben eine *E*-Erwetierung erfahren. So muß man **E**SI verwenden, falls alle 32 Bits des *Source-Indexregisters* gemeint sind, oder **E**BP, falls das auf den *Base-Pointer* zutrifft. Doch auch beim 80386 hat man keinen direkten Zugriff auf den *Instruction-Pointer*; weder in der 16-Bit-Version IP noch in seiner 32-Bit-Erweiterung **E**IP.

ACHTUNG Leider läßt sich nicht, analog zu AH, BH, CH und DH, das jeweils »obere« Wort eines Doppelwortes, wie die 32-Bit-Daten auch heißen, ansprechen. Auch wird es nicht nochmals in Bytes aufgeteilt. Bei den Rechenregistern ist jeweils nur das gesamte 32-Bit-Doppelwort ansprechbar (z.B. EAX), das »untere« Wort (AX) oder das »obere« (AH) oder »untere« (AL) Byte des »unteren« Wortes.

Sollen also alle 32 Bits des *A*-Registers mit dem Register *D* verglichen werden, so heißt der Befehl *CMP EAX, EDX*. Ebenso ist das Übertragen von Doppelworten anzugeben: *MOV EBX, DWordVar* lädt den Inhalt aus dem Doppelwort *DWordVar* in das Register *Extended BX* (und überschreibt dessen bisherigen Inhalt).

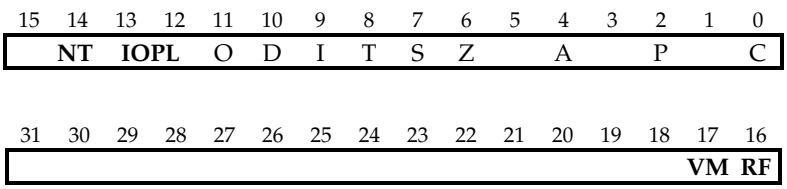
HINWEIS Wir stellen also fest, daß eine Änderung der Befehle des 8086 stattgefunden hat! Da der 80386 über 32 Bit breite Register verfügt, ist jeder der 8086-Befehle wie bisher mit 16-Bit-Daten, aber auch mit 32-Bit-Daten durchführbar. Die Befehle selbst haben identische Namen. Die Unterscheidung erfolgt nur durch das Voranstellen des *E* vor den Namen des entsprechenden 16-Bit-Registers.

ACHTUNG Die Erweiterung von Adreß- und Datenbus auf der einen und der Indexregister auf der anderen Seite auf 32 Bit hat noch andere Auswirkungen. Während noch beim 80286 der verfügbare Speicher nur

über einen Trick angesprochen werden konnte, weil die 16-Bit-Register nicht mit 24-Bit-Adressen arbeiten konnten, ist nun eine direkte Adressierung möglich. Da sowohl der Adreßbus als auch die Register 32 Bit breit sind, kann durch diese Register ein direkter Zugriff auf alle verfügbaren Speicherstellen erfolgen. Man nennt dies »lineare Adressierung«, weil hier nicht mit Segmenten gearbeitet werden muß. Falls Ihnen also einmal der Begriff *Flat-Model* (= »flaches« Modell, bei dem »echte« 32-Bit-Adressen verwendet werden) begegnet, so wissen Sie von nun an, was das heißt: Es handelt sich um ein Programmiermodell, in dem eine lineare Adressierung gewählt wird, die ohne Segmente auskommt. Sie ist erst ab den 80386-Prozessoren möglich.

Doch die Auswirkungen gehen weiter! Ganz analog kann man natürlich die 16-Bit-Segmentregister in die Adressenberechnung miteinbeziehen. Läßt man die veraltete Segmentierung außer acht, summiert sich dies zu einer 48-Bit-Adresse, wie sie als Operand in den Befehlen tatsächlich verwendet werden kann. Dies erhöht theoretisch den Adreßraum des 80386 auf 262.144 GByte, also das 65.536fache des physikalisch über 32 Leitungen ansprechbaren Adreßraums. Sie können also mehr Speicher ansprechen, als physikalisch verfügbar ist, weshalb man hierzu auch analog zu den Verhältnissen beim 80286 (20 Leitungen – 32-Bit-Adressen!) von *virtueller* Speicheradressierung spricht[4].

Dies sind wohl die wesentlichsten Änderungen, zumindest, was uns interessieren muß. Es existieren jedoch noch viele andere Veränderungen oder Ergänzungen, z.B. neue Flags (in **E**Flag). Dies bezieht sich nicht nur auf eine Auffüllung der noch freien Bits im Flagregister! Denn da auch dieses nun 32 Bits breit ist, können weitere Flags berücksichtigt werden:



[4] Womit auch die nähere Zukunft nicht mehr ganz so nebulös ist, zumindest im Hinblick auf Adressen. Denn die logische Konsequenz für einen der Nachfolger des Pentium dürften wohl 32 Bit breite Segmentregister sein und 64 Adreßleitungen. Aber von den resultierenden gigantischen Speichergrößen wage ich derzeit nicht einmal zu träumen! 2^{64} – das ist das 2^{16}fache dessen, was bisher zumindest virtuell angesprochen werden kann…

Die neuen Flags dienen zur Steuerung der IO-Privileg Levels und der Nested Tasks im Protected-Mode. Resume Flag und Virtual Mode Flag steuern das Umschalten in den neuen virtuellen 8086-Modus. In diesem Modus wird der Adreßraum des 80386 in Scheibchen von 1 MByte eingeteilt. Diese Speicherscheiben sind genau so groß wie der Adreßraum eines 8086. Das bedeutet, daß der 80386 so tun kann, als ob er mehrere 8086 gleichzeitig sei! Für uns heißt das, daß wir uns um diesen Modus nicht kümmern brauchen: Wir tun einfach ebenfalls so, als programmierten wir nur für den 8086. Den Rest überlassen wir einfach dem Betriebssystem und dem Prozessor. Es gibt außerdem noch verschiedene Spezialregister, die jedoch im Rahmen dieses Buches nicht beschrieben werden können.

Auch beim 80386 gibt es neue Befehle:

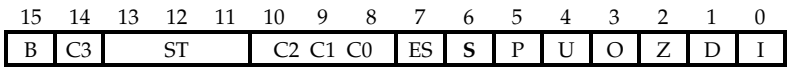
- BSF, BSR, BT, BTC, BTR, BTS, SHLD, SHRD. Hierbei handelt es sich um Befehle, mit denen einzelne Bits manipuliert werden können. Wir werden noch genauer darauf eingehen.
- CDQ, CWDE. Was CBW für Bytes und CWD für Worte ist, ist CDQ für Doppelworte. CWDE ist eine andere Methode, Worte zu konvertieren. Auch diese Befehle werden wir noch genauer erklären.
- CMPSD, INSD, LODSD, MOVSD OUTSD, SCASD, STOSD. Die 32-Bit-Erweiterung macht es selbstverständlich auch notwendig, die Stringbefehle des 8086 entsprechend zu ergänzen. So existieren alle diese Befehle in einer Form, in der auch Doppelwörter in Strings manipuliert werden können. Die Indexregister werden dann konsequent um vier erhöht oder verringert.
- IRETD ist eine Verfeinerung der Möglichkeiten des 80286. Dieser Prozessor konnte als erster durch die Einführung des Protected-Mode unterschiedliche Tasks quasi parallel ausführen. Dem wird nun dadurch Rechnung getragen, daß der Rücksprung aus Interruptroutinen auf unterschiedliche Weise erfolgen kann.
- JMP sowie die bedingten Sprünge können nun auch 16- bzw. 32-Bit-Distanzen zurücklegen. Das alles sind Auswirkungen der Erweiterung der Register.
- LGS, LSS, LFS. Diese Befehle entsprechen den Befehlen LDS und LES beim 8086 und übergeben den zusätzlichen Registern GS und FS das Segment einer Adresse. Natürlich muß auch bei diesen ein Operand ein Register sein, das den dazugehörigen Offset aufnimmt, also z.B. *LFS EBX, Var* oder *LGS CX, Var*. Neu ist auch, daß das *Stacksegmentregister* direkt beladen werden kann; es ist ein Tribut an die Tatsache, daß in *Multitasking*-Umgebungen eben jede Task ihren eigenen Stack hat, das korrespondierende Segmentregister also einfach manipulierbar sein sollte.

- MOV. Dieser Befehl hat Ergänzungen erhalten, da mit ihm auch die weiter oben angesprochenen Spezialregister manipuliert werden sollen. Uns interessiert nur, daß sonst allles beim alten bleibt!
- MOVSX, MOVZX. Diese Befehle sind eine Kombination aus MOV und CBW. Das heißt, daß MOVSX, MOV *with Signed Expansion*, ein Byte aus einer Speicherstelle oder einem Register liest und dieses Byte, vorzeichenbehaftet interpretiert, auf Wortgröße erweitert. Abgelegt werden kann es nur in einem 16-Bit-Register, nicht aber in einer Speicherstelle. MOVZX, MOV *with Zero Expansion*, tut das gleiche, interpretiert aber das Byte vorzeichenlos und erweitert es daher auch nur vorzeichenlos auf ein Wort.
- PUSHAD, POPAD, PUSHFD, POPFD. PUSHAD und POPAD tragen der Tatsache Rechnung, daß der 80386 über 32-Bit-Register verfügt. Während nämlich PUSHA/POPA nur die gesamten 16-Bit-Register des 80186/80286 auf den Stack schieben bzw. sie vom Stack holen, mußten für die 32-Bit-Pendants die erweiterten Versionen »D« (für DoubleWord) geschaffen werden. Gleiches gilt für das Flagregister. Soll also das EFlagregister auf den Stack gebracht bzw. von ihm geholt werden, so ist PUSHFD/POPFD zu benutzen.
- SETcc. Dies ist ein sehr interessanter Befehl. Er arbeitet praktisch wie ein Jcc-Befehl, also wie ein bedingter Sprung. Doch ist die Auswirkung eine gänzlich andere. Während bei einem bedingten Sprung in Abhängigkeit von den gesetzten Flags die Verzweigung des Programms erfolgt, so wird durch SETcc in Abhängigkeit von den Flags ein bestimmtes Byte auf 1 gesetzt, sonst auf 0! Das Programm wird also nicht verzweigt, es macht in jedem Fall mit dem nächsten Befehl weiter. Lediglich der Inhalt der Byte-Variablen oder des (8-Bit)-Registers, das als Operand übergeben werden muß, hat unterschiedliche Inhalte. Wenn Sie nun wissen wollen, welche Bedingungen geprüft werden, so schauen Sie bitte unter dem Abschnitt »Bedingte Sprünge« im Kapitel über die Befehle des 8086 nach!

Betrachten wir nun zuerst den 80387, bevor wir einige Befehle genauer erklären. Zunächst: FSETPM, mit dem der 80287 in den Protected-Mode geschaltet werden konnte, existiert nicht mehr. Der 80387 betrachtet dies einfach als NOP und übergeht es. Der Hintergrund: Da der 80386 nicht nur über den Protected-Mode, sondern auch über den Virtual-8086-Mode verfügt, müßte auch der 80387 über diese drei Modi verfügen. Die Entwickler übergaben daher die Verantwortung für die Berechnung korrekter Adressen vollständig dem 80386. Somit entfiel die Notwendigkeit, dem 80387 ebenfalls den Protected und den Virtual-8086-Mode zu verpassen und via SETPM in einen der beiden schalten zu können.

Ferner gibt es beim 80387 keinen Unterschied mehr zwischen affinem und projektivem Model der Unendlichkeiten. Er unterscheidet immer, unabhängig von den gesetzten Bits, zwischen -∞ und +∞! Dies kann bei Routinen, die dies nutzen, zu unterschiedlichen Ergebnissen auf 80386/30387- bzw. 80286/80287-Rechnern führen.

Es hat sich auch etwas beim Statuswort geändert:



Das Statuswortregister besitzt ein weiteres Bit: Bit 6 oder S. Dieses Bit zeigt eine *Invalid Operation Exception*, falls der Stack überläuft, also versucht wird, einen Wert in die Rechenregister zu laden, obwohl diese alle belegt sind (Sie erinnern sich an die Codierung mit NaNs beim 8087, wenn z.B. mit FLD ein Wert in das besetzte ST(7)-Register geladen werden sollte). Ebenso erzeugt das Poppen eines nicht besetzten, also als *empty* markierten Registers, einen Stackunterlauf (komischer Ausdruck!) und ebenfalls das Setzen von S.

Auch beim Coprozessor wurden nur wenige neue Befehle implementiert:

- FCOS, FSIN, FSINCOS. Die lange vermißten Berechnungen für Sinus und Cosinus sind endlich verfügbar. Sie können entweder einzeln mit FSIN und FCOS oder in einem Arbeitsgang mit FSINCOS berechnet werden.
- FPREM1. FPREM beim 8087 führt eine Modulodivision durch, die nicht den Regeln eines Normungsausschusses folgte. Dies wurde mit FPREM1 geändert: Hier erfolgt die Modulodivision gemäß IEEE.
- FUCOM, FUCOMP und FUCOMPP. Diese Befehle führen einen *ungeordneten Vergleich* durch, arbeiten aber ansonsten analog zu FCOM/FCOMP/FCOMPP.

Doch nun zu den wenigen Details:

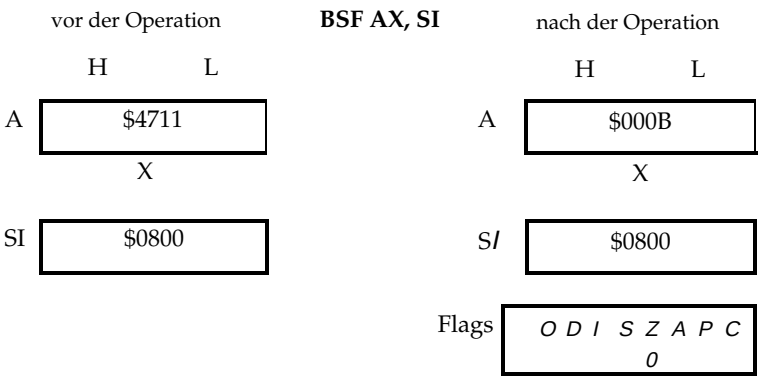
BSF

BSF, *Bit Search Forward*, besitzt zwei Operanden, die entweder 16 oder 32 Bits breit sein können:

- BSF reg, reg z.B. BSF AX, BX
  BSF EAX, ESI
- BSF reg, mem z.B. BSF CX,WordVar
  BSF AH, DWordVar

Im Quelloperanden, der nach Intel-Konvention immer der zuletzt genannte ist, steht die Bitfolge, die untersucht werden soll. Die Suche

nach einem gesetzten Bit beginnt bei BSF bei Bit 0 und erfolgt in Richtung zunehmender Bitnummern. Wird kein gesetztes Bit gefunden, weil der Inhalt der Quelle 0 ist, so wird das Zero-Flag gesetzt. Andernfalls wird es gelöscht und die Bitnummer des ersten gesetzten Bits im Zieloperanden übergeben:



$B als Ergebnis in AX signalisiert, daß Bit 11 im Wert von SI das erste gesetzte Bit ist, wenn die Suche bei Bit 0 beginnt.

ACHTUNG

Es werden auch die anderen Flags nach BSF verändert. Laut Intel-Dokumentation ist der Zustand *undefiniert*, was heißt, daß man die Flagstellung nicht berücksichtigen sollte.

BSR

BSR, *Bit Search Reverse*, arbeitet genauso wie BSF, nur daß die Suche beim höchsten Bit, also Bit 15 bzw. Bit 31 beginnt und in Richtung Bit 0 erfolgt. Auch hier wird die Bitnummer des ersten gesetzten Bits zurückgegeben.

BT, BTC, BTR, BTS

BT, *Bit Test*, BTC, *Bit Test and Complement*, BTR, *Bit Test and Reset* und BTS, *Bit Test and Set*, sind vier Befehle, die sich mit einzelnen Bits in einem Datum beschäftigen. Alle besitzen wie die beiden vorangehenden Befehle zwei Operanden:

- XXX reg, reg z.B. BT AX, BX
BTS EAX, ESI
- XXX mem, reg z.B. BTC WordVar, SI
BTR DWordVar, EDX
- XXX reg, const z.B. BTR WordVar,$4032
BTC DWordVar, $12345678
- XXX mem, const z.B. BTS BX, $4711
BT EAX, $98765432

Auch in diesen Fällen müssen die Operanden 16 bzw. 32 Bits breit sein, als zweiter Operand sind aber auch Konstanten erlaubt.

Die Funktion ist nun zunächst in allen Fällen die gleiche. Es wird geprüft, ob das Bit, das durch den zweiten Operanden bezeichnet wird, im ersten Operanden gesetzt ist. Wenn ja, wird das Carry-Flag gesetzt, andernfalls gelöscht. Danach unterscheiden sich die Befehle: BT ist mit seiner Aktion fertig! BTS jedoch setzt das spezifizierte Bit im Datum explizit, BTR dagegen löscht es explizit. Und BTC polt das Bit einfach um, oder *toggelt* es, wie man sagt.

		AX	BX	C
		1010010101001100	?	?
BT	AX, BX	1010010101001100	2	1
BTR	AX, BX	1010010101000100	3	1
BTS	AX, BX	1010010101011100	4	0
BTC	AX, BX	1010010101101100	5	0

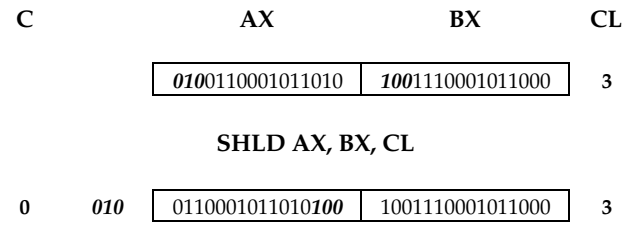
Beachten Sie bitte, daß neben dem Carry-Flag auch noch andere Flags gesetzt werden, die aber die übliche Funktion haben: Das Zero-Flag, wenn das Zielregister nach der Operation 0 ist, das Parity-Flag, wenn eine gerade Anzahl von Bits gesetzt ist, das Sign-Flag, wenn Bit 7/Bit 15/Bit 31 gesetzt sind, je nachdem, ob der Operand ein Byte, Wort oder Doppelwort war.

SHLD, SHRD SHLD und SHRD sind Abarten der Shift-Befehle SHR und SHL. Das *D* steht für *Double Precision* und soll anzeigen, daß mit diesen Befehlen Bitverschiebungen mit »doppelter Präzision«, also höchst genau ausgeführt werden können.

Wie wir bei den Schiebebefehlen des 8086 gesehen haben, kann man doch eigentlich gar nicht »ungenau« shiften, oder? Doch, man kann! Während man nämlich bei den »normalen« Shift-Befehlen lediglich 0-Bits nachschieben kann, kann man bei SHxD vorgeben, welche Bits verwendet werden sollen. Und das geht wie folgt. Die SHxD-Befehle besitzen hierzu *drei* Operanden. Operand 1 und Operand 3 sind die gleichen, wie in den SHx-Befehlen des 80186 auch. Sie bezeichnen das zu manipulierende Bitfeld (Operand 1) und die Konstante bzw. das Register, in dem die Anzahl zu verschiebender Bits steht (Operand 3).

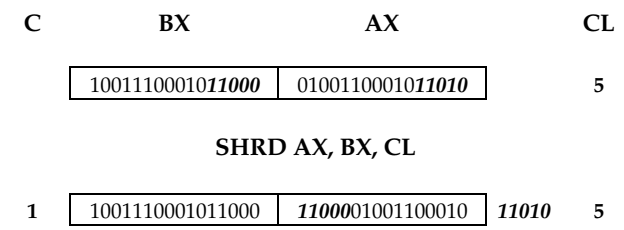
Mit Operand 2 kann (bzw. muß) nun ein weiteres Bitfeld angegeben werden. Aus diesem Bitfeld werden die Bits geholt, die in Operand 1 geschoben werden sollen. Das aber ist der eigentliche Clou! Denn an dieser Stelle konnten mit den SHx-Befehlen nur Nullen injiziert werden!

Schauen wir uns das einmal mit einem Wortregister an:



Wie Sie sehen können, wurden durch SHLD AX, BX, CL die in CL stehende Anzahl Bits (also 3) aus AX nach links herausgeschoben. Die fehlenden drei Bits wurden auf der linken Seite des BX-Registers entnommen. Das BX-Register hat seinen Inhalt dabei nicht verändert. Das Carry-Flag entält das zuletzt aus AX herausgeschobene Bit, hier also 0.

Natürlich funktioniert das ganze auch nach rechts:



ACHTUNG Beachten Sie bitte, daß sich die Anordnung der Register, verglichen mit dem letzten Schaubild, verändert hat. Dies hat nichts mit der Reihenfolge der Parameter zu tun, sondern soll lediglich anschaulich demonstrieren, wie man sich das Verlassen der Bits und die Bit-Injektion aus dem durch den zweiten Operanden spezifizierten Wert vorzustellen hat.

TIP Vielleicht vermissen Sie Rotationsbefehle, analog zu den Bitschiebebefehlen ROR oder RCL des 8086. Diese gibt es tatsächlich nicht, sie sind auch nicht nötig! Denn im Gegensatz zu SHx-Befehlen, bei denen die Bits tatsächlich verlorengehen und daher aufgefüllt werden müssen, bleiben bei ROx und RCx alle Bits erhalten. Probieren Sie dennoch einmal SHRD AX, AX, CL!

Übrigens werden die Flags wie üblich gesetzt: das Zero-Flag, wenn das Register nach der Bitverschiebung leer ist, Parity bei einer geraden Anzahl von gesetzten Bits, und Sign, wenn Bit 15/Bit 31 gesetzt sind. Overflow und Auxiliary sind undefiniert. Bleibt nur noch anzumerken, welche Operandenmöglichkeiten es gibt. Ganz analog zu den 80186-Schiebebefehlen kann man verschieben um:

- eine Anzahl von Positionen, die direkt als Konstante angegeben wird:
  XXX reg16, reg16, const8
  XXX WordVar, reg16, const8
  XXX reg32, reg32, const8
  XXX DWordVar, reg32, const8
- eine Anzahl von Positionen, die in CL verzeichnet ist, also
  XXX reg16, reg16, CL
  XXX WordVar, reg16, CL
  XXX reg32, reg32, CL
  XXX DWordVar, reg32, CL

Leider funktioniert das nicht mit Byte-Registern und -Speicherstellen.

CWDE, CDQ

CWDE heißt *Convert Word to Double Word Extended*. Es dient zur vorzeichenbehafteten Umwandlung eines Wortes in ein Doppelwort. Nun werden Sie einwenden, daß dies CWD im 8086-Befehlssatz ja schon kann und der 80386 somit auch. Richtig! Allerdings hat der 8086 keine 32-Bit-Register, und ein Doppelwort muß somit in eine Registerkombination expandiert werden. Der 80386 jedoch hat echte 32-Bit-Register, die ein Doppelwort aufnehmen können. CWD expandiert also AX in DX:AX, CWDE expandiert AX in EAX. Der Rest ist tatsächlich gleich!

CDQ, *Convert Double Word to Quad Word*, wandelt das *Doppelwort* in EAX in ein *Quadwort* in EDX:EAX vorzeichenbehaftet um. Es arbeitet somit vollständig analog zu den anderen Expansionsbefehlen.

Mit den nun bekannten Befehlen kann ein einzelnes, vorzeichenbehaftetes Byte im 80386er wie folgt in ein 8 Byte umfassendes Quadwort unter Beibehaltung des Vorzeichens expandiert werden:

```
cbw             ; expandiere Byte zu Word ; 8086
cwde            ; expandiere Word zu DWord ; 80386
cdq             ; expandiere DWord zu QWord ; 80386
```

Ergebnis: eine Zahl in EDX:EAX.

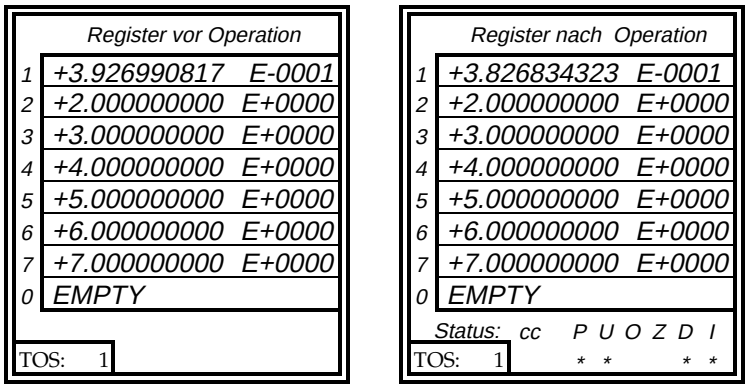
Die 8086-Variante bis zum Doppelwort (Quadworte würden hier alle Rechenregister belegen und sind daher Unsinn) hieße:

```
cbw             ; expandiere Byte zu Word
cwd             ; expandiere Word zu DWord
```

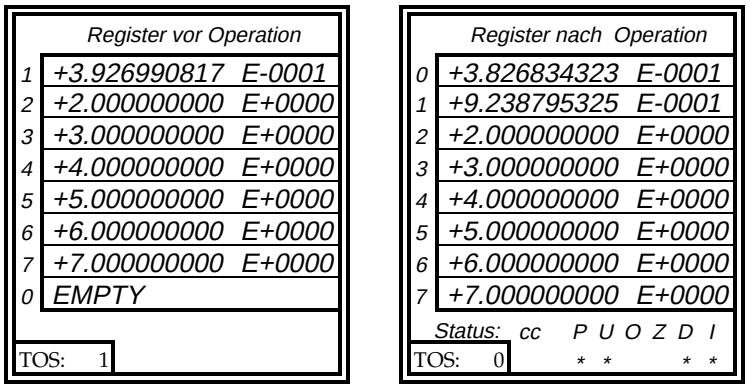
Ergebnis: eine Zahl in DX:AX.

FCOS, FSIN, FSINCOS

FSIN und FCOS ersetzten den Wert in TOS durch den Sinus bzw. Cosinus des Wertes, hier gezeigt am Sinus (vgl. Abbildung auf der folgenden Seite).



FSINCOS erledigt das gleiche, wobei jedoch der Sinus im TOS, der Cosinus in ST(1) steht, wofür natürlich ein Pushen des Stacks notwendig wird:



FPREM1

FPREM1 arbeitet praktisch genauso wie FPREM beim 8086. Auch hier wird so lange die Zahl in ST(1) vom TOS abgezogen, bis ein Rest übrig bleibt. Der Unterschied besteht lediglich darin, wie in bestimmten (»kritischen«) Fällen reagiert wird. Während FPREM sich, weil damals noch nicht definiert, nicht an den sogenannten IEEE-Standard 754 hält, tut FPREM1 dies sehr wohl!

Falls der Rest einer Division mit FPREM/FPREM1 in der Nähe des Divisors liegt, so rundet FPREM in jedem Fall in Richtung »0« – egal ob dies »richtiger« ist oder nicht. FPREM1 dagegen rundet in Richtung auf die »nächste geradzahlige« Zahl, also je nach Wert zur nächsten ganzen Zahl auf oder ab.

ACHTUNG

Zwar handelt es sich bei FPREM/FPREM1 um Befehle, die die Restbildung nicht durch (fehlerbehaftete) Division erzeugen, sondern

durch (die wesentlich exaktere) wiederholte Subtraktion des Divisors vom Dividenden. Doch hat niemand verlangt, daß lediglich Integer durch FPREM/FPREM1 manipuliert werden dürfen. Gültige Operanden sind also auch Reals – und bei Subtraktionen von Reals fallen unweigerlich Rundungen an. Daher kann es natürlich auch zu Rundungsfehlern kommen. In diesem Punkt unterscheiden sich FPREM und FPREM1: Während FPREM immer abrundet, was zu einer Akkumulation des Rundungsfehlers führt, rundet FPREM1 in diesem Fall in Abhängigkeit von der Situation »korrekter« auf und ab.

FUCOM, FUCOMP, FUCOMPP

Hinter diesen Befehlen verstecken sich Spezialfälle der Befehle FCOM, FCOMP und FCOMPP. Während nämlich bei den letzteren die Verwendung einer *NaN* zu einem Ausnahmezustand führt, der durch Setzen des entsprechenden Flags angezeigt wird, behandeln die »U«-Zwillinge NaNs als gültige Zahlen, die lediglich nicht verglichen werden können. So seltsam es klingen mag: FUCOM, FUCOMP und FUCOMPP machen haargenau das gleiche wie FCOM, FCOMP und FCOMPP, setzen aber nicht das Exception-Flag, falls NaNs verwendet werden, sondern lediglich den Condition Code entsprechend.

IMUL

Eine Veränderung sollten wir noch erwähnen: die Ergänzung des IMUL-Befehls. Er ist nun auch in einer Version verwendbar, die drei Operanden benutzt:

- IMUL reg16, reg16, const8
- IMUL reg16, WordVar, const8
- IMUL reg32, reg32, const8
- IMUL reg32, DWordVar, const8

Hierbei bezeichnet der erste Operand das Ziel der Operation, also den Ort, an dem das Produkt abgelegt werden soll. Wie Sie sehen, kann dies nur ein Register sein. Das Produkt wird dabei aus dem zweiten Operanden und einer vorzeichenbehafteten Konstanten berechnet. Hinsichtlich der Flags läßt sich das gleiche sagen, was schon beim 8086 und der Erweiterung beim 80186 gesagt wurde!

11 Der Adreßraum des 80386

Wie im vorangehenden Kapitel schon angedeutet, besitzen die INTEL-Prozessoren der Generation 80386 – und natürlich deren Nachfolger – 32 Adreßleitungen. Somit sind 2^{32} einzelne Speicherstellen ansprechbar, was einem Adreßraum von 4.294.967.296 Bytes entspricht. Da parallel auch die Register des 80386 32 Bit breit sind, läßt sich jede Adresse, die diese Prozessoren ansprechen können, direkt in

einzelnen Registern – und damit auch in den Adreßregistern – ablegen. Als Konsequenz entfällt die bei den Vorgängern notwendige Speichersegmentierung in Segment:Offset.

Was heißt das nun für den Programmierer? Es wird alles erheblich einfacher! Erstens: Die Zeit der 64-KByte-Segmente ist vorbei, Programme und Datenstrukturen können nun beliebig groß werden, solange sie zusammen nicht größer als 4 GByte sind. Zweitens: Alle Programmadressen sind Near-Adressen (was einem angesichts der riesigen Zahl an Speicherstellen schwerfällt zu glauben)! Da es keine Segmente mehr gibt, gibt es natürlich auch keine Sprünge über Segmentgrenzen hinweg. Jede Adresse im gesamten Adreßraum des 80386 ist gleichberechtigt und beliebig anspringbar, ohne daß irgendwelche Kunststückchen durchgeführt werden müssen. Dieses Speichermodell wird sehr treffend als »flaches Speichermodell«, *Flat [Memory] Model*, bezeichnet. Wie man leicht einsehen kann, ist die Zeigerarithmetik in diesem Modell sehr einfach: Es gibt keinen Überlauf mehr an Segmentgrenzen, der berücksichtigt werden muß, einfache lineare Addition von Distanzen zu Adressen. Drittens: Es gibt keinen *heap* mehr – zumindest nicht notwendigerweise. Da das Datensegment beliebig groß sein kann – die einzige Beschränkung ist, daß Code und Daten zusammen nicht größer als 4 GByte sind –, entfällt der Zwang zur Unterscheidung von Daten und »speziellen« Daten!

Nun werden Sie sich fragen, wozu dann noch die Segmentregister notwendig sind, wenn doch der ganze Adreßraum in einem Register abgebildet werden kann. Nachdem der Instruction-Pointer EIP ja 32 Bit breit ist, entfällt die Notwendigkeit für CS. Auch DS kann entfallen, wenn die anzusprechenden Adressen für Daten über EDI, ESI, EBX etc. direkt verwaltet werden können – ähnliches gilt für den Stack. Und theoretisch haben Sie auch recht. Daß dennoch auch weiterhin CS, DS, ES und SS, ja sogar die neu hinzugekommenen Segmentregister FS und GS eine Existenzberechtigung behalten werden, werden wir noch sehen.

Mit dem 80386 beginnt nun eine Zeit, in der erstmalig mehr Speicherplatz angesprochen werden kann, als physikalisch vorhanden ist. Denn es wird sicherlich noch einige Zeit dauern, bis es PCs gibt, die über 4 GByte RAM verfügen – falls dies mit den heute üblichen Prozessorfamilien jemals realisiert werden sollte. Bisher war immer das Betriebssystem der limitierende Faktor: Schon ab dem 8086 war wohl jeder PC mit mehr RAM bestückt, als linear adressiert werden konnte, und ab dem 80286 sogar mit mehr, als auch mit Speichersegmentierung möglich war – die magische 1-MByte-Grenze von DOS und die sogenannten DOS-Extender lassen grüßen. Das ist nun anders, und diese Errungenschaft hat natürlich Konsequenzen.

Denn das Betriebssystem muß nun sicherstellen, daß Programme, die den Adreßraum des 80386 nutzen, nicht versuchen, auf Adressen zurückzugreifen, die es physikalisch gar nicht gibt. Dies wird in den heutigen 32-Bit-Betriebssystemen recht trickreich gemacht, wie wir ebenfalls noch sehen werden.

Eine weitere wesentliche Neuerung, die direkt mit 32-Bit-Adressierung im Zusammenhang steht, ist die Berechnung einer effektiven Adresse EA. Die EA konnte bisher, bei Nutzung von 16-Bit-Adressen, auf verschiedene Arten ermittelt werden: direkt durch Angabe einer gültigen 16-Bit-Adresse (vgl. JMP) oder indirekt, wobei dann die EA irgendwie berechnet oder zumindest erst beschafft werden mußte: EA = Basis + Index + Konstante. Hierbei haben wir als Basisregister die Register BP und BX und als Indexregister DI und SI kennengelernt.

Dies ist zwar unter der 32-Bit-Adressierung prinzipiell gleichgeblieben, doch wurden die Möglichkeiten stark erweitert:

- Prinzipiell kann nun jedes der 32-Bit-Register für die indirekte Adressierung benutzt werden. Es gibt keine Beschränkung auf BP, BX, SI und DI!
- Jedes der möglichen Register kann sowohl als Basis- als auch als Indexregister herhalten (eine kleine Ausnahme besprechen wir später). Mehr noch, zur Berechnung der EA kann ein Register beide Funktionen gleichzeitig erfüllen!
- Der Index kann mit den Werten 1, 2, 4 und 8 skaliert werden.

Es ist klar, daß diese Erweiterungen Änderungen an den Opcodes im 32-Bit-Modus notwendig werden lassen. Wir werden dies jedoch im dritten Teil des Buches genauer kennenlernen.

Abschließend sei bemerkt, daß der 80386 selbstverständlich auch die Prozessoren bis zum 80286 emulieren und somit als »schneller« 80286 mit 32-Bit-(Rechen)-Registern aufgefaßt werden kann. Das bedeutet, daß natürlich auch der Adreßraum des 8086 noch seine Berechtigung hat. Ja mehr noch: Bis zur vollständigen Nutzung der Möglichkeiten des 80386 und seiner potenteren Nachfolger in echten 32-Bit-Betriebssystemen wie Windows 95, Windows NT oder OS/2 bleibt dem Programmierer die schöne neue Welt vorenthalten. Das gilt auch für die 32-Bit-Erweiterung von Windows 3.xx, das sog. Win32s. Hierbei handelt es sich um eine DLL, die 32-Bit-Adressen in die 16-Bit-Adressen des segmentierten Speichermodells umsetzt. Von echter 32-Bit-Programmierung kann also keine Rede sein.

12 Änderungen beim 80486

Die auffälligste Änderung beim 80486 ist, daß er keinen arithmetischen Coprozessor mehr besitzt. Beim 80486 ist dieser eingebaut.

Allerdings darf man sich darauf nicht verlassen. Denn wirklich eingebaut ist der Coprozessor nur bei den Chips des 80486, die die kleine, aber wichtige Namenserweiterung *DX* haben. Neben diesen »DX-486ern« gibt es allerdings auch »Schmalspur-486er« mit Namen *SX*. Stellen Sie sich diese am besten als *DX* mit kaputtem Coprozessorteil vor.

Rechner, die diesen Prozessor besitzen, funktionieren genauso wie ihre großen Brüder; das Rechnen geht nur etwas langsamer, da Fließkommaberechnungen wie bei jedem Prozessor, der keinen arithmetischen Coprozessor hat, emuliert werden müssen. Aber auch für diese Rechner gibt es Abhilfe in Form eines Coprozessors 80487SX. Diesen stellen Sie sich am besten als vollständigen 80486DX vor.

Was nun passiert, wenn man einen 80487SX in den dafür vorgesehenen Platz auf der Platine steckt, ist einfach überwältigend: der 80486SX wird einfach abgeschaltet! Da der 80487SX ja ein verkappter 80486DX ist, übernimmt er alle Aufgaben, die der 80486SX eigentlich erledigen müßte. Dieser wird somit absolut überflüssig. Es ist jedoch nicht möglich, den 80486 zu entfernen und an seine Stelle den 80487 zu setzen, weil die beiden Prozessoren nicht pin-kompatibel sind. Dies heißt nichts anderes, als daß der Coprozessor nicht in den Sockel für den Prozessor paßt, weil er 169 »Füßchen« hat und nicht 168, wie der 80486SX. Dennoch sollte man nicht die Flinte ins Korn werfen: Man kann den 80486SX einfach entfernen und den 80487 in seinem *Upgrade Socket* wie einen 80486DX betreiben.

Weil die Situation bei diesem Prozessortyp eben so ist, wie sie ist, bleiben wir gleich bei der Besprechung des 80486DX und setzen die Kombination 80486SX/80487SX mit diesem gleich.

Eine weitere Veränderung beim 80486 ist, daß er über einige neue Möglichkeiten verfügt, die sich jedoch hauptsächlich in der Hardware äußern, weniger aber in der Programmierung. So sind die Befehle des 80386/80387, die der 80486 natürlich kennt, erheblich schneller geworden. Dies wurde möglich, da Intel den Chip vollkommen neu gestylt und den Microcode optimiert hat. Hinzu kamen auch andere Hardwarekomponenten, die jedoch hauptsächlich im Hintergrund arbeiten, so daß ich sie hier nicht ansprechen möchte.

Am Umfang des Befehlssatzes hat sich beim 80486/80487 gegenüber seinem Vorgänger wenig geändert. Es sind lediglich sechs neue Befehle hinzugekommen, von denen drei für den täglichen Gebrauch verwendet werden können.

- XADD. Die *Extended Addition* funktioniert wie ein ganz normaler ADD-Befehl, nur daß sich nach der Addition der Inhalt des Zielregisters, also des Operanden 1, in Operand 2, also in der Quelle wiederfindet. Falls also z.B. in AX der Wert $1234 steht und in BX $9876, so findet sich nach XADD AX, BX in AX das Ergebnis der Addition wieder, also $AAAA, während in BX der ehemalige Inhalt von AX, also $1234 steht. Aus diesem Grund kann bei XADD keine Konstante verwendet werden.
- CMPXCHG. Dieser Vergleichsbefehl, *Compare and Exchange*, arbeitet etwas seltsam. Er besitzt zwei Operanden, vergleicht aber zunächst nur den Inhalt des Akkumulators, also AL, AX oder EAX, mit dem Inhalt des ersten Operanden. Sind beide Werte gleich, so wird der erste Operand mit dem Wert des zweiten beladen. Sind sie nicht gleich, so erhält der Akkumulator den Wert aus dem ersten Operanden. Enthalten wir uns eines Kommentars, wie sinnvoll ein solcher Befehl ist. Daß er allerdings häufiger benutzt wird, sehen Sie daran, daß Intel ihn eingeführt hat[5].
- BSWAP. BSWAP, *Byte Swap*, dient zur Überführung einer Zahl im Intel-Format in das Motorola-Format. Dahinter steckt, daß die beiden Prozessorhersteller Intel und Motorola unterschiedliche Verfahren benutzen, zum Daten abzuspeichern. Während Motorola nämlich die Zahl $4711 z.B. in der Reihenfolge abspeichert, in der wir es auch erwarten, also das höherwertige Byte des Wortes zuerst, dann das niederwertige, so erfolgt diese Abspeicherung bei Intel genau umgekehrt. Im Speicher steht also $47 vor $11, wenn $4711 im Motorola-Format abgelegt wird, während nach Intel-Konvention $11 dem Byte $47 vorangeht.

  Dies ist aber nur eine unterschiedliche Art, Daten mit mehr als einem Byte abzuspeichern! In beiden Fällen heißt die Zahl $4711 und besitzt auch deren Wert. Es ist lediglich Interpretationssache des betreffenden Prozessors, also bei PCs der Intel-Prozessoren und deren Clones, die somit alle das Intel-Format benutzen.

  BSWAP dient dazu, Daten der verschiedenen Formate ineinander zu überführen, um Kompatibilität zu Intel-fremden Prozessoren

---

5 … dachte ich zunächst! Aber schon ein Autorenkollege sagte einmal: »Wenn man nicht alles selbst nachprüft…«. Sie werden im zweiten Teil des Buches aus allen Wolken fallen, wenn wir über CMPXCHG genauer sprechen.

herzustellen. Die beiden »äußeren« Bytes, also das niedrigstwertige und das höchstwertige, werden vertauscht, dann die jeweils nächsten »nach innen« gehenden Bytes. Aus $4711 wird so $1147, aus $12345678 wird $78563412

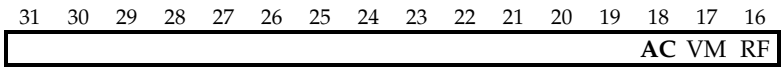
- INVDPG. *Invalidate Page* ist ein Befehl, der auf eine spezifische Eigenschaft des 80486 zielt. Dieser Prozessor hat nämlich einen sogenannten *Translation Lookaside Buffer*, der bei der *virtuellen Speicherverwaltung* benutzt wird. Dieser Puffer dient dazu, logische Adressen in physikalische umzusetzen.

  INVDPG dient nun dazu, eine Seite, die in diesem Buffer gehalten wird, für ungültig zu erklären, sie somit praktisch zu löschen!
- INVD. Auch INVD, *Invalidate Cache*, werden wir nicht nutzen – er überschreitet die Grenzen dieses Buches bei weitem! Nur soviel: Der 80486 hat einen sogenannten *Hardware-Cache*, in den und aus dem Daten transportiert werden, wenn der Prozessor auf sie zugreift. Ich möchte es damit bewenden lassen, zu erklären, daß der Cache ein Spezialspeicher auf dem Chip ist, über den der Datenaustausch mit dem eigentlichen Speicher erfolgt. Dies beschleunigt Schreib-/Lesevorgänge des Prozessors erheblich, da die aktuellen, benötigten Daten nicht über den langsamen Datenbus geschaufelt werden müssen, sondern praktisch direkt auf dem Chip gehalten werden können.

  Dennoch muß hin und wieder der Cache auch Daten mit dem Speicher austauschen. Und um vorher den (nicht mehr gültigen) Inhalt aus dem Cache zu entfernen, muß dieser geleert werden. Dies erfolgt mit INVD.

  INVD verwirft also alle Daten, die sich im Cache befinden, sofort, d.h. ohne Überprüfung, ob sie in den Speicher gerettet wurden.
- WBINVD, *Write Before Cache Invalidation*, tut das gleiche wie INVD. Allerdings wird vorher noch der Inhalt des Caches in den Speicher zurückgeschrieben, so daß keine Daten verlorengehen.

Natürlich hat der 80486 auch Änderungen im Flagregister erfahren. Dargestellt ist hier nur das obere Wort des EFlagregisters mit den Bits 31 bis 16. Das niederwertige Wort hat sich (natürlich!) gegenüber dem 80386 nicht verändert.



Das ***A**lignment-**C**heck-Flag* steuert, wie sich der Prozessor verhalten soll, wenn ein Zugriff auf »nicht ausgerichtete Daten« erfolgt. Ohne ins Detail gehen zu wollen: Wenn ein Datum an *DWord*-Grenzen aus-

gerichtet ist, so kann durch den Prozessor wesentlich schneller darauf zurückgegriffen werden. Das *AC*-Flag hilft nun dem Programmentwickler, indem, falls es gesetzt ist, geprüft wird, ob diese Ausrichtung vorliegt. Wenn nicht, so wird ein Interrupt ausgelöst.

Falls Ihnen das nichts weiter gesagt hat, so macht das auch nichts. Wir werden dieses Flag nur ein einziges Mal benutzen, nämlich dann, wenn wir eine Routine zur Feststellung des Prozessortyps vorstellen.

13 Zusammenarbeit zwischen dem 80486 und der Floating-Point-Unit

Wie im vorangehenden Kapitel schon beschrieben, ist aus der Zusammenarbeit zwischen dem Prozessor und seinem Coprozessor eine Symbiose geworden. Man kann nun beide Teile nicht mehr voneinander trennen, wenn man vom 80486SX einmal absieht. Dies hat natürlich gewaltige Vorteile, denn der Datenaustausch zwischen Komponenten auf einem Chip ist immer schneller und einfacher als über Datenleitungen zwischen zwei Chips.

Außerdem konnte die Zusammenarbeit zwischen den beiden Teilen auf diese Weise optimiert werden. So greifen beide Teile auf den gleichen *Cache* zurück, es entfallen Steuerleitungen usw. Mit dem 80486 (DX) ist also zum erstenmal das gelungen, was beginnend mit dem Paar 8086/8087 dem Programmierer schon vorgegaukelt wurde: ein Chip mit vielen Registern und sehr potenten Befehlen, der alle Aufgaben bewältigen kann.

14 Änderungen beim Pentium

Eine der wesentlichen Neuerungen beim Pentium ist ein neuer Betriebsmodus, der *System Management Mode* (SMM). Dieser Modus dient dazu, den Prozessor nach einer bestimmten Zeit der Inaktivität in einen *Stand-by*-Modus zu schicken, ohne Daten zu verlieren. Er wird durch ein externes Signal eingeschaltet und kann nur durch einen der neu hinzugekommenen Befehle wieder verlassen werden.

Deutlich verbessert wurden auch die Debugging-Möglichkeiten der 80386/80486er mit ihren dafür vorhandenen Spezialregistern. Dies äußert sich z.B. in einem neu hinzugekommenen Register, das allerdings nicht für die allgemeine Verwendung benutzt werden kann. Dieses als *Control Register* CR4 bezeichnete Register nimmt Flags auf, die zur Steuerung von pentiumspezifischen Aktionen dienen. Ein weiteres Stichwort, das aber nicht weiter erläutert werden soll, ist die neue und schnellere arithmeti-

sche Bearbeitung von Daten mittels *Pipelinig*, die vor allem die aufwendige Fließkommaberechnung erheblich beschleunigt.

Durch diese Art der Fließkommaberechnung können mehrere Zahlen quasi gleichzeitig wie an einem Fließband bearbeitet werden. Da dies vollständig im Hintergrund abläuft und mit den uns zur Verfügung stehenden Mitteln nicht beeinflußt werden kann, verweise ich Interessierte auf weiterführende Literatur.

Zwangsläufig hat sich aufgrund dieser Neuerungen auch beim Pentium im Befehlsumfang gegenüber seinem Vorgänger etwas, wenn auch sehr wenig geändert. Es sind lediglich sechs neue Befehle hinzugekommen.

- CMPXCHG8B, *Compare and Exchange 8 Bytes*. Dieser Befehl ist eine Erweiterung des CMPXCHG-Befehls des 80486. Er besitzt formal wie dieser drei Operanden, von denen zwei jedoch implizit vorgegeben sind. Der Unterschied zu CMPXCHG besteht darin, daß CMPXCHG8B mit 8 Byte großen Daten umgeht, also 64-Bit-Daten. Da jedoch die Prozessorregister gegenüber dem 80486 nicht breiter geworden sind, also ebenfalls »nur« 32 Bits breit sind, müssen alle vier Allgemeinregister zur Verwendung kommen, um zwei der drei Operanden aufnehmen zu können. Daraus resultiert die implizite Vorgabe von zwei Operanden in EDX:EAX und ECX:EBX.

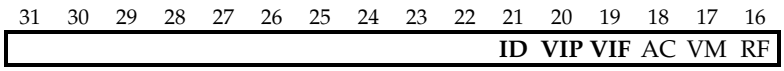
  Ansonsten läuft alles wie beim CMPXCHG-Befehl ab. Der Wert in EDX:EAX wird mit dem 8-Byte-Wert des Operanden verglichen. Haben beide den gleichen Inhalt, so wird dieser in ECX:EBX gespeichert und das Zero-Flag gesetzt. Andernfalls wird der Wert aus dem Operand in EDX:EAX kopiert und das Zero-Flag gelöscht.
- CPUID. Darauf haben wahrscheinlich nicht nur die Programmier-Freaks sehnsüchtig gewartet. Während man nämlich bisher nur die Eigenheiten und marginalen Unterschiede im Design der einzelnen Prozessortypen dazu heranziehen konnte, in mehr oder weniger trickreichen Routinen den Prozessortyp festzustellen[6], hat Intel dem Pentium nun einen Befehl spendiert, der nicht nur kundtut, daß ein Pentium vorhanden ist, sondern darüber hinaus auch weitere, zum Teil sehr wichtige Informationen zur Verfügung stellt (inklusive der für Intel sicherlich äußerst wichtigen Meldung *GenuineIntel*!). Aus der Tatsache, daß CPUID als Prozessortyp beim Pentium 5 zurückgibt, kann man schließen, daß dies auch beim Pentium-Nachfolger beibehalten wird.

---

6 Was wir natürlich auch noch tun werden!

- RSM, *resume from system management mode*. Dieser Befehl schaltet den Pentium aus dem neu hinzugekommenen Modus, dem *system management mode* (SMM), zurück in den aktiven Betriebszustand.
- RDMSR/WRMSR. Dieses Befehlspaar dient dazu, auf die neuen, prozessorspezifischen Register zugreifen zu können, mit denen verschiedene prozessorspezifische Funktionen ermöglicht werden: *Performance Monitoring*, *Execution Tracing*, *Machine Checking* usw. Die Besprechung dieses Befehls und der involvierten Register würde im Rahmen dieses Buches zu weit führen.
- RDTSC. Mit diesem Befehl wird der sog. *Time Stamp Counter* ausgelesen, ein modellspezifisches Register, in dem ein Zähler geführt wird. Dieser Zähler wird bei jedem Prozessor-Reset auf »0« gesetzt und mit jedem Prozessorzyklus inkrementiert. Auch in diesem Fall verzichte ich auf eine weitere Besprechung.

Natürlich hat auch der Pentium Änderungen im Flagregister erfahren.



Nicht nur, daß es seit dem 80386 virtuelle Modi, virtuellen Speicher und andere virtuelle Dinge gibt! Ab dem Pentium gibt es sogar *virtuelle Interrupts*! Diese haben mit dem *Virtual-Interrupt-Pending*-Bit (VIP) und dem *Virtual-Interrupt-Flag* (VIF) zu tun! Auch hier gilt wieder: Ich kann Ihnen keine weitere Erklärung zu diesen Mechanismen geben, da sie offensichtlich top secret sind. *ID* nun ist sehr interessant! Läßt sich dieses Flag verändern, so unterstützt der Prozessor den Befehl CPUID. Auf diese Weise kann man feststellen, ob CPUID aufgerufen werden darf.

Beim/Seit dem Pentium muß nun die Zusammenarbeit mit dem Coprozessor nicht mehr berücksichtigt werden. Auf jedem Pentium-Chip ist eine *Floating-Point-Unit* vorhanden, die dem Prozessor die Fähigkeit zur Ausführung von Coprozessorbefehlen gibt.

15 Änderungen beim Pentium Pro

Es hat sich wenig bei der Weiterentwicklung des Pentium zum P6, der nun Pentium Pro heißt, getan. Nicht etwa hinsichtlich der Neuerungen oder Verbesserungen, jedoch hinsichtlich dessen, was im Rahmen dieses Buches besprochen werden soll: Es gibt fünf neue Befehle, keine neuen Allzweckregister, keine neuen Flags. Ich möchte aber hier beileibe nicht in den Chor derer einstimmen, die den Pentium Pro als Flop bezeichnen, nur

weil er unter Windows 3.xx gegenüber den neuen, hochgetakteten Pentiums nicht nur keine Leistungsverbesserung bringt, sondern vielleicht hier und da sogar Leistungseinbußen aufweist.

Der Grund hierfür liegt darin, daß der Pentium Pro für 32-Bit-Systeme ausgelegt und optimiert wurde. Hier liegen zweifelsohne seine Stärken, was sich spätestens dann zeigt, wenn er mit echten 32-Bit-Betriebssystemen und 32-Bit-Anwendungen arbeitet. 16-Bit-Code bremst ihn naturgemäß und architekturbedingt aus.

Das »Wenige«, das sich im Befehlsvorrat getan hat, hat es aber in sich:

- CMOVcc, *Conditional Move*. CMOVcc sieht aus wie SETcc und Jcc, so daß angenommen werden kann, daß *cc* wieder einmal für verschiedene Bedingungen steht, die wir von Jcc her schon kennen. Stimmt. Die Reaktion des Prozessors ist auch ziemlich eindeutig ablesbar: ein MOV. Summa: CMOVcc prüft wie Jcc und SETcc die verschiedenen Flags bzw. Flagkombinationen (des 32-Bit-EFlagregisters) und kopiert je nach Erfüllung der Bedingung den Wert aus der Quelle in das Ziel oder nicht.
- FCMOVcc, *Floating Point Conditional Move*. Der gleiche Befehl wie CMOVcc, jedoch erfolgt hier das Kopieren innerhalb der Floating-Point-Unit des Pentium Pro (der hat ja, wie auch schon der Pentium, keinen separaten Coprozessor mehr, deshalb sollte man dies auch so ausdrücken!). Beachten Sie, daß mit diesem Befehl erstmalig eine Aktion mit Fließkommazahlen vom Zustand der Flags im Flagregister der CPU abhängig gemacht wird!
- FCOMI, FCOMIP, FUCOMI und FUCOMIP: *Floating Point Compare and Set Eflags – with Stack Pop / Unordered / Unordered with Stack Pop*. Ein sehr wichtiger Befehl, da er die CPU-Flags in Abhängigkeit vom Vergleich zweier Werte in der Floating-Point-Unit setzt! Also gehören die umständlichen Kopierspielchen der FPU-Flags über das AX-Register/Speicherstelle und explizites Laden in das Flagregister endlich der Vergangenheit an!
- RDPMC, *Read Performance Monitoring Counters*. Dieser Befehl führt bestimmte Aktionen beim Pentium Pro aus, deren Besprechung in diesem Buch zu weit geht. Belassen wir es dabei zu wissen, daß es ihn gibt.
- UD2. UD2 ist eine undefinierte Instruktion und führt eine *Invalid Instruction Exception* aus. Sie dient zum Austesten von Software in der Entwicklungsphase.

Die Befehle CPUID, RDMSR und WRMSR wurden verändert und an die neuen Gegebenheiten des Pentium Pro angepaßt. Da die Besprechung der letzten beiden Befehle den Rahmen dieses Buches sprengen würde, sei auf weitere Details verzichtet – die Änderungen an CPUID werden in Teil 3 besprochen.

16 Änderungen beim Pentium II

In der Ahnenreihe der Intel-Prozessoren folgen dem altehrwürdigen 80386 und seinem Nachfolger, dem 80486, zwei neue Prozessorgenerationen: P5 und P6. Der P5 ist unter dem Namen Pentium bekannt geworden. Nicht so der P6. Denn beim P6 handelt es sich um eine ganze Familie, die beim Pentium Pro beginnt, mit dem Pentium II fortgesetzt wurde und in den »neuen«, auf bestimmte Anwendungsbereiche hin maßgeschneiderten Pentiums wie dem Xeon oder Celeron ein vorläufiges Ende gefunden hat. Die Unterschiede sind programmtechnisch gesehen marginal. So besitzen neuere Prozessoren einen größeren Cache (den Sie aber niemals direkt nutzen können!), andere haben einen *Sleep Mode*, vielleicht gar einen *Deep Sleep Mode*, um in batterieabhängigen Geräten für das Stromsparen zu sorgen. Manche besitzen Eigenschaften, die sie für Server prädestinieren, *Workstations* glänzen lassen oder in Low-End-Geräten ihren Zweck mehr als erfüllen. Alles in allem: nichts Weltbewegendes. (Wie bereits gesagt: vom Standpunkt des Hochsprachen- oder Assemblerprogrammierers aus betrachtet.)

Wenn das so ist, sollte es zwischen dem Pentium Pro und dem Pentium II also eigentlich keine Unterschiede geben, die uns als Programmierer interessieren müßten, und somit müßte dieses Kapitel überflüssig sein. Es sollten die Befehlssätze beider Prozessortypen identisch sein. Wie ist die Antwort von Radio Eriwan? »Im Prinzip ja! Aber ...«

Wie im täglichen Leben sind auch bei Prozessorfamilien manche Individuen ein wenig gleicher als die anderen. So wartet der Pentium II hardwareseitig mit einer neuen Möglichkeit für ein *Umschalten* zwischen einer Privilegstufe, in der Anwendungen realisiert werden, und der Privilegstufe auf, in der Kernfunktionen des Betriebssystems anzusiedeln sind (CPL = 0). Hintergrund: Optimierung der Performance. (Wir werden auf das Thema Privilegien in einem eigenen Kapitel zurückkommen!)

Um das zu erreichen, hat Intel den *Fast System Call* erschaffen. Wie gesagt, das ist eine Möglichkeit für Anwendungen in niedriger privilegierten Stufen, schnell auf Funktionen des am höchsten privilegierten Systemkerns zuzugreifen. Diese *Fast System Calls* können mit zwei neuen Befehlen erreicht werden:

SYSENTER, SYSEXIT

Beide Funktionen kopieren die Inhalte bestimmter Felder in modellspezifischen Registern des Prozessors in die Register CS:EIP und

SS:ESP. Damit ist ein sehr schneller Einsprung mit Stackwechsel möglich. Ich werde aber auf diese speziellen Funktionen nicht weiter eingehen, da sie erstens nicht abwärtskompatibel sind und zweitens nur in wenigen Ausnahmesituationen eine Rolle spielen.

Wie kann man feststellen, daß diese beiden Funktionen unterstützt werden? Über die *Feature-Bits* des Prozessors, die man mittels des CPUID-Befehls in EDX zurückbekommt, wenn man in EAX 1 übergibt. Ist Bit 11, das *Sysenter-Present-Bit* SEP, gesetzt, wird der *Fast System Call* unterstützt. Allerdings nur ab dem Pentium II – und dort auch nicht von allen Prozessoren. Daher muß noch geprüft werden, ob die Prozessorfamilie = 6 ist, die Modellnummer ≥ 3 und die Stepping-ID ebenfalls ≥ 3. (Vor dem Pentium II galt dieses Bit als reserviert!)

Aber weil uns Programmierer das nun wirklich nicht sonderlich interessiert – wer will sich schon aufbürden, Windows NT oder 9x aufzubohren! –, möchte ich es damit bewenden lassen.

17 Die MMX-Technologie

Intel hat dem Bedarf an bestimmten Datenformaten und Datenverarbeitungen, die sich im Rahmen von Multimedia und Kommunikation durchgesetzt haben, Rechnung getragen. Und zwar dadurch, daß die MMX-Technologie eingeführt wurde. MMX steht für *Multi Media Extension*. Damit ist bereits viel gesagt!

Unter Multimedia sind zwei Datentypen besonders wichtig: Bytes, die im Bereich Video/Grafik eine dominante Rolle spielen können, und Worte, die im Audiobereich dominieren, aber auch bei Video/Grafik zum Einsatz kommen. Noch eines ist unter Multimedia wichtig: und zwar die Verarbeitung einer großen Menge an Daten – Daten, die durchaus im Rahmen paralleler Algorithmen bearbeitet werden könnten.

Kein Befehl der Intel-Prozessoren vor Einsatz der MMX-Technologie ermöglichte es bislang, Daten parallel zu verarbeiten. (Unter paralleler Datenverarbeitung verstehe ich hier tatsächlich die parallele Verarbeitung gleichartiger, voneinander unabhängiger Daten, nicht etwa die parallele Ausführung parallelisierbarer Instruktionen!) Zwar konnten die Allzweckregister ab dem 80386 vier Bytes oder zwei Worte aufnehmen. Aber eine unabhängige, parallele Verarbeitung mittels eines Befehles war und ist nicht möglich. Performancesteigerungen im Multimediabereich beruhten in der Regel auf einer deutlich höheren Taktrate neuerer Prozessoren, die oft genug aufgrund

unzureichender Programmierung und/oder betriebssystem- oder hardwarebedingter Rahmenbedingungen auf ein fast unbedeutendes Maß zurückgeschraubt wurde: Ein doppelt so schneller Prozessor hat niemals eine doppelt so schnelle Programmausführung gewährleistet.

Mit MMX hat Intel dies geändert. Nun ist zum erstenmal ein echter Performancegewinn möglich geworden, weil mit den neuen Befehlen bis zu acht (Video-) Bytes oder vier (Audio-) Worte mittels eines Befehls manipuliert werden können. Das bedeutet, daß allein aufgrund dieser »Parallelisierung« ein Faktor vier bis acht (zumindest theoretisch!) an Performance gewonnen werden kann – eine geeignete Programmierung vorausgesetzt. MMX steht noch für etwas weiteres: Nach Angaben von Intel resultiert die MMX-Technologie aus einer intensiven Analyse der Notwendigkeiten und Erfordernisse bestehender Multimediaanwendungen. Die MMX-Befehle wurden so implementiert und optimiert, daß eine möglichst hohe Performance erreicht wird.

17.1 Neue MMX-Datenformate

Mit der MMX-Technologie hat Intel vier »neue« Datenformate definiert. Sie heißen PackedByte, PackedWord, PackedDoubleWord und QuadWord. In Wirklichkeit handelt es sich dabei eigentlich um keine neuen Datentypen. So stellt ein PackedByte lediglich ein Array[0..7] of Bytes dar, ein PackedWord ein Array[0..3] of Words, ein PackedDoubleWord ein Array[0..1] of DoubleWords, um im Pascal-Stil zu sprechen, und ein QuadWord ist eine LongInt, wie sie die FPU-Befehle schon längst kennen.

Was aber alle diese neuen Datentypen gemeinsam haben, sind: 64 Bits zur Codierung des Datums. Zu Einzelheiten über die Darstellung der neuen Datentypen siehe das Kapitel »Darstellung von Zahlen im Coprozessorformat«.

Was ist nun das Besondere an diesen neuen Datentypen? Die Antwort lautet: eigentlich gar nichts! Lassen wir für den Moment das QuadWord außen vor, so verhält sich jedes Element eines »gepackten Feldes« gleich und wie die Basistypen: Alle acht Bytes eines PackedByte sind echte Bytes, alle vier Worte des PackedWords sind Worte und die beiden Doublewords des PackedDoubleWord sind zwei echte Doppelworte. Das heißt: sie können vorzeichenbehaftet sein oder vorzeichenlos!

Warum also »PackedBytes« & Co.? Unter MMX wird mit Datenstrukturen gearbeitet! Die Parameter der MMX-Befehle stellen nicht einzelne Daten dar, sondern Felder von Daten, eben die PackedXXXX. Warum das so ist, werden wir gleich sehen.

Das QuadWord ist eigentlich nur eine andere Bezeichnung für ein Feld von 64 Bits. Man hätte es auch LongInt nennen können, hätte dann aber suggeriert, daß das QuadWord tatsächlich eine Zahl, ein Datum ist. Das ist es aber nicht, wie wir noch sehen werden!

17.2 Die »neuen« MMX-Register

Da selbst die neuesten Prozessoren der P5- bzw. P6-Reihe »nur« 32 Bit breite Allzweckregister besitzen, können die neuen Daten nicht in diesen Registern bearbeitet werden. Es wurden also, um MMX zu ermöglichen, neue Register notwendig. Hier folgt gleich eine gute und eine schlechte Nachricht. Die gute: Es gibt acht neue Register, in denen die MMX-Befehle ablaufen. Die schlechte: Sie kennen die Register schon.

Intel hat ein wenig nachgedacht und festgestellt, daß man schon bestehende Register nutzen könnte, die ein etwas stiefmütterliches Dasein führen. So wird von der FPU, also der Fließkommaeinheit, recht selten und nur in Anwendungen Gebrauch gemacht, bei denen Realzahlen verwendet und bearbeitet werden sollen. Dies ist auch in den meisten Anwendungsprogrammen, selbst bei rechenintensiven Programmen wie Tabellenkalkulationen oder Grafikprogrammen, ja selbst in FIBU- oder anderen kaufmännischen Programmen nicht oder nicht häufig der Fall – die Nutzung der FPU ist eher eine Domäne wissenschaftlicher Programme. (Im kaufmännischen Bereich wird aufgrund von Genauigkeit mit Integers gearbeitet, die so konstruiert werden, daß keine Nachkommastellen notwendig sind. So wird z.B. jede Währung in Einheiten eines 10.000stel angegeben, eine DM also als Vielfaches von 1/100 Pfennig. Dann reicht einfach Integer-Arithmetik vollkommen aus, ja ist in vielerlei Hinsicht sogar der Fließkommarechnung überlegen!)

Intel hat nun die brachliegenden Register der FPU für die Nutzung der MMX-Technologie zweckentfremdet. Mit anderen Worten: Sie können MMX nur *anstelle* der FPU einsetzen, nicht etwa zusammen mit ihr, wollen Sie nicht heilloses Durcheinander erzeugen oder sollten Sie nicht ganz genau wissen, was Sie an welcher Stelle tun! Die Register wurden zwar neu bezeichnet, als MM0 bis MM7. Dabei handelt es sich aber nur um Aliasnamen der FPU-Register R0 bis R7. Doch etwas genauer:

Die MMX-Register M0 bis M7 sind de facto die Bits 0 bis 63, also die Mantissen, der FPU-Register R0 bis R7. Bitte beachten Sie, daß es sich tatsächlich um die Hardwareregister handelt, die Sie ansprechen, nicht etwa um die Stackregister! Denn die FPU arbeitet ja, wie Sie wis-

sen, mit einem dynamischen Rechenstack, der aus den Stackregistern ST0 bis ST7 besteht. Dynamisch heißt in diesem Zusammenhang, daß ein bestimmtes Stackregister nicht immer das gleiche Hardwareregister repräsentiert – welches Hardwareregister der FPU-Einheit der TOS (ST0) ist, entscheidet ja das TOS-Feld im Statuswort der FPU-Einheit. Steht dort eine 0, so ist R0 der TOS, R1 ST1 und R7 ST7. Steht im TOS-Feld der Wert 5, so ist R5 der TOS (ST0), R6 ST1 und R4 ST7. Aber das wissen Sie bereits alles von weiter oben. Während also mit den FPU-Befehlen je nach Wert im TOS-Register mit dem gleichen Befehl (z.B. FADD ST0, ST1) dynamisch unterschiedliche Hardwareregister angesprochen werden, wird bei den MMX-Befehlen statisch immer das angegebene Hardwareregister angesprochen.

Die Technik einer dynamischen Rechenstackverwaltung hat sicherlich nur Sinn, wenn man die Register als Teil eines Stapels für die »umgekehrt polnische Notation« benutzt, über die FPU-Berechnungen ablaufen. So können z.B. zwei Zahlen addiert werden, wie es auch »von Hand« im täglichen Leben passiert: Aus zwei Zahlen wird eine – die Summe. Sie nimmt auch nur noch einen Speicherplatz, ein Register, ein. Das zweite, für die Addition notwendige Register ist wieder frei und kann neu genutzt werden, ohne den verbliebenen »Müll« entsorgen zu müssen. Für die Zwecke der MMX-Technologie hat die Dynamik dagegen nicht nur keine Bedeutung, weil z.B. Additionen hier nutzungsbedingt anders ablaufen sollen, sondern sie ist sogar eher verunsichernd. Somit hat man auf einen dynamischen Zugriff auf die Register über die Angabe eines TOS (top of Stack; Zeiger auf den Beginn des Registerstapels) bei MMX verzichtet.

Anders gesagt: Das TOS-Feld im Statuswort hat bei der MMX-Technologie keine Bedeutung – und nicht nur das! Alle anderen Register oder Registerteile, die Informationen enthalten, die für die Arbeit mit Realzahlen wichtig sind, sind bei MMX nicht relevant oder haben eingeschränkte oder andere Bedeutungen. So z.B. die Tag-Felder im Tag-Wort. Diese Felder beinhalten im Falle der FPU-Nutzung der Register ja die Information, ob das im dazugehörigen Register stehende Datum gültig ist, eine Null oder eine Ausnahme (NaN) darstellt oder ob das Register gar leer ist. Im Falle von MMX signalisieren diese Felder nur noch, ob das Register leer ist oder nicht. Denn als Wert kann ja nur einer der neuen Datentypen enthalten sein. Andere Beispiele sind die Exponenten- oder Vorzeichenfelder, die unter MMX ebenfalls keine Bedeutung haben (so gelten die Bits 79 bis 65 aller Register, in denen diese Informationen enthalten sind, unter MMX als reserviert!).

Wundert sich nun noch irgend jemand, daß für die MMX-Technologie also nicht der Prozessor zuständig ist, sondern der Coprozessor? In der Tat ist die MMX-Technologie ein Aufbohren des Befehlssatzes des Coprozessors (oder besser gesagt: der FPU-Unit) und eine Anpassung von dessen Fähigkeiten an die neuen Befehle.

17.3 »Herumwickeln« und »Sättigen«

Wichtig ist auch ein weiteres Feature der MMX-Technologie. Von den Befehlen, die mit den Allzweckregistern des Prozessors arbeiten (ADD, SUB etc.), kennen Sie das Problem von Über- und Unterlauf. Das bedeutet, daß nach arithmetischen Manipulationen die gültigen Wertebereiche für das Datum über- bzw. unterschritten werden können. Signalisiert wird das in den genannten Fällen durch das Setzen und Löschen verschiedener Flags, so des Carry-Flags, des Overflow-Flags und des Adjust-Flags. Je nach verwendetem Datum – vorzeichenlose oder vorzeichenbehaftete Bytes, Worte oder Doppelworte oder BCDs – hat also anhand dieser Flags nach der Berechnung eine Interpretation des Ergebnisses zu erfolgen: Ein gesetztes Flag signalisiert, daß das Ergebnis der Berechnung nur bedingt richtig ist und korrigiert werden muß.

Interessiert einen diese Information nicht, unterbleibt also eine nachträgliche Interpretation des Zustandes der Flags, so führt eine Überschreitung des Wertebereichs zu einem Ergebnis, das modulo Wertebereich ist. Das heißt, die Addition von 48 zu 240 bei vorzeichenlosen Bytes liefert 288, was über dem Wertebereich von 256 für Bytes liegt und daher 32 (= 288 mod 256) ergibt. Diese Modulo-Bildung entspricht bildlich dem »Zusammenkleben« des Zahlenstrahls an den Bereichsenden, bei Bytes also dem Zusammenkleben von 255 und 0: Nach 255 kommt eben die »0«. Das »Zusammenkleben« erfolgt, weiterhin bildlich gesprochen, indem man das Ende des Strahls zum Anfang »herumwickelt«. Aus diesem Grunde nennt man Berechnungen, denen solche Gesetze zugrunde liegen, auch Wrap-around Calculations (»herumgewickelte« Berechnungen).

Die MMX-Technologie dient vor allem zur Unterstützung von Multimedia-Anwendungen oder zur Kommunikation. In diesen Fällen bewegen sich die Daten in der Regel innerhalb der gültigen Wertebereiche. Über- bzw. Unterläufe haben keinen realen Hintergrund. (Was würde auch ein Überlauf aussagen, wenn z.B. bei der Berechnung einer Farbe für ein Farbattribut im 256-Farben-Modus der Wert 387 herauskäme? Es gibt nur 256 Farben – und die Berechnung muß auf diese

Randbedingungen abgebildet werden: entweder, indem die 387 modulo 256 genommen wird, was dem »Herumwickeln« entspricht, oder indem der maximal gültige Wert, hier 255, angenommen wird.) Aus diesem Grund unterstützt MMX auch keinen Über- und Unterlauf, sondern den *Wrap-around Mode*.

Aber auch den zweiten Fall des Beispiels unterstützt die MMX-Technologie. Da in den Fällen, in denen der Wertebereich über- bzw. unterschritten wird, das Ergebnis durch den jeweiligen Maximal- bzw. Minimalwert ersetzt werden kann, spricht man in diesem Fall vom »Sättigen« der Berechnung. In Pascal-ähnlicher Notation läßt sich also der *Saturation Mode* wie folgt darstellen:

```
temp := calculation(value1, value2);
if temp < minvalue then temp := minvalue;
if temp > maxvalue then temp := maxvalue;
result := temp;
```

Hurra! Auf diese Weise können niemals mehr die Wertebereiche über- oder unterschritten werden, Ausnahmebehandlungen mittels Exceptions und/oder Flagabfrage gehören der Vergangenheit an!

Aber das hat Konsequenzen! Bei Berechnungen dürfen Sie nicht mehr davon ausgehen, daß Sie irgendwie über Über- oder Unterläufe informiert werden. Entweder erfolgen sie gar nicht, weil im *Saturation Mode* gearbeitet wird, oder sie werden – im *Wrap-around Mode* – nicht angezeigt, weil ja bis zu acht voneinander unabhängige Werte gleichzeitig bearbeitet werden und solche zusätzlichen Informationen mit den bekannten Flags gar nicht angezeigt werden können. MMX-Befehle verändern die Flags des Prozessors und des Coprozessors nicht!

Natürlich gilt das alles sowohl für vorzeichenlose Daten wie auch für vorzeichenbehaftete. Dabei ergibt sich allerdings ein Problem, das wir weiter oben schon angesprochen haben, als wir generell über vorzeichenlose bzw. -behaftete Zahlen und die sie nutzenden Befehle gesprochen haben. Woran erkennt der (in diesem Fall Co-) Prozessor, ob das Byte $FD nun als –3 oder als 253 interpretiert werden soll? Denn schließlich würde nach einer Addition von 4 in beiden Fällen der Wert $01 herauskommen. Im ersten Fall ist er aber als +1 ohne Überlauf zu interpretieren, im zweiten als 1 mit Überlauf, da ja das Ergebnis 257 den maximal darstellbaren Bereich von 255 für vorzeichenlose Bytes überschreitet. Im Sättigungsmodus müßte daher im zweiten Fall zu 255 gesättigt werden, während im ersten Fall die Sättigung unterbleibt.

Die Antwort auf das Problem lautet: Nachdem hier nicht wie bei den arithmetischen Befehlen der Allzweckregister mit verschiedenen Flags und Flagkombinationen gearbeitet und das Ergebnis entsprechend interpretiert werden kann, müssen unterschiedliche Befehle dafür herhalten. Halten wir also fest: Für die arithmetischen Befehle, die im Rahmen der MMX-Technologie eingesetzt werden, gibt es analoge Befehle für Berechnungen mit jeweils vorzeichenbehafteten und vorzeichenlosen Daten im Sättigungsmodus und im *Wrap-Around*-Modus.

17.4 Neue MMX-Befehle

Wenn die neue Technik wirklich hält, was sie verspricht, hat das Konsequenzen für die Datenbehandlung. Denn dann ist ein PackedByte-Datum nichts anderes als ein Feld von acht Bytes, die vorzeichenbehaftet oder vorzeichenlos sein können. Also müssen die MMX-Befehle auch unabhängig voneinander auf acht Bytes erfolgen. Da dies im Rahmen eines einzigen Registers des Coprozessors erfolgt, werden acht Bytes *gleichzeitig* bearbeitet. Analoges gilt für PackedWords und PackedDoubleWords. Genau das ist es, was die MMX-Technologie so interessant macht. Denn auf diese Weise können pro Zeiteinheit bis zu achtmal mehr Daten verarbeitet werden, als es ohne MMX möglich ist. Intel nennt dies SIMD Execution Model (Single Instruction, Multiple Data).

Doch Vorsicht! Das Ganze wird sich natürlich nur in den Fällen auswirken, in denen wirklich große Mengen von Daten auf die gleiche Weise bearbeitet werden müssen! Also wenn z.B. im Rahmen von Bildschirmausgaben, Grafikberechnungen, Kommunikation oder Multimedia viele gleichartige Daten mit den gleichen Befehlen bearbeitet werden sollen. Absoluter Unsinn ist die Anwendung von MMX z.B. zur Steuerung einer Schleife oder zur Berechnung einiger weniger Daten, da dies viel zu aufwendig und somit kontraproduktiv wäre!

Die MMX-Technologie arbeitet also mit »gepackten« Daten. Daher beginnen (fast) alle MMX-Befehle mit »P«, so wie die FPU-Befehle mit »F« beginnen.

Der MMX-Befehlssatz umfaßt Befehle zum

- arithmetischen Manipulieren der Daten
- logischen Manipulieren der Daten
- Datenaustausch
- Datenvergleich
- Datenkonversion
- MMX-Status

Behalten Sie bitte im Hinterkopf, daß die einzigen Datentypen, die zu tatsächlichen Berechnungen verwendet werden, die Datentypen PakkedByte, PackedWord und PackedDoubleWord sind. Alle arithmetischen Manipulationen oder Vergleichsberechnungen und auch die Datenkonvertierungen laufen ausschließlich mit diesen Typen ab. QuadWords finden keine Verwendung!

Die Domäne der QuadWords sind der Datenaustausch und die logischen Operationen! Denn weil immer registerweise mit bis zu acht Daten gleichzeitig gearbeitet wird, müssen diese Daten auch »auf einmal« in die Register geladen oder aus ihnen entfernt werden. Das erfolgt in 64-Bit-Einheiten, eben den QuadWords. Wenn »logisch« gearbeitet wird, so werden auch keine Bytes, Worte oder Doppelworte eingesetzt, sondern mehr oder weniger viele, voneinander unabhängige Bits, so daß bei den logischen Manipulationen 64 Bits betrachtet werden, keine PackedXXXX. Diese 64 Bits nennt Intel QuadWord. (Wenn Sie mich fragen, hätte man auf diese Definition auch verzichten können! Aber sei es drum!)

Die arithmetischen Befehle umfassen lediglich drei der vier Grundrechenarten: Addition und Subtraktion sowie Multiplikation.

PADDB, PADDW, PADDD, PADDSB, PADDSW, PADDUSB, PADDUSW

Die Befehlsgruppe Packed Addition umfaßt sieben verschiedene Befehle. So wird im *Wrap-around-Modus* die Addition von Bytes (PADDB; P*acked Addition of Bytes*), Worte (PADDW; *Packed Addition of Words*) und Doppelworte (PADDD; *Packed Addition of Doublewords*) unterstützt. Im *Sättigungsmodus* können vorzeichenbehaftete *Bytes* (PADDSB; *Packed Addition and Saturation of Bytes*) und *Worte* (PADDSW; *Packed Addition and Saturation of Words*) sowie deren vorzeichenlose Pendants (PADDUSB; *Packed Addition Unsigned and Saturation of Bytes* und PADDUSW; *Packed Addition Unsigned and Saturation of Words*) addiert werden. Warum die *Packed Addition* im *Sättigungsmodus* nicht auch mit vorzeichenbehafteten oder vorzeichenlosen Doppelworten ermöglicht wird, ist mir, ehrlich gesagt, ein Rätsel! Vielleicht, weil Doppelworte bei Multimedia und Kommunikation nicht *die* herausragende Rolle spielen!

PSUBB, PSUBW, PSUBD, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW

Packed Subtraction ist die zur Addition analoge Befehlsgruppe zum Subtrahieren von Daten. Es gibt analog im *Wrap-around-Modus die* Subtraktion von Bytes (PSUBB; *Packed Subtraction of Bytes*), *Worten* (PSUBW; *Packed Subtraction of Words*) und Doppelworten (PSUBD; *Packed Subtraction of Doublewords*). Im *Sättigungsmodus* können vorzeichenbehaftete Bytes (PSUBSB; *Packed Subtraction and Saturation of Bytes*) und Worte (PSUBSW; *Packed Subtraction and Saturation of Words*) sowie deren vorzeichenlose Pendants (PSUBUSB; *Packed Subtraction Unsigned and Saturation of Bytes* und PSUBUSW; *Packed Subtraction Unsigned and Saturation of Words*) subtrahiert werden.

Bei der gepackten Addition und Subtraktion ist zu beachten, daß die acht Bytes, vier Worte bzw. zwei Doppelworte unabhängig voneinander sind! Der Befehl *PADDSB MM0, MM1* z.B. kann in Pascal-ähnlicher Schreibweise wie folgt interpretiert werden:

```
MM0[07..00] := SaturateByte(ADD(MM0[07..00], MM1[7..00]));
MM0[15..08] := SaturateByte(ADD(MM0[15..08], MM1[15..08]));
MM0[23..16] := SaturateByte(ADD(MM0[23..16], MM1[23..16]));
MM0[31..24] := SaturateByte(ADD(MM0[31..24], MM1[31..24]));
MM0[39..32] := SaturateByte(ADD(MM0[39..32], MM1[39..32]));
MM0[47..40] := SaturateByte(ADD(MM0[47..40], MM1[47..40]));
MM0[55..48] := SaturateByte(ADD(MM0[55..48], MM1[55..48]));
MM0[63..56] := SaturateByte(ADD(MM0[63..56], MM1[63..56]));
```

Analoges gilt z.B. für die Subtraktion vorzeichenloser Worte im Sättigungsmodus – *PSUBUSW MM4, MM7*:

```
MM4[15..00] := SaturateByte(SUB(MM4[15..00], MM7[15..00]));
MM4[31..16] := SaturateByte(SUB(MM4[31..16], MM7[31..16]));
MM4[47..32] := SaturateByte(SUB(MM4[47..32], MM7[47..32]));
MM4[63..48] := SaturateByte(SUB(MM4[63..48], MM7[63..48]));
```

oder die Subtraktion von Doppelworten im Wrap-around-Modus, wie z.B. in *PSUBD MM0, MM6*

```
MM0[31..00] := SUB(MM0[31..00], MM6[31..00]);
MM0[63..32] := SUB(MM0[63..32], MM6[63..32]);
```

Bitte beachten Sie, daß in keinem Fall irgendein Flag verändert oder auf das Statuswort des Coprozessors zugegriffen wird! Es erfolgt auch im Wrap-around-Modus keinerlei Hinweis auf einen Über- oder Unterlauf!

PMULLW, PMULHW

Multiplikationen haben, verglichen mit Additionen oder Subtraktionen, einen gravierenden Nachteil, zumindest was den gestreßten Programmierer angeht: Sie können Werte erzeugen, die nicht mehr nur ggf. unter Zuhilfenahme eines Überlaufflags dargestellt werden können. Denn das Ergebnis der Multiplikation zweier Worte füllt den Wertebereich von Doppelworten. Da in den MMX-Registern parallel mit bis zu vier Worten gearbeitet wird, ist eine Multiplikation nicht ganz problemlos: Wie können die vier Doppelworte, die bei der Multiplikation von vier Worten mit vier Worten entstehen, in die MMX-Register eingetragen werden?

Direkt geht es nicht: Die MMX-Register sind nur 64 Bits breit. Also muß man zwei Register heranziehen. Nun gibt es generell zwei Möglichkeiten, diese zu nutzen. Möglichkeit 1: Jedes Register faßt zwei Doppelworte. Das Problem dabei ist dann, daß die Quellregister eine andere »Einteilung« aufweisen als die Zielregister: Die einen enthiel-

ten Worte, die anderen Doppelworte. Zweitbeste Möglichkeit! Möglichkeit 2: Die beiden Zielregister fassen jeweils ein Wort des Ergebnis-Doppelwortes: Das eine die niederwertigen Worte, das andere die höherwertigen. Dies hat den Vorteil, daß alle beteiligten Register die gleiche »Einteilung« hätten.

Auch die MMX-Befehle müssen sich an die Intel-Konvention halten, die besagt, daß das Ergebnis der Berechnung in das Zielregister kommt, dem für die Berechnung ein Operand entnommen wurde. Also z.B. ADD EBX, EAX; hier wird zum Registerinhalt von EBX der von EAX addiert und das Ergebnis in EBX abgelegt. Dies hat auch mit den MMX-Befehlen so zu sein. Schon allein aus diesem Grunde wurde Möglichkeit 2 realisiert. Ein zweiter Grund liegt darin, daß es immer nur einen Zieloperanden geben kann. Für Möglichkeit 1 bräuchten wir zwei Operanden. Auch für Möglichkeit 2 bräuchten wir zwei, wenn Intel nicht zwei Befehle spendiert hätte, mit denen man eine »richtige« Berechnung nach Intel-Manier durchführen kann. Das geht so:

```
Temp[031..000] := MUL(MMx[15..00], MMy[15..00])
Temp[063..032] := MUL(MMx[31..16], MMy[31..16])
Temp[095..064] := MUL(MMx[47..32], MMy[47..32])
Temp[127..096] := MUL(MMx[63..48], MMy[63..48])
```

Je nachdem, ob Sie sich nun für die niederwertigen Anteile des Ergebnisses interessieren oder für die höherwertigen, verwenden Sie einen der beiden Befehle PMULLW (*Packed Multiply Low Word*) oder PMULHW (*Packed Multiply High Word*), hier im Beispiel sei es *PMULLW MMx, MMy*:

```
MMx[15..00] := LowWord(Temp[031..000]);
MMx[31..16] := LowWord(Temp[063..032]);
MMx[47..32] := LowWord(Temp[095..064]);
MMx[63..48] := LowWord(Temp[127..096]);
```

Analoges gilt natürlich für PMULHW. Und noch eins: Multiplikationen lassen sich nur mit Worten, nicht aber mit Bytes durchführen. Warum nicht? Ich weiß es nicht ....

PMADDWD PMADDDW ist eine Kombination einer Multiplikation und einer Addition. Wie aus dem Mnemonic PMADD**WD** bereits hervorgeht, geht diese Berechnung mit einer Konversion von Worten in Doppelworte einher. Diese resultiert daraus, daß nun einmal die Multiplikation zweier Worte miteinander zu einem Doppelwort führt.

PMADDWD MMx, MMy multipliziert zunächst jeweils die Worte des ersten Operanden mit denen des zweiten und erzeugt damit vier Doppelworte:

```
Temp[031..000] := MUL(MMx[15..00], MMy[15..00])
Temp[063..032] := MUL(MMx[31..16], MMy[31..16])
Temp[095..064] := MUL(MMx[47..32], MMy[47..32])
Temp[127..096] := MUL(MMx[63..48], MMy[63..48])
```

Jeweils zwei aufeinanderfolgende Doppelworte werden dann addiert, so daß als Ergebnis der Operation zwei Doppelworte entstehen, die der Rechenvorschrift

R0 = (X1 * Y1) + (X0 * Y0); R1 = (X3 * Y3) + (X2 * Y2)

gehorchen:

```
MMx[31..00] := ADD(Temp[063..032], Temp[031..000]);
MMx[63..32] := ADD(Temp[127..064], Temp[095..064]);
```

Wer Berechnungen mit Daten durchführen will, möchte auch Vergleiche von Daten anstellen können! Auch das kann mit der MMX-Technologie durchgeführt werden, wenn es auch, verglichen mit den Befehlen für die Allzweckregister, nur sehr wenige Möglichkeiten gibt, nämlich zwei:

PCMPEQB, PCMPEQW, PCMPEQD PCMPGTB, PCMPGTW, PCMPGTD

- Vergleich, ob die Bytes, Worte oder Doppelworte in den beiden Operanden gleich groß sind und
- Vergleich, ob die Bytes, Worte oder Doppelworte im Zieloperanden (erster Operand) größer sind als im zweiten Operanden (Quelloperand).

In beiden Fällen können nicht, wie im Falle der Allzweckregisterbefehle, Flags bemüht werden, um das Ergebnis des Resultates anzuzeigen! Auch läßt sich das nicht über das Statuswort des Coprozessors realisieren. Daher muß das Ergebnis im Zieloperanden codiert werden. Führt also ein Vergleich zu einem wahren Ergebnis, sind also bei *PCMPEQx MMx, MMy* beide Daten gleich oder ist bei *PCMPGTx MMx, MMy* das Datum im Zielregister MMx größer als das im Quellregister MMy, so wird in das Zielregister an die betreffende Position $FF im Falle von Bytes, $FFFF im Falle von Worten und $FFFFFFFF im Falle von Doppelworten geschrieben. Andernfalls wird »0« eingetragen:

```
IF MMx[15..00] > MMy[15..00] THEN MMx[15..00] := $FFFF
                             ELSE MMx[15..00] := $0000;
IF MMx[31..16] > MMy[31..16] THEN MMx[31..16] := $FFFF
                             ELSE MMx[31..16] := $0000;
IF MMx[47..32] > MMy[47..32] THEN MMx[47..32] := $FFFF
                             ELSE MMx[47..32] := $0000;
IF MMx[63..48] > MMy[63..48] THEN MMx[63..48] := $FFFF
                             ELSE MMx[63..48] := $0000;
```

```
IF MMx[31..00] = MMy[31..00] THEN MMx[31..00] := $FFFFFFFF
                             ELSE MMx[31..00] := $00000000;
IF MMx[63..32] = MMy[63..32] THEN MMx[63..32] := $FFFFFFFF
                             ELSE MMx[63..32] := $00000000;
```

Es gibt auch ein paar sinnvolle Konvertierungsbefehle. So kann man unter bestimmten Voraussetzungen Worte in Bytes »packen« oder Doppelworte in Worte. (Das »Packen« eines QuadWords in ein Doppelwort macht aus dem eingangs Gesagten keinen Sinn!)

PACKSSWB, PACKSSDW, PACKUSWB

Betrachten wir ein Wort. Wenn wir es in ein Byte »packen« wollen, so geht das nur, wenn eine von zwei Bedingungen erfüllt ist. Entweder, das »Wort« benutzt nicht alle Bits zur Codierung der Information – ähnlich wie die BCDs, die man ja auch »packen kann« – oder der Wert des Wortes ist nicht außerhalb des Bereiches eines Bytes. Den ersten Fall können wir mit den BCDs als erledigt betrachten! Das bedeutet aber für den zweiten Fall, daß es eine Rolle spielt, ob mit oder ohne Vorzeichen gearbeitet wird. Soll also ein vorzeichenbehaftetes Worte auf ein vorzeichenbehaftetes Byte abgebildet werden, so darf der Wert des Wortes den Bereich –128 bis 127 nicht überschreiten.

Was geschieht aber, wenn genau das der Fall ist? Dann kann, entsprechend der MMX-Technologie, das Byte »gesättigt« werden. Das heißt, alle Werte des Worts, die -128 unterschreiten, werden auf »-128« gesetzt und alle Werte, die 127 überschreiten, auf »127«. Bewerkstelligt wird das durch den Befehl PACKSSWB, *Pack with Signed Saturation Word to Byte*. Es gibt auch den analogen Befehl für Doppelworte: *Pack with Signed Saturation Doublewords to Words*: PACKSSDW.

Bei PACKSSDW und PACKSSWB wird also das Vorzeichen des Ausgangswertes berücksichtigt und in den Endwert übertragen. Es gibt aber auch eine Alternative: PACKUSWB, *Pack with Unsigned Saturation Word to Byte*. Zwar ist auch hier der Ausgangswert, ein Wort, vorzeichenbehaftet. Aber es erfolgt eine Sättigung auf ein vorzeichenloses Byte. Warum es kein analoges *Pack with Unsigned Saturation Doubleword to Word* gibt, entzieht sich meinen Verständnis! Gibt es im Multimedia- und Kommunikationsbereich tatsächlich keinen Bedarf dafür?

Nun aber ein kleines Problem: Aus vier Worten eines Registers bzw. aus zwei Doppelworten machen wir mit den Befehlen vier Bytes bzw. zwei Worte. Was passiert mit den restlichen, frei gewordenen Bits der Register? Antwort: Sie werden dazu genutzt, um weitere vier Worte bzw. zwei Doppelworte zu packen – und zwar aus dem zweiten Operanden.

Daher einmal kurz die Pascal-ähnliche Notation dessen, was bei diesen Befehlen abläuft, zunächst am Beispiel *PACKSSDW MMx, MMy* erläutert:

```
IF MMx[31..00] > $00007FFF THEN MMx[15..00] := $7FFF
   ELSE IF MMx[31..00] < $FFFF8000 THEN MMx[15..00] := $8000
      ELSE MMx[15..00] := WORD(MMx[31..00]);
IF MMx[63..32] > $00007FFF THEN MMx[31..16] := $7FFF
   ELSE IF MMx[63..32] < $FFFF8000 THEN MMx[31..16] := $8000
      ELSE MMx[31..16] := WORD(MMx[63..32]);
IF MMy[31..00] > $00007FFF THEN MMx[47..32] := $7FFF
   ELSE IF MMy[31..00] < $FFFF8000 THEN MMx[47..32] := $8000
      ELSE MMx[47..32] := WORD(MMy[31..00]);
IF MMy[63..32] > $00007FFF THEN MMx[63..48] := $7FFF
   ELSE IF MMy[63..32] < $FFFF8000 THEN MMx[63..48] := $8000
      ELSE MMx[63..48] := WORD(MMy[63..32]);
```

Der entsprechende Befehl für vorzeichenlose Sättigung, PACKUSWB, funktioniert so:

```
IF MMx[15..00] > $00FF THEN MMx[07..00] := $FF
   ELSE IF MMx[15..00] < $0000 THEN MMx[07..00] := $00
      ELSE MMx[15..00] := BYTE(MMx[15..00]);
IF MMx[31..16] > $00FF THEN MMx[15..07] := $FF
   ELSE IF MMx[31..16] < $0000 THEN MMx[15..07] := $00
      ELSE MMx[15..07] := BYTE(MMx[31..16]);
IF MMx[47..32] > $00FF THEN MMx[23..16] := $FF
   ELSE IF MMx[47..32] < $0000 THEN MMx[23..16] := $00
      ELSE MMx[23..16] := BYTE(MMx[47..32]);
IF MMx[63..48] > $00FF THEN MMx[31..24] := $FF
   ELSE IF MMx[63..48] < $0000 THEN MMx[31..24] := $00
      ELSE MMx[31..24] := BYTE(MMx[63..48]);
IF MMy[15..00] > $00FF THEN MMx[39..32] := $FF
   ELSE IF MMy[15..00] < $0000 THEN MMx[39..32] := $00
      ELSE MMx[39..32] := BYTE(MMy[15..00]);
IF MMy[31..16] > $00FF THEN MMx[47..40] := $FF
   ELSE IF MMy[31..16] < $0000 THEN MMx[47..40] := $00
      ELSE MMx[47..40] := BYTE(MMy[31..16]);
IF MMy[47..32] > $00FF THEN MMx[55..48] := $FF
   ELSE IF MMy[47..32] < $0000 THEN MMx[55..48] := $00
      ELSE MMx[55..48] := BYTE(MMy[47..32]);
IF MMy[63..48] > $00FF THEN MMx[63..56] := $FF
   ELSE IF MMy[63..48] < $0000 THEN MMx[63..56] := $00
      ELSE MMx[63..56] := BYTE(MMy[63..48]);
```

Wer einpacken kann, muß auch auspacken können! Betrachten wir also zunächst einmal, welche Befehle es gibt:

PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ

Unpack High Bytes to Words, *Unpack High Words to Doublewords* und *Unpack High Doublewords to Quadword* sind die ersten drei von sechs Befehlen, die zum »Entpacken« vorgesehen sind. Nachdem das Packen von Daten eine Reduktion bewirkt, muß umgekehrt das Entpacken eine Inflation bewirken. Daher verwundert es uns nicht, wenn in irgendeiner Weise nur Teile eines gepackten Datums verwendet werden.

Im Falle des »Packens« wurden zwei Operanden in einen »gepackt«. Heißt das nun, daß im umgekehrten Falle ein Operand in zwei »entpackt« wird? Nein! Denn dazu müßte der Befehl zwei Zieloperanden besitzen, was laut Intel nicht möglich ist. Also kann die Inflation nur erfolgen, wenn ein halber Operand in einen ganzen aufgebläht wird. Zusammen mit

PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ

Unpack Low Bytes to Words, Unpack Low Words to Doublewords und Unpack Low Doublewords to Quadword kann dann das Gewünschte erreicht werden.

Was passiert nun bei allen sechs Befehlen genau? Schauen wir uns zunächst *PUNPCKLDQ MMx, MMy* an (weil es weniger Schreibarbeit für mich bedeutet!):

```
MMx[31..00] := MMx[31..00];
MMx[63..32] := MMy[31..00];
```

Das »Entpacken« entpuppt sich also gar nicht als »Inflation« eines Doppelworts zu einem Quadwort! Es ist vielmehr das »Mischen« von zwei Doppelworten zu einem Quadwort. Der Buchstabe »L« im Mnemonic signalisiert hierbei, daß die niederwertigen Doppelworte aus den beiden Operanden verwendet werden. Es geht auch mit den beiden höherwertigen, wie uns *PUNPCKHDQ MMx, MMy* zeigt:

```
MMx[31..00] := MMx[63..32];
MMx[63..32] := MMy[63..32];
```

Also: Unter »Entpacken« versteht Intel offensichtlich, zumindest was MMX betrifft, das Mischen zweier Daten zu einem neuen Datum nach der Formel:

```
Word := SHL(Byte2, 1) + Byte1
DWord := SHL(Word2, 2) + Word1
QWord := SHL(DWord2, 4) + DWord1
```

Das führt (ich kann mir diesmal die Schreibarbeit nicht ersparen, um das sog. »interleaving« zu demonstrieren) zu folgendem, wenn man einmal die niederwertigen Bytes mit *PUNPCKLBW MMx, MMy* zu Worten »entpackt«:

```
MMx[07..00] := MMx[07..00];
MMx[15..08] := MMy[07..00];
MMx[23..16] := MMx[15..08];
MMx[31..24] := MMy[15..08];
MMx[39..32] := MMx[23..16]
MMx[47..40] := MMy[23..16];
MMx[55..48] := MMx[31..24];
MMx[63..56] := MMy[31..24];
```

Analoges gilt natürlich auch für *PUNPCKHBW MMx, MMy*:

```
MMx[07..00] := MMx[15..08];
MMx[15..08] := MMy[15..08];
MMx[23..16] := MMx[31..24];
MMx[31..24] := MMy[31..24];
MMx[39..32] := MMx[47..40]
MMx[47..40] := MMy[47..40];
MMx[55..48] := MMx[63..56];
MMx[63..56] := MMy[63..56];
```

Welchen Sinn machen also die »Entpackungsbefehle«? Zunächst fällt mir spontan ein recht interessantes Anwendungsgebiet ein, das, erheblich vereinfacht und auf das Wesentliche reduziert, so aussehen könnte (MOVD und MOVQ bekommen wir gleich; es sind Ladebefehle!):

```
MOV       EAX, $F0F0F0F0           ; Attribut
MOVD      MM1, EAX                 ; 4 mal Attribut
MOV       EAX, Offset TextBuffer   ; Quelle: Text
MOV       EBX, Offset ScreenBuffer ; Ziel: Bildschirm
MOV       ECX, TextSize            ; Stringgröße
SHR       ECX, 2 ; immer 4 Zeichen
L1:       MOVD  MM0, DS:[EAX]      ; 4 Zeichen lesen
PUNPCKLBW MM0, MM1                 ; mit Attribut mischen
MOVQ      ES:[EBX], MM0            ; 8 Zeichen schreiben
ADD       EAX, 4 ; Zeiger erhöhen
ADD       EBX, 8 ; dito
LOOP      L1    ; zurück zur Schleife
```

Diese Schleife, die ein Bildschirmattribut mit dem Zeichen aus einem auf dem Bildschirm auszugebenden Textstring mischt und tatsächlich auszugibt, ist mindestens achtmal schneller als die Lösung, die ohne MMX möglich ist (wenn man einmal von bestimmten Optimierungen absieht!).

Auch eine andere Lösung fällt mir spontan ein: Denken Sie einmal an PMULHW und PMULLW. Wie könnte man tatsächlich vier Bytes mit vier Bytes zu »echten« vier Worten multiplizieren?

```
MOV       EAX, Offset ByteArray1
MOV       EBX, Offset ByteArray2
MOV       EDX, Offset WordArray
MOV       ECX, ArraySize
SHR       ECX, 2 ; weil immer vier Daten auf einmal!
L2:       MOVD  MM2, [EAX]         ; vier Multiplikatoren in low MM2
MOVD      MM1, [EBX]               ; vier Multiplikanden in low MM1
MOVD      MM0, MM1                 ; die gleichen in low MM0
PMULHW    MM1, MM2                 ; high Produkt in low MM1
PMULLW    MM0, MM2                 ; low Produkt in low MM0
PUNPCKLBW MM0, MM1                 ; High-Low-Paare in MM0
MOVQ      [EDX]
ADD       [EAX], 4;
ADD       [EBX], 4
ADD       [EDX], 8;
LOOP      L2
```

Sie sehen also, daß die »Entpackungs«-Befehle durchaus sinnvoll und hilfreich sind, auch wenn die Wortwahl der Mnemonics in meinen Augen nicht sehr glücklich ist.

Es gibt übrigens keine Befehle, die beim Entpacken auch »sättigen«. Aber ist das nach allem, was wir über die Arbeitsweise der Entpackungsbefehle herausgefunden haben, noch ein Wunder?

PSLLW, PSLLD, PSRLW, PSRLD, PSRAW, PSRAD

Die Shift-Befehle unter MMX gleichen den Shift-Befehlen, die mit den Allzweckregistern möglich sind! Mit einer Ausnahme: Es ist kein Flag insolviert, wie beispielsweise das Carry-Flag im Falle der Allzweckregister! Ansonsten gibt es logisches Verschieben nach links (SLL; shift left logically), logisches Verschieben nach rechts (SRL; Shift Right Logically) und arithmetisches Verschieben nach rechts (SRA; Shift Right Arithmetically). Ein arithmetisches Verschieben nach links gibt es genausowenig wie im Falle der Allzweckregister, da das mit dem logischen Verschieben nach links, zumindest, was das Ergebnis betrifft, identisch ist. Insoweit nichts Neues!

PSLLQ, PSRLQ

Allerdings kann man mit den Shift-Befehlen nur Worte und Doppelworte verschieben. Bytes sind ebensowenig manipulierbar wie – Quadworte, wollte ich fast sagen. Aber letzteres stimmt nur teilweise. Denn Quadworte können zumindest logisch nach links und rechts verschoben werden (PSLLQ und PSRLQ), was den Bit-Charakter dieses Datums unter

streicht – Quadworte sind keine echten Zahlen! (Denn einzelne, *echte* 64-Bit-Integer lassen sich effektiver mit der FPU und den LongInts manipulieren!)

Ansonsten gibt es über die Shift-Befehle nichts weiter zu sagen. Sie arbeiten, wie gesagt, absolut analog zu den bereits bekannten Shift-Befehlen, nur daß sie eben vier Worte, zwei Doppelworte oder ein Quadwort gleichzeitig verschieben. Die freigewordenen Bits werden, wie bei den anderen Befehlen auch, mit »0« aufgefüllt.

Mit den Shift-Befehlen haben wir aber den Übergang von den arithmetischen Befehlen zu den bitorientierten Befehlen vorgenommen. Während die arithmetischen Befehle – und zumindest das arithmetische Verschieben von Bits hat als arithmetischer Befehl aufgefaßt zu werden – mit »echten« Zahlen arbeiten, also Bitpaketen, die als Wert zu interpretieren sind, arbeiten die bitorientierten Befehle mit einzelnen, voneinander unabhängigen Bits. Die »Zahlen« sind hier also als Bitfelder zu interpretieren. Aus genau diesem Grunde gibt es nur Befehle, die mit Quadworten arbeiten – PackedBytes, PackedWords und PackedDoubleWords spielen keine Rolle:

PAND, PANDN, POR, PXOR

Diese Bitmanipulationsbefehle unterscheiden sich in rein gar nichts von den Zwillingen AND, OR und XOR, mit denen bitweise Operationen in den Allzweckregistern möglich sind. Lediglich PANDN, *Packed And Not*, hat kein Pendant! Einzige Unterschiede: Es werden 64 Bits gleichzeitig bearbeitet, eben ein Quadwort, und es werden keine Flags verändert!

Zu PANDN läßt sich noch folgendes sagen: Es ist *nicht*, wie man zunächst anhand der Namensgebung zu erkennen glaubt, eine AND-Operation mit *anschließender* NOT-Operation! Vielmehr wird der erste Operand zunächst negiert und dann mit dem zweiten Operanden durch AND verknüpft:

```
x = y AND (NOT x);
```

Am besten läßt sich die Wirkung der Befehle auf einzelne Bits der Operanden in folgendem Schema darstellen:

Bit 1 \ Bit 2	0	1
PAND		
0	0	0
1	0	1
PANDN		
0	0	1
1	0	0
POR		
0	0	1
1	1	1
PXOR		
0	0	1
1	1	0

Fehlt eigentlich nur noch ein NOT-Pendant. Das gibt es allerdings nicht. Ich weiß auch nicht, warum. Aber man kann es über den PANDN-Befehl simulieren. Denn der kann ja einen der beiden Ope-

randen negieren. Also muß man den zweiten Operanden so wählen, daß in Verbindung mit der sich anschließenden AND-Operation das korrekte Ergebnis herauskommt. Das bedeutet:

```
»PNOT« = 1 AND (NOT x);
```

Also wird als erster Operand ein Register verwendet, das $FFFFFFFFFFFFFFFF enthält. Der zweite Operand wird dann durch PANDN negiert.

MOVD, MOVQ

Nun fehlt uns zu unserem Glück mit MMX eigentlich nur noch eines: Wie bekomme ich die Daten in die MMX-Register und wieder heraus? Dazu gibt es genau zwei Befehle: MOVD und MOVQ. MOVD kopiert vom Quelloperanden ein DWord, also 32 Bit, in den Zieloperanden. Es macht also dann Sinn, wenn nur 32 Bit bewegt werden müssen, z.B. im Entpackungsbeispiel zum Laden der vier Worte zur Multiplikation, oder wenn nur 32 Bit bewegt werden *können*, z.B. beim Austausch mit Allzweckregistern.

Folgerichtig kann MOVD *nicht* dazu eingesetzt werden, Daten zwischen MMX-Registern auszutauschen! Denn für die MMX-Register gibt es nur 64-Bit-Daten, so wie es für den Coprozessor nur TempReals gibt. Unterschiedliche Ladebefehle mit unterschiedlichen Optionen ändern an dieser Tatsache hier wie dort nichts! Noch etwas: Sobald Daten mit MOVD bewegt werden, ist immer nur das niederwertige Doppelwort, also die Bits 31 bis 0 des MMX-Registers, betroffen. Beim Laden eines MMX-Registers werden die Bits 63 bis 32 automatisch gelöscht, beim Speichern aus einem MMX-Register werden nur die Bits 31 bis 0 verwendet.

MOVQ dagegen bewegt alle 64 Bits eines MMX-Registers. Damit ist klar, daß dieser Befehl verwendet werden muß, wenn Daten zwischen MMX-Registern ausgetauscht werden sollen, oder aber, wenn das mit dem Speicher erfolgen soll. Eine Kommunikation mit Allzweckregistern oder Allzweckregisterkombinationen ist nicht möglich.

Wenn Ihnen dieses Verhalten ein wenig merkwürdig vorkommt, denken Sie bitte an folgendes: MMX ist keine neue Technik mit neuen Registern, einer neuen Unit zur Berechnung usw. – es ist schlicht und ergreifend ein etwas anderes, zusätzliches Verhalten, das man der Floating-Point-Unit mitgegeben hat. MMX ähnelt nicht nur aufgrund des Ortes der Datenmanipulation, den Coprozessorregistern, sondern eben auch in seinem Verhalten der FPU – auch wenn ausschließlich mit Integers gearbeitet wird und es den Stack mit seinen verschiedenen Möglichkeiten nicht gibt. Wenn Sie sich einmal nicht ganz sicher sein sollten, was bei MMX-Befehlen passiert, sollten Sie daran denken, daß die Nähe von MMX zu FPU um Dimensionen größer ist als zu der Integerarithmetik mit den Allzweckregistern. Die Ladebefehle sind so ein Beispiel.

Einen Unterschied der MMX-Ladebefehle zu denen der FPU gibt es dann doch. Das ist auch der Grund dafür, daß sie MOVx heißen und nicht PLDx. Denn während bei den FPU-Ladebefehlen immer nur als leer markierte Register geladen werden können – andernfalls wird eine Exception ausgelöst – können die MOVx-Befehle Registerinhalte überschreiben. Sie müssen das auch! Denn es gibt keinen Befehl analog zu FFREE, mit dem einzelne Register als leer markiert werden können. Genausowenig erfolgt beim Kopieren eines Registerinhaltes in einen Operanden ein »Poppen«, mit dem das Register automatisch »geleert« wird, da es ja keinen Stack gibt.

Doch das Problem des »Aufräumens« führt uns zu einem weiteren Befehl, der im Rahmen von MMX wichtig ist:

EMMS

EMMS ist so etwas wie der Aufräumbefehl, wenn man mit der Nutzung von MMX fertig ist. Denn jeder MMX-Befehl außer EMMS setzt ja das *Tag-Feld* aller Register auf »valid« und den TOS auf »0«. Man hat dann aber nur noch wenige Möglichkeiten, die FPU-Register wieder für das zu nutzen, wozu sie einmal gedacht waren: für FPU-Berechnungen. Man müßte schon entweder mit FINIT oder den Umgebungsladebefehlen FRSTOR und FLDENV für klare Verhältnisse sorgen (was sowieso nie falsch sein kann!).

Doch manchmal wäre dies »mit Kanonen nach Spatzen geschossen«. Denn sowohl die Initialisierung als auch das Laden einer Umgebung sind relativ zeitaufwendig – im Zeitalter knapper Ressourcen ein fast unverantwortliches Unterfangen, wenn es nicht absolut notwendig ist. Denn die MMX-Befehle ändern ja an der FPU-Umgebung nicht viel: Lediglich die Information über die Lage des TOS geht verloren, dagegen werden das Kontrollwort und das Statuswort nicht angetastet. Da ja die FPU-Register für die MMX-Befehle benötigt werden sollen, müssen sie sogar leer sein, weshalb es keinen Schaden bedeutet, den TOS auf 0 zu setzen.

Das aber heißt, daß man für die »alte« Vor-MMX-FPU-Umgebung ganz einfach sorgen könnte, indem man lediglich die Register leer fegt. Genau das tut EMMS durch das Setzen der *Tag-Felder* der Register auf »empty«. Unterbliebe dies, würde das nächste Laden eines FPU-Registers mit einem FPU-Befehl zu einem Stacküberlauf samt dazugehöriger Exception führen!

Fairerweise muß noch eine weitere Anpassung beschrieben werden, da sie von Intel schlecht dokumentiert wurde. FSAVE/FNSAVE haben unter MMX Konkurrenz bekommen:

FXSAVE, FXRSTOR

Wenn man sich alles überlegt, braucht man eigentlich über die bis jetzt genannten Befehle hinaus keine weiteren, um mit MMX arbeiten zu können: Daten können in die Register geladen und von dort abge-

holt werden, sie können arithmetisch oder logisch bearbeitet werden, »gepackt« oder »entpackt«, miteinander verglichen und auch bitweise manipuliert werden. Selbst der Status der MMX-Berechnungen kann gesichert oder restauriert werden. Denn nachdem MMX in den FPU-Registern abläuft, können ja auch FPU-Befehle verwendet werden, solange die nicht irgendwelche FPU-spezifischen Daten erwarten. Das machen aber weder FSTENV/FLDENV noch FSAVE/FRSTOR sowie deren N-Cousins (FNSAVE und FNSTENV).

Wieso besteht dann eine Notwendigkeit, daran etwas zu ändern? Die Antwort lautet: Geschwindigkeit! Als FSAVE & Co. implementiert wurden, kam es beim Sichern und Laden von Umgebungen und Registerinhalten weniger auf die Geschwindigkeit an: Fließkommaberechnungen sind vergleichsweise selten, laufen in der Regel innerhalb großer Blöcke ab, in denen, gemessen an der Gesamtausführungszeit, die Lade-/Speicherzeiten kaum ins Gewicht fallen, und lassen sich nicht zuletzt recht gut mit »Nicht-Fließkomma-Aktionen« parallelisieren. Warum sollte der Coprozessor nicht noch seine Register sichern, während der Prozessor bereits Bildschirmpositionen berechnet? (Das ist ja auch der Grund für die N-Zwillinge der Speicherbefehle; sie warten nicht ab, bis die Aktion erfolgt ist!)

Bei MMX ist das etwas anderes! MMX wird für Multimedia und Kommunikation eingesetzt. Daher kann man Aktionen nicht so ohne weiteres parallelisieren. Ferner ist Multimedia/Kommunikation ein heute recht häufig anzutreffendes Teilgebiet moderner Software, also alles andere als selten und ein »Spezialfall«. MMX-Module können klein sein, müssen aber häufig und ausgiebig genutzt werden.

Langer Rede kurzer Sinn: Das Laden und Speichern von FPU-Umgebungen ist kein »Sonderfall« mehr, sondern häufig praktizierte Notwendigkeit (siehe die Task-Switches bei Multitasking), vor allem, wenn mit MMX nun noch weitere Nutzungsmöglichkeiten offenstehen – und heftigst genutzt werden. FSAVE und FRSTOR mußten daher für MMX optimiert werden: FXSAVE und FXRSTOR sind die optimierten Zwillinge für FSAVE (genauer: FNSAVE) und FRSTOR. Wenn sie sich auch in Details unterscheiden (siehe Referenz), sind ihre Aufgaben die gleichen! Eine weitere Besprechung ist an dieser Stelle damit nicht erforderlich.

17.5 Beispiele für die Nutzung von MMX-Befehlen

Nun wissen wir, was MMX zu leisten in der Lage ist. Die Möglichkeiten sind schon recht bedeutend, wenn es in meinen Augen auch noch einige Ungereimtheiten gibt, die zu sehr auf die speziellen Aspekte von Multimedia ausgerichtet sind. Zwar heißt MMX nichts

anderes als Multi Media Extension; und damit wäre mein Einwurf gleich ad absurdum geführt. Aber dennoch glaube ich, daß man die MMX-Technologie auch bedeutend breiter verwenden könnte, wenn es die geeigneten Features gäbe, die MMX zu einem wirklich »runden« Paket machten. Einige Kritikpunkte habe ich bei der Besprechung der Befehle schon angebracht.

Aber ich möchte Ihnen ein paar Beispiele dafür geben, daß die Art, wie die MMX-Befehle arbeiten, sowie die Auswahl der implementierten Befehle nicht von ungefähr kommt, sondern durchaus ihre Berechtigung haben. Um mir nicht neue Beispiele ausdenken zu müssen, verwende ich lieber gleich die, die Intel selbst auch anbringt.

Stellen Sie sich in den Nachrichten den Wetterfrosch vor, der vor einer Wetterkarte das so schöne, mitteleuropäische Wetter präsentiert. Wir wissen, daß hier eine Menge von Informationen in der richtigen Weise bearbeitet werden muß, um das Gesehene auch zu ermöglichen. Dazu agiert der Wetterfrosch vor einem sog. Blue Screen, also einer wie auch immer einheitlich eingefärbten Wand. Diese wird – im Rechner – durch die Wetterkarte ersetzt. Und das geht so:

Zunächst muß im Videobild, das von dem Wetterfrosch aufgenommen wird, für jedes Pixel berechnet werden, ob es ein »Blue-Screen-Pixel« ist oder nicht. Das kann durch einen Vergleich mit der Farbe des Blue Screen einfach bewerkstelligt werden. Auf diese Weise erhalten wir eine Maske, die angibt, ob an dieser Pixelposition später ein Pixel der Wetterkarte stehen soll oder nicht. Diese Maske wird nun eingesetzt, um aus dem Videobild die Informationen herauszuholen, die nicht die Blue-Screen-Pixels darstellen. Dazu muß die Maske invertiert werden: Wir wollen alle Pixel, die nicht den Blue Screen darstellen. Anschließend kann mittels einer UND-Verknüpfung mit den ursprünglichen Videobild die wichtige Information extrahiert werden. Die ursprüngliche, nicht invertierte Maske kann aber auch benutzt werden, um in der Wetterkarte jenen Bereich auszublenden, an dem der Wetterfrosch stehen soll: Ganz einfach durch eine UND-Verknüpfung der Maske mit dem Bild der Wetterkarte. Der letzte Schritt ist die OR-Verknüpfung der beiden Teilbilder.

Macht man das mittels der herkömmlichen Befehle, so heißt das erstens, daß eine Programmverzweigung notwendig wird, da der CMP-Befehl die Flags verändert, nicht aber die Registerinhalte. Zweitens wird jedes einzelne Pixel einzeln auf diese Weise bearbeitet. Zusammen ist dies ein recht zeitaufwendiges Verfahren, was vor allem in der Programmverzweigung begründet ist.

Macht man das mittels MMX, so reicht die Folge PCMPEQW – MOVQ – PANDN – PAND – POR aus, um mit vier Pixels (im 256-Farben-Modus sogar 8!) gleichzeitig das Gewünschte zu erreichen – ohne zeitaufwendige

Programmverzweigung. Im Einzelnen: PCMPEQW, auf das Videobild des Wetterfrosches vor dem Blue Screen und dem Vergleichswert »blue screen color« angewendet, erzeugt eine Maske, an der überall 0 steht und an der nichts Wetterfroschhaftes zu finden ist. Diese Maske, invertiert und mit dem Videobild UND-verknüpft, was PANDN in einem Rutsch erledigt, liefert den »extrahierten« Wetterfrosch. Die mit MOVQ vorher kopierte Maske, UND-verknüpft mit der Wetterkarte, liefert die Schablone, in die mittels OR-Verknüpfung der Wetterfrosch eingepaßt wird. Das war es! (Vielleicht ist Ihnen auch jetzt klarer, warum es ausgerechnet den Befehl PANDN gibt und warum PCMPEQx so schöne Masken erzeugt.)

Ein weiteres Beispiel aus dem Videobereich: 24-Bit- (»true color«) Farbe und Überblenden. Stellen Sie sich vor, Sie möchten von einem True-Color-Bild auf ein anderes überblenden. Das bedeutet, Sie müssen pro Pixel 64 Bit verwalten und 32 Bit berechnen (je acht für die Farben Rot, Grün und Blau sowie für die Intensität; und das für das Ausgangs- und Endbild). Die Rechenvorschrift ist einfach: Jede Farbe jedes Pixels des einen Bildes wird mit der Intensität des Bildes 1 multipliziert und zu dem Produkt aus Intensität und Farbe jedes Pixels des anderen Bildes addiert. Beim Überblenden variiert nun die Intensität des Bildes 1 von 255 (volle Intensität) bis 0 (dunkel) in frei wählbaren Schritten. Die Intensität von Bild 2 ist klar: 255 – Intensität 1, denn die Gesamtintensität kann ja 255 nicht überschreiten! Macht man das nun konventionell, so müssen, eine 640x480-Auflösung vorausgesetzt, 640x480 = 307.200 Pixel berechnet werden. Das macht 3x307.200 Farben pro Bild – und das 255mal (das letzte Bild muß nicht berechnet werden: Es ist das Endbild). Das bedeutet: 470.016.000mal Laden und Multiplizieren sowie 235.008.000mal Addieren und Speichern. Das macht: 1.410.048.000 Operationen.

Vergleichen wir das mit der MMX-Technologie. Bei ihr werden vier Pixel auf einmal geladen, weshalb nur 117.504.000 Ladeoperationen notwendig werden. Für die Multiplikation wird nun eine Kombination aus UNPACK – PMUL eingesetzt, die die vier Pixelbytes in Worte »expandiert« und mit der geladenen Intensität multipliziert. Macht zweimal 117.504.000 Operationen. Über PADD – PACK werden die berechneten Werte addiert und wieder auf Bytegröße »gepackt«, was zusammen mit dem abschließenden Speichern dreimal 58.752.000 Operationen umfaßt. Das sind zusammen 528.768.000 Operationen, also 37,5% der konventionellen Lösung. Wahnsinn: fast zwei Drittel Ersparnis und das im Videobereich!

Auch an diesem Beispiel sehen Sie, daß die implementierten MMX-Befehle sehr wohl überlegt ausgewählt wurden. Es ging bei MMX nicht darum, Werkzeuge für die Bearbeitung von Werten auf allge-

meiner Basis zur Verfügung zu stellen, sondern ganz gezielt für den Einsatz bei speziellen Aufgabenstellungen, wie sie im Bereich Multimedia häufig auftreten. Auch das letzte Beispiel soll das zeigen: Im signalverarbeitenden Bereich von »natürlichen« Daten wie Sound, Video und Audio oder Mustererkennung spielt das Punkt-Produkt aus der Vektorrechnung eine entscheidende Rolle. Der Befehl PMADD wurde zur Optimierung der dazu notwendigen Basisberechnung implementiert. Mit seiner Hilfe lassen sich Matrixberechnungen um über zwei Drittel beschleunigen.

17.6 MMX und die Floating-Point Unit

Sie sehen – MMX ist nicht uninteressant und lädt zum Nutzen ein. Aber einer breiten Anwendung hat der »dumme« Anwender noch einen Riegel vorgeschoben: Es kauft sich eben nicht jeder sofort einen neuen Rechner, wenn es Prozessoren mit neuen Features gibt. Das heißt, daß der arme Programmierer für Leute mit und ohne MMX entwickeln muß – und unterscheiden können muß, ob nun ein MMX-Rechner vorliegt oder nicht. Wie also erkennt ein Programm, ob der Rechner die MMX-Technologie unterstützt? Über CPUID. Bit 23 des Feature-Flagregisters, das nach Aufruf von CPUID in Register EDX abgelegt wird, signalisiert im gesetzten Zustand die Verfügbarkeit der MMX-Technologie:

```
MOV       EAX, 000000001
CPUID
TEST      EDX, 000800000
JZ        MMX_Emulation
```

Schön, wenn die Prüfroutine ein gesetztes MMX-Bit vorfindet. Was aber, wenn nicht? Dann muß MMX emuliert werden. Bei diesem Gedanken fällt einem sofort die FPU-Emulation ein, die von modernen Prozessoren sogar hardwareseitig unterstützt werden kann. Gibt es auch die Möglichkeit, MMX analog zur FPU zu emulieren? Hat das EM-Flag in CR0, das ja bei der Emulation der FPU eine Rolle spielt, bei der Nutzung von MMX eine ähnliche Bedeutung? Leider nein: Die MMX-Emulation wird nicht ähnlich wie die Emulation der FPU unterstützt. Das bedeutet, daß bei einem gesetzten EM-Flag jedes Nutzen eines MMX-Befehls mit einer Invalid-Opcode-Exception (#DU) quittiert wird. Schade, eigentlich!

An dieser Stelle folgen noch ein paar Informationen und Hinweise, die Ihnen das Arbeiten mit MMX erleichtern sollen. Einige kennen Sie schon, sie werden hier dennoch nochmals aufgeführt.

Zunächst: Kapseln Sie MMX-Routinen und FPU-Routinen, wenn Sie beides benötigen. Verlassen Sie sich niemals darauf, daß andere Anwendungen/DLLs das auch tun. Gehen Sie also niemals davon aus, daß Sie einen »aufgeräumten« Satz FPU-Register vorfinden werden, wenn Ihre Anwendung startet. Es ist guter Stil und wird viele Probleme vermeiden helfen, wenn Sie sauber zwischen FPU und MMX unterscheiden und entsprechende Befehle nicht mischen – es sei denn, das ist beabsichtigt und Sie wissen, was Sie tun (wie immer)! Machen Sie es besser: Hinterlassen Sie, wenn Sie mit MMX-Berechnungen fertig sind, mittels EMMS eine aufgeräumte MMX-Umgebung. Analoges gilt natürlich auch für die FPU. Das hilft Ihnen, aber auch anderen! (Denn dann müssen nicht die anderen das nachholen, was Sie versäumt haben: für klare Verhältnisse sorgen.) Besonders wichtig ist dieser Hinweis, wenn Sie fremde DLLs oder andere Libraries nutzen. Achten Sie darauf, daß in solchen Fällen »saubere« Übergabebedingungen herrschen, indem Sie z.B. vor jedem Aufruf einer DLL-Routine, von der Sie nicht sicher sein können, daß sie keine FPU-Befehle enthält, eine MMX-Umgebung aufräumen! (Dies ist wirklich wichtig! Denn wenn z.B. eine DLL mit mathematischen Funktionen und höheren Berechnungen aufgerufen wird, so werden in der Regel mehr als eine Routine genutzt: Initialisierung der DLL, Aufruf verschiedener Funktionen etc. In der Regel wird aber eine einmal initialisierte Unit nicht vor jedem weiteren Funktionsaufruf nochmals prüfen, ob die FPU tatsächlich initialisiert ist oder etwa wieder initialisiert werden müßte – das, und das jeweilige FSAVE/FRSTOR nach und vor jeder Routine würde einen nicht tolerierbaren Overhead bedeuten! *Sie* wissen als einziger, wie Sie die Bibliothek nutzen – und ggf. mit MMX mischen! Also liegt die Verantwortung bei *Ihnen*, vor allem, weil »alte« Bibliotheken eventuell noch gar nichts von MMX »wissen« können.)

Je nachdem, ob das Betriebssystem kooperatives oder pre-emptives Multitasking ermöglicht, ist auch darauf zu achten, daß bei einem Taskwechsel ggf. entsprechende Schritte unternommen werden, die für eine geregelte Zusammenarbeit notwendig sind. Kooperative Multitasking-Betriebssysteme führen bei einem Taskwechsel *keine* Sicherung der Prozessor-, Coprozessor- und MMX-Umgebung durch! Damit ist es Aufgabe des Programmierers, diesen Zustand zu sichern, bevor er das Umschalten zum nächsten Task ermöglicht. Pre-emptive Multitasking-Betriebssysteme dagegen sind selbst dafür verantwortlich, daß die entsprechenden Sicherungen erfolgen und jeder Task den Zustand wieder vorfindet, bei dem er verlassen wurde. Der Programmierer muß sich in diesem Fall um nichts kümmern – im Gegenteil: Kümmerte er sich darum, würden die Dinge zweimal erfol-

gen, was zu deutlichen Performanceeinbußen führen würde. Das aber bedeutet wiederum, daß es Aufgabe des Programmierers ist, ggf. festzustellen, welcher Betriebssystemtyp vorliegt, und entsprechende Fallunterscheidungen zu treffen, die die Gegebenheiten berücksichtigen.

Denken Sie immer daran: Wenn ein MMX-Befehl einen Wert in ein »MMX-Register« schreibt, so werden die Bits 63 bis 0 des korrespondierenden FPU-Registers damit belegt. Alle weiteren Bits im 80-Bit-FPU-Register werden auf »1« gesetzt. (Das bedeutet: die FPU würde eine per MMX-Befehl geladene Zahl als negative Unendlichkeit bzw. negative NaN auffassen.) Alle MMX-Befehle außer EMMS setzen außerdem das TOS-Feld im Statusregister auf »0« und schreiben den Wert »00« in alle Tag-Felder, so daß alle Register als »gültig« markiert sind – unerheblich davon, welches und wie viele Register tatsächlich angesprochen wurden. (EMMS schreibt »11« in alle Tag-Felder und markiert somit alle Register als »leer«.) Weitere Veränderungen an FPU-Registerinhalten erfolgen nicht, insbesondere gibt es keine Veränderungen an CS:EIP oder DS:EDP oder im Opcode-Feld, im Statuswort oder in den Bits 0 bis 10 und 14 bis 15 des Kontrollworts.

Hochsprachenprogramme wie Pascal, Delphi oder C/C++ unterstützen bis heute noch nicht die MMX-Technologie. Das bedeutet, daß Sie Übergabemodalitäten zu regeln haben, wenn Sie Funktionen mit Hilfe der MMX-Technologie implementieren. Das wiederum heißt zweierlei: Sie müssen offenlegen, wie die Übergaben der Parameter und des Ergebnisses einer Funktion zu erfolgen haben, die MMX-Befehle enthält, wenn Ihre Funktion auch von anderen genutzt werden soll. So könnte man Parameter über die MMX-Register übergeben und das Ergebnis der Funktion ebenfalls. Man kann jedoch auch mit Zeigern und dem Stack arbeiten. Ich persönlich würde mit Zeigern auf selbstdefinierte 64-Bit-Strukturen (die Sie ja immer noch PackedBytes etc. nennen können) und Stack arbeiten, da auf diese Weise die Verantwortung für das Aufräumen der MMX-Umgebung bei der Routine liegt und dem dort Rechnung getragen werden kann, während im ersten Fall das rufende Modul die Verantwortung hat – was dann, im Falle fehlender EMMS-Befehle zu den oben geschilderten Inkompatibilitätsproblemen führen kann. Wie dem auch sei – es muß dokumentiert sein, wie es zu erfolgen hat. Noch ein Tip: Entscheiden Sie sich in Hinblick auf die Wiederverwendung, Portierung, Programmpflege und Lesbarkeit für eine Übergabeart, die Sie künftig nutzen wollen. Definieren Sie sie einmal *und halten Sie sich selbst daran*!

18 Windows 9x und Windows NT

Ich möchte mich an dieser Stelle nicht an der mir allzu akademisch erscheinenden Diskussion beteiligen, ob nun oder ob nicht, warum etwa oder warum nicht und inwieweit oder nicht Windows 9x ein Betriebssystem, ein Betriebssystemaufsatz (und somit immer noch DOS-abhängig) oder was auch immer ist – oder nicht! Ich möchte auch nicht an Disputen teilnehmen, welches der Systeme – Windows 95, Windows NT und OS/2 – nun das beste, sicherste, schnellste, ausgereifteste oder was auch immer ist. Faktum ist für mich, daß bereits sehr viele Computer unter Windows 95 laufen – und es werden immer mehr. Nicht zuletzt, da die Softwarehersteller beginnen, sich auf dieses Betriebssystem einzustellen, wie sich z.B. auf der *Software Development Conference* in San Francisco im März 1996 ausmachen ließ. Auch wenn andere Betriebssysteme überlegen sein sollten – man sollte niemals vergessen, weshalb und für wen ein Programm entwikkelt wird: zum Selbstzweck und der heiligen Theorie wegen oder für eine breite Allgemeinheit und die Praxis. Es mag ja sein, daß es »bessere« Betriebssysteme gibt. Aber wer entscheidet das? An dieser Stelle fällt mir immer der Autofahrer ein, auf dessen Grabstein die Inschrift steht »Er war schneller!«.

Zwar stehen mit OS/2 und Windows NT schon seit geraumer Zeit »echte« 32-Bit-Betriebssysteme zur Verfügung. Einen gewissen Durchbruch und somit nennenswerte Verbreitung hat, aus welchen Gründen auch immer, allerdings bisher nur Windows 95 geschafft. Dies scheint andere dazu zu beflügeln, für dieses Betriebssystem zu programmieren. Somit darf auch an dieser Stelle nicht versäumt werden, ein paar Informationen zu geben, die Ihnen weiterhelfen sollen. Es wird jedoch in keiner Weise der Versuch unternommen werden, auch nur ansatzweise in Windows 9x einzuführen – dies wäre im Rahmen dieses Buches ein hoffnungsloses Unterfangen. Zudem gibt es schon sehr viele und sehr gute Bücher zum Thema Windows 9x-Programmierung. Entwicklungssysteme wie Delphi und Visual Basic mit ihrer Art der Windows-Programmierung sind großartige Hilfen. Vielmehr soll versucht werden, auf Unterschiede zum guten alten DOS/Windows hinzuweisen, die beim Programmieren vor allem mit Assembler von Bedeutung sind.

Windows 9x ist ein 32-Bit-Betriebssystem, was heißt, daß die Adressierung von Speicherstellen in diesem System mit 32 Bit erfolgt, nicht etwa mit 16:16 Bit (eine andere Darstellung der Segmentierung des Speichers mit 20-Bit-Adressen) wie bisher. Falls Sie also mit diesem Betriebssystem arbeiten und nur für dieses Betriebssystem programmieren werden, so

vergessen Sie alles, was Sie im Kapitel *Der Adreßraum des 8086* ab Seite 15 über die Adressierung von Speicherstellen erfahren haben! Es gilt nun der 32-Bit-Adreßraum, der ab dem 80386 verfügbar ist, und somit das *Flat-Memory-Modell*. Hierzu jedoch noch einige Informationen.

Unter Windows 9x – und ich werde mich im folgenden auf dieses 32-Bit-Betriebssystem bei der Besprechung beschränken, Analoges gilt auch für Windows NT und OS/2 – verfügt jedes Programm über 4 GByte Adreßraum. Jedes! Da, wie wir schon gesehen haben, das Betriebssystem in jedem Fall die Verantwortung dafür hat, daß nur existente Adressen verwendet werden (können), ist es unerheblich, ob man den verfügbaren Speicher für die einzelnen Anwendungen vor dem Programmstart beschränkt oder erst während der Laufzeit. Ganz im Gegenteil: Geeignete Speicherzugriffsmechanismen vorausgesetzt, erleichtert man es dem Programmierer ungemein, wenn er bei der Entwicklung des Programms so tun kann, als ob der gesamte Adreßraum des 80386 dem eigenen Programm zur Verfügung steht, ohne wissen zu müssen, was um ihn herum passiert. Diese Verantwortung, nämlich den verfügbaren RAM auf einzelne Anwendungen zu verteilen und sicherzustellen, daß nur legale Adressen verwendet werden, kann man getrost dem Betriebssystem überlassen – dazu ist es da.

Windows 9x trägt diese Verantwortung sehr elegant! Ohne nun in Details gehen zu wollen, sei lediglich bemerkt, daß das Betriebssystem Mechanismen besitzt, die die jeweils angesprochene Speicherstelle dem Ort des Geschehens, dem RAM, zur Verfügung stellt. Das bedeutet z.B., daß Programm- oder Datenteile, die zur Zeit nicht benutzt werden, aus dem RAM ausgelagert werden können, bis sie wieder benutzt werden – z.B. auf die Festplatte. Solche Mechanismen kennen wir alle von der SWAP-Datei aus Windows 3.xx zur Genüge. Die Fähigkeiten zu solchen Auslagerungen wurden in Windows 9x jedoch erheblich verfeinert und ausgeweitet.

Hier kommen wieder die Segmentregister ins Spiel, die im *Flat-Memory-Modell* überflüssig erschienen. Denn der Adreßraum, den ein Programm besitzt, also die 4 GByte, sind virtuell. Mehr noch: sowohl das Code»segment« (also der Teil des Speichers, der den Code enthält) als auch das (oder die) Datensegment(e) können theoretisch jeweils 4 GByte groß werden. Das bedeutet, daß jedes Programm mit Adressen arbeitet, die gar nicht existieren! Das Betriebssystem ist dafür verantwortlich, daß dem Programm immer die aktuellen Daten (Code) im physikalischen Speicher zur Verfügung stehen (was man im Fachjargon als P*aging* bezeichnet). Dies erfolgt, indem das Betriebssystem die virtuelle Adresse, die das Programm nutzen möchte, in eine physikalische umrechnet, prüft, ob der entsprechende Datenbereich (Codebereich) im RAM vorliegt, und, wenn das nicht der Fall

ist, dafür sorgt, daß er – woher auch immer – in den physikalischen RAM geladen wird. Um dies alles bewerkstelligen zu können, braucht das System sogenannte *Deskriptoren*, in denen für Daten und Code die notwendigen Informationen stehen, wie z.B. Zugriffsberechtigungen, tatsächliche Größe, Anfangsadresse des betrachteten »Segments« etc. Die Adresse genau dieser Deskriptoren wird im *Flat-Model*, wie auch im Modell mit Speichersegmentierung, in den Segmentregistern gehalten.

An dieser Stelle gibt es bereits einige Einschränkungen. Zwar stehen den einzelnen Programmen unter Windows 9x und Windows NT tatsächlich 4 GByte virtuellen Speichers zur Verfügung – aber uneingeschränkt nutzen können sie ihn nicht unbedingt. Das liegt einfach daran, daß einige andere Programme, Programmteile und Bibliotheken auch in diesem Adreßraum liegen müssen (mehr als 4 GByte sind ja nicht adressierbar!), z.B. das Betriebssystem selbst und auch DOS-Teile. Hier unterscheiden sich Windows 9x und Windows NT etwas voneinander.

Unter Windows 9x liegt in den »untersten« 4 MByte des Adreßraums das »alte DOS« und ein 16-Bit-Windows. Genauer gesagt: Treiber und Module, die für die Kompatibilität mit Windows 3x und DOS (BIOS!) zuständig sind. (Auf die ersten 4 kByte davon hat Otto Normalprogrammierer keinen Zugriff, da das Segment, in dem diese 4 kByte liegen, das Segment »0« ist. Das heißt, der Selektor, der auf dieses Segment zeigt, hat den Wert »0«, ist also, wie man sagt, ein »Nullselektor«. Den darf Otto unter Windows nicht nutzen! Schade: Ade, du schöner BIOS-Datenbereich! Der restliche Bereich ist zwar für Schreib- und Lesezugriffe zugänglich, man sollte aber aus verschiedenen Gründen die Finger von ihm lassen.) Der restliche Adreßraum bis zur Grenze von 2 GByte steht den Anwendungen unter Windows 9x frei zur Verfügung. Er ist der eigentliche »virtuelle Adreßraum«, den eine Anwendung benutzen kann. (Das nenne ich Deflation: aus theoretischen 4 GByte werden im Handstreich noch nicht einmal »läppische« 2 GByte. Aber das ist doch auch schon etwas. Wer meckert, sollte erst einmal diesen Bereich vollständig nutzen können.) Danach kommt 1 GByte großer Bereich mit speicherresidenten Dateien, gemeinsam verwendeten DLLs und Libraries, 16-Bit-Anwendungen (»DOS-Boxen«) und gemeinsam genutztem Speicher. Auch dieser Bereich ist für Schreib- und Lesezugriffe freigegeben und kann von jedermann verwendet werden. Das letzte GByte bis zur Grenze des virtuellen 4-GByte-Raums ist für das Betriebssystem reserviert. Zwar ist es für Schreib- und Lesezugriffe freigegeben, man sollte aber aus offensichtlichen Gründen auf gar keinen Fall auf diesen Bereich zugreifen: Hier liegen die VxDs, die Speicherverwaltung und das Dateisystem – also das eigentliche Windows 9x.

Bei Windows NT sieht die Sache ein wenig anders aus: Die »unteren« 2 GByte des virtuellen Adreßraums sind für die Anwendungen, also für Schreib- und Lesezugriffe frei zugänglich. Die oberen enthalten das Betriebssystem und sind gesperrt. Beide Bereiche werden jeweils »nur« von einem 64-kByte-Bereich angeführt, auf den kein Zugriff besteht. Er dient dem Betriebssystem dazu, die Validität von Adressen zu überprüfen. So ist der Bereich von $00000000 bis $0000FFFF, also die »untersten« 64 kByte, das »Nullsegment«, das über einen Nullselektor angesprochen werden müßte, den es auch unter Windows NT nicht erlaubterweise geben darf (siehe dazu Einzelheiten im nächsten Kapitel). Analoges gilt für den Bereich $7FFF0000 bis $7FFFFFFF, also die »unteren« 64 kByte des »oberen« 2-GByte-Bereichs. (Übrigens: Warum unter Windows 9x »nur« die unteren 4 kByte, unter Windows NT dagegen satte 64 kByte zugriffsgeschützt sind, wird Ihnen nach der Lektüre des nächsten Kapitels klarer. Es hat etwas mit der Granularität der Segmente zu tun!)

Gehen wir nun ein wenig genauer auf die Mechanismen ein, die Windows 9x und NT vom altehrwürdigen Windows 3.x unterscheiden. Diese betreffen wohl am ausgeprägtesten die Adressierung mit den dazugehörigen Schutzmechanismen. Die folgenden Betrachtungen gelten grundsätzlich für die 32-Bit-Betriebssysteme aus dem Hause Microsoft, also für Windows 95, Windows 98 und Windows NT.

18.1 Die Berechnung von Adressen Teil 1 – Theorie: Von der effektiven zur physikalischen Adresse

18.1.1 Die logische Adresse

Betrachten wir zunächst das Problem aus dem Blickwinkel des Assemblerprogrammierers. Wie bekommt der ein Datum in ein Prozessorregister oder von dort in das RAM? Wie wir inzwischen hinlänglich wissen, erfolgt das über solche Befehle wie MOV AX, *DWord* oder ADD *LongInt*, EBX. Wie wir aus dem weiter vorn Gesagten auch wissen, spielen dabei die effektive Adresse EA und eines der Segmentregister eine Rolle. Das galt für den Real-Mode. Im Real-Mode wurde zunächst – rein formal – aus der EA eine sogenannte logische Adresse (LA) gebildet, indem man die gewünschte Speicheradresse als Offset zu einer Segmentgrenze auffaßte und schrieb:

logische Adresse (LA) = Segment : effektive Adresse (EA)

Somit ist die LA eine abstrakte Adresse, die der Prozessor noch vor dem eigentlichen Zugriff auf die Speicherstelle in eine »echte«, physikalisch ansprechbare Adresse umrechnen mußte. Wir wissen auch

von weiter oben, daß man diese physikalische Adresse (PA) erhält, indem man die im Segmentregister stehende Segmentgrenze mit 16 multipliziert und den Offset, die effektive Adresse, hinzuaddiert:

physikalische Adresse (PA) = Segment * 16 + EA

Was hat sich beim Übergang zum Protected-Mode geändert? Aus Sicht des Assemblerprogrammierer zunächst gar nichts! Denn auch im Protected-Mode werden MOV, ADD und alle anderen Befehle verwendet, die auf RAM-Adressen zugreifen. Sie werden in der gewohnten Art verwendet, das heißt unter Nutzung einer logischen Adresse. Auch die – rein formale – Notation dieser LA ist gleich geblieben:

logische Adresse (LA) = Segment : effektive Adresse (EA)

Das aber heißt: Auch im Protected-Mode arbeitet man mit Speichersegmentierung. Hat sich somit gegenüber dem Real-Mode nichts geändert? Doch! Die Speichersegmentierung im Real-Mode war ein Zwang, der aus der Verfügbarkeit von nur 16 Bit zur Adressierung und dem Vorhandensein von mindestens 1 MByte RAM resultierte. Echte Aufgaben hatten die Segmente nicht, weshalb sich auch niemand großartig um sie kümmerte. Die Konsequenz war, daß jeder auf Segmente zugreifen konnte, wie es ihm gerade paßte, und damit mehr oder weniger großen Schaden anrichtete. Im Protected-Mode dagegen hat man aus der Not eine Tugend gemacht. Man nutzt hier die Segmente als tatsächliche Einheiten, z.B. als Modul, in dem ausführbarer Code in Form von Programmen oder Libraries steht oder aber Daten. Jedes Segment hat hier bestimmte Eigenschaften, die ihm zugeteilt werden. Das sind z.B. eine explizit angegebene Segmentgröße, die Position des Segments im RAM (was im Real-Mode der in den Segmentregistern verzeichneten Segmentgrenze entspricht) und weitere Angaben, die vor allem von den Schutzmechanismen des Betriebssystems benutzt werden. Wir werden weiter unten noch darauf zurückkommen.

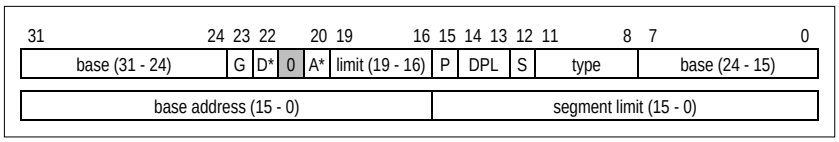
18.1.2 Segmente und ihre Deskriptoren

Segmente können im Protected-Mode an jeder beliebigen Stelle des theoretisch ansprechbaren Adreßraums liegen. Die Adressen der Segmentgrenzen müssen damit 32 Bit groß sein. Segmente sollen beliebig groß werden können. Es muß also die Information über die Größe des Segments verfügbar sein, und, wie bereits gesehen, müssen Segmente zumindest theoretisch so groß werden können, daß sie den gesamten theoretischen Adreßraum einnehmen können: 4 GByte. Somit muß auch das Segment 32 Bit groß sein. Doch damit nicht genug! Segmente sollen geschützt werden können. Das bedeutet, daß ihnen

auch Attribute oder Eigenschaften mitgegeben werden müssen: Handelt es sich um ein Datensegment, ein Codesegment oder gar einen Stack oder liegt ein besonderes »System«-Segment vor, in dem wichtige Systemdaten gehalten werden? Eine Fülle von Informationen also, die, verglichen mit der Situation im Real-Mode, sicherlich nicht mehr in einzelnen 16-Bit-Segmentregistern gehalten werden können.

All die eben angesprochenen Daten werden daher in sogenannten Deskriptoren gehalten. Deskriptoren sind nichts anderes als Datenstrukturen, in denen die Daten, die ein Segment beschreiben (daher der Name), enthalten sind. Nach dem eben Gesagten ist es nicht schwer, den Aufbau solcher Deskriptoren zu verstehen.

Segment Descriptor



D* = D/B; A* = AVL

Segmentdeskriptoren sind Strukturen von zwei Doppelworten (zweimal 32 Bits), in denen Angaben zur Basisadresse des Segments, die Segmentgröße und weitere Informationen verzeichnet sind. Die etwas fremd anmutende Aufteilung der Felder basiert auf der Kompatibilität zum 80286-Prozessor, da dieser nur 24-Bit-Adressen verwalten konnte. Die Bits 31 bis 16 des zweiten Doppelwort sind bei 80286-Prozessoren somit auf 0 zu setzen.

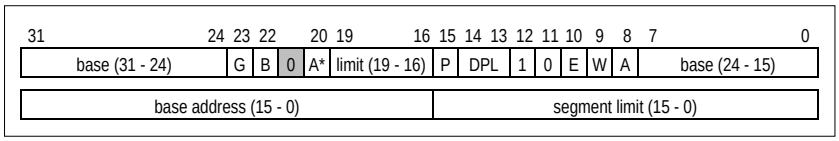
(Da die Adressen, vor allem aber auch die in den Limit-Feldern verzeichneten Segmentgrößen so irrsinnig zerstückelt sind, hatte Intel ein Einsehen und – spendierte einen Befehl, mit dem sich recht einfach die Segmentgröße in Bytes feststellen läßt: LSL. Siehe hierzu den Referenzteil des Buches!)

Die Basisadresse des Segments, also die Adresse, die im Real-Mode in die Segmentregister eingetragen wird, liegt im Protected-Mode in den Deskriptoren, zerstückelt in den Bits 31 bis 16 des ersten Doppelworts sowie in den Bits 31 bis 24 und 7 bis 0 des zweiten Doppelworts vor. Zusammengesetzt bilden diese Bits eine echte, lineare (32-Bit-)Adresse, die auf die Segmentbasis des betreffenden Segments zeigt. Die Information über die Größe des Segments befindet sich in den Bits 15 bis 0 des ersten Doppelworts sowie in den Bits 19 bis 16 des zweiten. Die 20 Bit der Segmentgröße können zunächst nicht die theoretisch mögliche und daher zu fordernde Segmentgröße von 4 GByte codieren. Denn mit 20 Bit lassen sich »nur« 2^{20} = 1.048.576 »Zellen« ansprechen. Daher gibt es noch das sogenannte *Granularity-Bit G* (Bit 23 des zweiten Doppelworts). Bei gelöschtem

Bit *G* sind die Zellen 1 Byte groß, bei gesetztem 2^{12} = 4.096 Byte. Auf diese Weise lassen sich Segmentgrößen von 1 MByte (*G* = 0; von 0 bis 1 MByte in 1-Byte-Schritten) bis 4 GByte (*G* = 1; von 4 kByte bis 4 GByte in 4-kByte-Schritten) verwalten.

Das Bit *S* (*System*) zeigt zusammen mit dem Feld *Type* an, ob es sich bei dem Segment um ein Systemsegment (*S* = 0; und wenn ja, um welches) oder ein Code- oder Datensegment (*S* = 1) handelt:

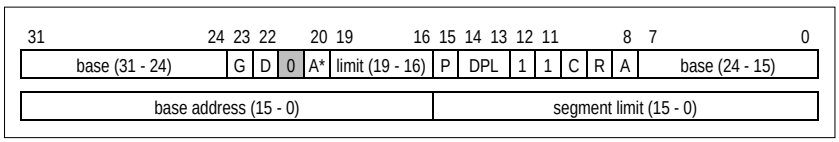
Data Segment Descriptor



G = granularity; B = big; A* = AVL; P = present; E = expansion direction; W = writable; A = accessed

Datensegmente (S = 1: Application und Bit 11 = 0: Data) besitzen als Type-Feld drei Flags. So bezeichnet das Flag *E* (*Expand Down*) Datensegmente, die »von oben nach unten« verwaltet werden, wie z.B. bei Stacks. Das Flag *W* (*Write Enable*) gibt an, ob auf das Datensegment nur lesend zugegriffen werden darf (read only: W = 0) oder auch schreibend (read/write: W = 1). Wurde auf dieses Segment bereits zugegriffen, so ist das *Flag A* (*Accessed*) gesetzt. Das Betriebssystem kann dieses Bit im Rahmen der virtuellen Speicherverwaltung und beim Debuggen wieder löschen. Bei jedem Zugriff auf das Segment setzt es der Prozessor automatisch.

Code Segment Descriptor

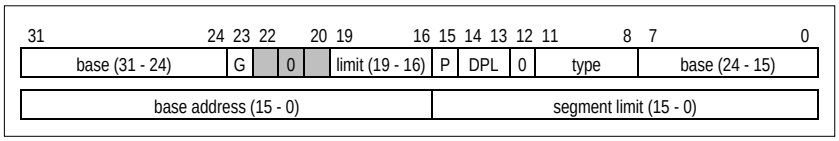


G = granularity; D = default; A* = AVL; P = present; C = conforming; r = readable; A = accessed

Auch Codesegmente (S = 1: Application und Bit 11 = 1: Code) besitzen als Type-Feld drei Flags. Das Conforming-Flag C definiert im gesetzten Zustand ein Conforming-Codesegment, also ein »anpassungsfähiges« Codesegment. Andernfalls ist es non-conforming, nicht anpassungsfähig. Diese Unterscheidung spielt bei der Schutzkonzeption des Protected-Mode eine Rolle. Was es damit auf sich hat, können Sie im Kapitel »Schutzmechanismen« nachlesen. Das Flag *R* (*Read Enable*) gibt an, ob das Codesegment nur zur Ausführung von Code gedacht ist (execute only: R = 0) oder aber auch Daten aufnehmen kann, auf die man dann lesend zugreifen können muß (execute/read: R = 1). Dies kann dann interessant sein, wenn z.B. in

einem ROM auch für Instruktionen relevante Daten stehen können. In jedem Fall kann auf Codesegmente nur lesend zugegriffen werden: Der Protected-Mode verhindert mit seinen Schutzkonzepten einen schreibenden Zugriff auf Codesegmente. Wie auch im Falle von Datensegmenten zeigt das *Flag* A (*accessed*) an, ob auf das Codesegment bereits zugegriffen wurde.

System Segment Descriptor

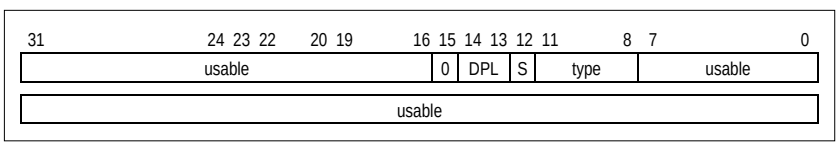


Systemsegmente (S = 0) sind alle Segmente, die keine Code- oder Datensegmente sind. In diesem Fall gibt das Feld *type* an, um was für ein Systemsegment es sich handelt. Hierzu gehören Segmente für Deskriptortabellen (LDT: type = 0010), *TaskState Segments* (TSS: type = x0x1), Gates (Call-Gates: type = x100; Task-Gate: type = 0101; Interrupt-Gates: type = x110; Trap-Gates: type = x111) oder reservierte Segmente (type = x000, 1010 und 1101).

Das Feld DPL (*Define Privileg Level*), die Bits 13 und 14 des zweiten Doppelworts, dient der Angabe, welchen *Privileg Level* das Segment besitzt, und ermöglicht so eine Überprüfung auf legitime Zugriffe auf das Segment im Protected-Mode.

Das Bit *P*, *Segment Present*, zeigt an, ob das Segment zur Zeit verfügbar ist (*P* = 1) oder nicht (z.B. weil das Segment als nachladbar gekennzeichnet ist und zwecks Bereitstellung von RAM aus diesem gelöscht worden war). Falls *P* = 0 ist, generiert der Prozessor eine »*Segment not present*«-*Exception*, falls ein Selektor in ein Segmentregister geladen wird, der auf den vorliegenden Deskriptor zeigt. Hierdurch kann im Rahmen des *Exception-Handlers* dafür Sorge getragen werden, das betreffende Segment verfügbar zu machen. Sollte P gelöscht sein, so sind die Bits 31 bis 0 des ersten Doppelworts sowie die Bits 31 bis 16 und 7 bis 0 des zweiten Doppelworts für die Software verfügbar. Betriebssysteme können hier z.B. verzeichnen, woher das »nicht anwesende« Segment geladen werden kann:

Non-present Segment



AVL (*available*). Dieses Bit kann von der Software (Betriebssystem) für eigene Zwecke benutzt werden.

Das Bit *D/B* hat, in Abhängigkeit davon, ob ein Codesegment (*S* = 1; Bit 11 = 1) oder ein Datensegment (*S* = 1; Bit 11 = 0) vorliegt, unterschiedliche Bedeutung. In Codesegmenten heißt das Bit *D, Default Operation Size,* und gibt an, ob die 32-Bit-Adressierung (*D* = 1) mit 32-Bit-Adressen und 32- oder 8-Bit-Operanden oder ob die 16-Bit-Adressierung (*D* = 0) mit 16-Bit-Adressen und 16- oder 8-Bit-Operanden verwendet wird. In Datensegmenten heißt das Bit *B, Default Stack Size,* und steuert Besonderheiten von sogenannten *Expand-Down-Segmenten*, also Segmenten, die »nach unten« wachsen, wie z.B. das Stacksegment. Ist *B* = 0, so wird für PUSH und POP das SP-Register verwendet. Die Obergrenze des Stacksegments ist dann $FFFF (64-kByte-Segmente). Bei *B* = 1 wird das ESP-Register verwendet, und die Obergrenze beträgt $FFFFFFFF (4-GByte-Segmente). Bei *Expand-Down-Segmenten* bezeichnet die Segmentgrenze (*limit*) die Untergrenze des Segments.

18.1.3 Die globale Deskriptortabelle und ihr Register

Soweit, so gut. Deskriptoren sind also nichts anderes als Datenstrukturen, die die dazugehörigen Segmente beschreiben! Auf Deskriptoren muß der Prozessor also zurückgreifen, wenn er mit einem bestimmten Segment arbeiten möchte, z.B. um den darin stehenden Code auszuführen oder die verzeichneten Daten zu manipulieren. Daher muß der Prozessor diese Deskriptoren, die ja ebenfalls im RAM liegen müssen, auch finden können. Denn in modernen Multitasking-Umgebungen können ja Dutzende von Segmenten und somit ihre Deskriptoren gleichzeitig im RAM gehalten werden.

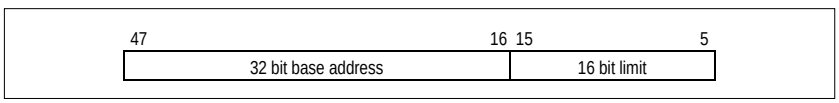
Wie immer, wenn sich solche oder ähnliche Problematiken ergeben, heißt die wohl beste Lösung: Liste. Der Prozessor verfügt also über eine Liste, in der er einen bestimmten Deskriptor finden kann. Diese Liste heißt im Falle der Deskriptoren *Deskriptor table*, d.h. Deskriptortabelle. Eine Deskriptortabelle ist also nichts anderes als die hintereinander aufgereihten Deskriptoren aller irgendwie interessierenden Segmente.

Die Deskriptortabelle und die Deskriptoren, aus denen sie besteht sind Daten! Damit liegt sie auch in einem (Daten-)Segment. Dieses Segment aber hat selbst eine Basisadresse und eine bestimmte Größe. Daher braucht der Prozessor wieder von irgendwo her die Information, wo im RAM die Deskriptortabelle liegt und wie groß sie ist. Wurde also das Problem nur verschoben und eine ziemlich komplizierte Art der Speicheradressierung gewählt, um dem Programmierer das

Leben so schwer wie nur irgend möglich zu machen? Nein! Denn erstens hat das Führen einer Deskriptortabelle zumindest das Problem gelöst, daß Dutzende von Deskriptoren wahllos verstreut im RAM liegen und wieder auffindbar sein müssen: Nun muß nur noch die Lage einer Datenstruktur verzeichnet werden. Zweitens gibt es zwei Möglichkeiten, wie dem Prozessor die Lage dieser Datenstruktur bekanntgegeben werden kann. So liegt im Real-Mode die Interrupttabelle an der Adresse $0000:0000. Der Prozessor »weiß« also automatisch, wo er die benötigten Daten finden kann. Die zweite Möglichkeit besteht darin, dem Prozessor ein spezielles Register zu spendieren, in dem Lage und Größe der Deskriptortabelle komfortabel gehalten werden können. Diesen Weg hat man im Protected-Mode gewählt, um möglichst flexibel zu sein.

Der Prozessor verwaltet eine für das Betriebssystem lebensnotwendige Liste. Die Bedeutung dieser Liste ist so groß, daß sie den schönen Namen Global Descriptor Table (GDT) erhalten hat. Adresse und Größe dieser GDT werden in einem speziellen Register, dem Global Descriptor Table Register (GDTR) gehalten. Über bestimmte Prozessorbefehle, LGDT (Load Global Descriptor Table) und SGDT (Store Global Descriptor Table), kann der Inhalt dieses Registers ausgelesen (SGDT) oder verändert (LGDT) werden.

Global Descriptor Table Register



Das *Global Descriptor Table Register* (GDTR) umfaßt also eine 32-Bit-Adresse sowie einen 16-Bit-Wert. Beide Angaben geben die Lage der *Global Descriptor Table im Speicher und ihre Größe* wieder. Wenn wir nun die Informationen, die im GDTR verzeichnet sind, mit dem vergleichen, was weiter oben über Segmente gesagt wurde, so fallen uns zwei Dinge sofort auf. Erstens: Es gibt keinerlei weitergehende Informationen als Adresse und Größe des Segments. Das Segment, das über das GDTR angesprochen werden kann, ist also offensichtlich nicht geschützt. Das ist auch richtig so, denn die GDT ist ja so fundamental, daß sie zu jedem Zeitpunkt und von jedermann benutzt werden können muß – sowohl lesend, um die notwendigen Informationen zu erhalten, als auch schreibend, um Segmente verwalten zu können. Zweitens: Es gibt kein Granularity-Bit. Mit den 16 Bits, die die Segmentgröße codieren, lassen sich also »nur« 2^{16} Bytes ansprechen, das Segment kann also »nur« 64 kByte groß werden. Da jeder Tabelleneintrag (Deskriptor) 8 Bytes umfaßt, kann eine GDT folglich 8.192

Einträge besitzen, was wohl zunächst einmal selbst für sehr anspruchsvolle Software samt Betriebssystem ausreichen sollte. (Wir werden aber weiter unten sehen, daß man noch erheblich mehr Informationen verwalten kann, wenn auch nicht mit der GDT!)

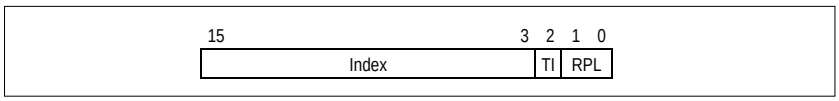
Zurück zur GDT. Der erste Eintrag der GDT ist ein sogenannter »Nulldeskriptor«. Dies bedeutet, daß alle Bits dieses Deskriptors gelöscht sind und von niemandem benutzt werden (dürfen!). Obwohl ohne weitere Funktion, führt das Laden eines Segmentregisters mit dem Selektor 0 (d.h. dem Zeiger auf den Nulldeskriptor) *nicht* zu einer Exception! Erst ein etwaiger Zugriff auf das (nicht vorhandene) Segment, auf das der erste Eintrag der GDT vermeintlich zeigt, führt zu einer #GP-Exception des Prozessors.

18.1.4 Segmentselektoren und ihre Register

Wie paßt das nun alles zusammen, wenn man an die Adressierung denkt, von der wir ja ausgegangen sind?

Der Prozessor »kennt« über sein GDTR die Adresse (und Größe) einer Liste, in der Einträge über die verfügbaren Segmente stehen: die GDT. Einzelheiten zu diesen Segmenten, wie Lage im RAM, Größe und Eigenschaften, sind in dieser Tabelle verzeichnet. Also braucht er nur noch zu wissen, an welcher Stelle der Liste nun der Deskriptor des Segments steht, auf das der Programmierer im Rahmen der Adressierung zugreifen will. Diese Information aber ist mit einem einfachen »Zeiger« in die Liste erhältlich. Solche Zeiger auf Einträge in Deskriptortabellen nennt man Selektoren:

Selektor



Selektoren bestehen aus 16 Bit Informationen und können somit in den uns schon aus dem Real-Mode bekannten Segmentregistern CS, DS, ES, FS, GS und SS stehen. Die Bits 15 bis 3 des Selektors codieren den Zeiger in die Deskriptortabelle. Bit 2 des Selektors wird als *Table Indicator* bezeichnet. Der Name ist Programm: Es gibt offensichtlich zwei unterschiedliche Deskriptortabellen, die über dieses Bit ausgewählt werden können. So ist es auch: Ist das Bit gesetzt, so verweist der Selektor auf eine sogenannte *Local Descriptor Table* (LDT), andernfalls auf die *Global Descriptor Table* (GDT). Was es mit der LDT auf sich hat, werden wir noch sehen! Die Bits 0 und 1 heißen *Requestor Privileg Level* (RPL). Sie spielen bei einem der Schutzkonzepte, die im Pro-

tected-Mode greifen, eine entscheidende Rolle; wir werden noch darauf zurückkommen. (An dieser Stelle nur so viel: Gesetzt den Fall, das zu betrachtende Segment dürfe nur ab einer bestimmten sogenannten Privilegstufe benutzt werden, so würde ein einfacher Vergleich der im Selektor verzeichneten RPL mit der augenblicklich herrschenden Privilegstufe (CPL) des rufenden Programms klären, ob das Programm dieses Segment nutzen darf oder nicht.)

Der Index besitzt also 13 Bytes und kann somit auf 8192 (2^{13}) Einträge in der entsprechenden Deskriptortabelle zeigen (was bedeutet, daß die Deskriptortabellen maximal 8192 Einträge besitzen können). Durch die Anordnung der Bits ist eine Berechnung des Offsets in die Tabelle einfach: der Inhalt des Selektors wird ausgelesen, die Bits 0 bis 2 werden gelöscht (de facto ist dies gleichzusetzen mit einer Multiplikation des Index mit 8, der Größe der Tabelleneinträge), und der resultierende Wert wird zur Basisadresse der gewählten Deskriptortabelle addiert.

Es wurde bislang behauptet, daß auch im Protected-Mode die Segmentregister, die die Segmentselektoren aufnehmen, nur 16 Bit groß sind. Stimmt das? Frei nach Radio Eriwan: im Prinzip ja. Richtig ist, daß lediglich 16 Bit dieser Register beschrieben oder ausgelesen werden können. Dennoch sind die Register bedeutend größer: Sie umfassen 80 Bit an Informationen. Die restlichen 64 Bit sind ein Cache, den der Prozessor – und nur er selbst! – automatisch verwaltet. Sie dienen zur Aufnahme der Daten des Deskriptors aus der Deskriptortabelle, auf die der Selektor zeigt. (Wie wir noch sehen werden, ist damit der Aufbau der Segmentregister identisch mit dem Aufbau des sogenannten Local Descriptor Table Registers, das die Daten der LDT enthält.) Sobald in das frei beschreibbare Feld des Segmentregisters ein neuer Selektor geschrieben wird (was immer dann erfolgt, wenn eines der Segmentregister DS:, ES:, FS:, GS: oder SS: mittels MOV oder entsprechenden Befehlen beschrieben wird) liest der Prozessor den an der bezeichneten Stelle in der Deskriptortabelle stehenden Deskriptor aus und legt die Information im nicht beschreibbaren Teil ab. Auf diese Weise wird erreicht, daß zur Berechnung der virtuellen Adresse nicht jedesmal der zeitaufwendige Weg über die Deskriptortabelle und die Deskriptoren selbst gegangen werden muß.

Segment Register

18.1.5 Lokale Deskriptortabellen und deren Register

8.192 Einträge in der globalen Deskriptortabelle scheinen eine Menge zu sein! Man stelle sich vor, daß damit 8.192 Segmente mit theoretisch jeweils 4 GByte Größe verwaltet werden können. Aber der Schein trügt wie so oft. Ein Prinzip des Protected-Mode ist ja die Schutzwürdigkeit von Segmenten. Das wiederum heißt, daß Module, Programme oder auch nur einige Programmteile sehr individuell behandelbar sein müssen. Berücksichtigt man dann noch, daß auch andere Mechanismen wie Multitasking und Zugriffe über Gates – was das ist, klären wir später – über die Speichersegmentierung erfolgen, so schmilzt der »riesige« Brocken von 8.192 Tabelleneinträgen schnell zusammen.

Dieser Umstand aber führte dazu, daß man einen weiteren Mechanismus brauchte, der die Begrenzungen der GDT überwand. Man fand ihn in der sogenannten Local Descriptor Table (LDT). Die LDT ist absolut identisch zur GDT aufgebaut! Das bedeutet: Auch sie ist ein Segment, auch sie enthält Deskriptoren, und auch sie wird über Selektoren angesprochen. Die Deskriptoren sind identisch mit denen, die in der GDT stehen. Kurzum, eine LDT ist nichts weiter als eine weitere GDT. Das geht so weit, daß es auch ein dem GDTR analoges Register gibt, in dem Adresse und Größe der LDT verzeichnet sind: das Local Descriptor Table Register (LDTR). Auch dieses Register kann mit den zu LGDT/SGDT analogen Befehlen manipuliert werden: LLDT und SLDT. Allerdings gibt es eine Ausnahme. Die LDT ist nicht *global*! Das wiederum heißt, daß es nicht nur eine LDT geben muß! Das aber wiederum bedeutet zweierlei. Erstens: Da es nur ein LDTR gibt, kann auch nur eine LDT zu einem beliebigen Zeitpunkt aktiv sein! Zweitens: Die Informationen zu der jeweils aktiven LDT selbst, also ihre Lage im RAM, ihre Größe und – die LDT ist ein Segment! – etwaige Zugriffsrechte müssen irgendwo beschrieben sein! Wo? Natürlich in einem Descriptor und der muß global verfügbar sein! Das bedeutet, daß LDTs wie andere Segmente auch in der GDT verzeichnet sein müssen.

Betrachten Sie daher die GDT und die LDT als »Einheit«: als eine Liste, in der Deskriptoren verwaltet werden, die die verfügbaren Segmente beschreiben. Diese Liste hat einen konstanten, immer gültigen und verfügbaren Teil, die GDT, und einen variablen Teil, der der jeweiligen Situation (Multitasking?) angepaßt ist bzw. werden kann – die LDT.

Noch ein Wort zum LDTR: Es ist absolut identisch zu den Segmentregistern aufgebaut! Das bedeutet, daß es einen »sichtbaren« 16-Bit-Teil gibt, der mittels LLDT/SLDT verändert werden kann und einen 16-Bit-Selektor aufnimmt, und einen »unsichtbaren« 64-Bit-Cache für den korrespondierenden Deskriptor aus der GDT. Der einzige Unter-

schied, der zu den Segmentregistern besteht, ist der, daß das LDTR Zugriffe auf die LDT ermöglicht, während über die Segmentregister auf Code- und Datensegmente sowie auf den Stack zugegriffen wird.

Local Descriptor Table Register

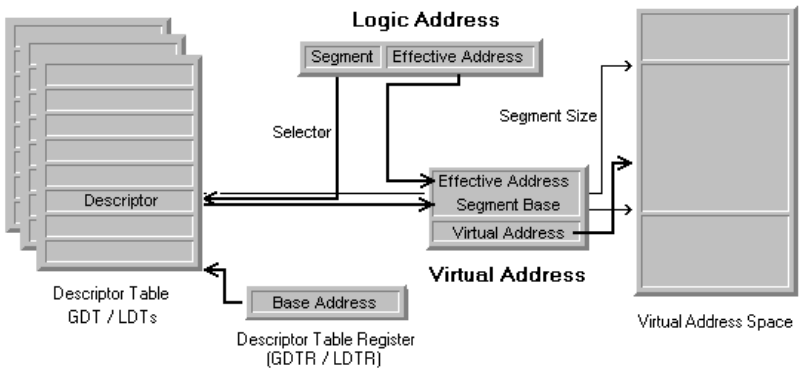
18.1.6 Die virtuelle Adresse

Die Berechnung einer Speicherstelle im Protected-Mode läßt sich also wie folgt zusammenfassen: Zunächst holt sich der Prozessor aus dem Descriptor Table Register die Adresse der benötigten Deskriptortabelle. Ob es sich dabei um die GDT oder die LDT handelt, wird ihm durch ein gelöschtes oder gesetztes TI-Flag im Selektor kundgetan. Den im Segmentregister stehenden Index des Selektors betrachtet er als Zeiger in die entsprechende Tabelle. An ihr steht die Adresse des Deskriptors, der das betreffende Segment näher beschreibt. Aus diesem Deskriptor liest er die 32-Bit-Segmentadresse des Segments aus und addiert die effektive Adresse (EA) hinzu, die als 32-Bit-Wert dem Assemblerbefehl mitgegeben wurde.

Zwei Dinge sind hierbei noch zu klären. Erstens: Ist das nun die physikalische Adresse (PA) wie im Falle des Real-Modes? Zweitens: Muß dieser Prozeß der Berechnung einer Adresse tatsächlich so kompliziert sein?

Zum ersten Problem. Tatsächlich ist das Ergebnis dieser Berechnung *nicht* die PA. Denn diese kann ja, weil 32 Bit groß, einen Wert bis zu 4 GByte beschreiben. Aber so viel physikalischen Speicher haben wohl die wenigsten Rechner. Daher bezeichnet man die berechnete Adresse auch als virtuelle Adresse (VA), also als Adresse in einem virtuellen, nicht tatsächlich vorhandenen Adreßraum, der maximal 4 GByte groß werden kann. Sie muß noch in einem weiteren Schritt in eine »echte« PA umgerechnet werden, die der Prozessor dann auch tatsächlich ansprechen kann. Zum zweiten Problem: Ja, der Prozeß der Adreßberechnung muß tatsächlich so kompliziert sein. Denn was bisher unterschlagen wurde, ist, daß der Prozessor während des ganzen Berechnens noch einige Aktivitäten durchführt, die sicherstellen, daß kein Unbefugter auf die betreffenden Segmente zugreifen kann. So wird z.B. anhand der im Segmentdeskriptor verzeichneten Größe des Segments geprüft, ob die mit der EA definierte Speicherstelle tatsächlich noch im Segment liegt oder nicht. Ist dies nicht der Fall, so meldet der

Prozessor eine Zugriffsverletzung, die wir alle als »General Protection Fault« und an der entsprechenden, harmlos aussehenden Meldung des Betriebssystems kennen und lieben gelernt haben. Aber auch wenn die EA tatsächlich in diesem Segment zu Hause wäre, könnte dennoch ein Zugriff verboten sein, z.B. weil dem entsprechenden Programm (z.B. dem von Ihnen geschriebenen) ein Zugriff auf das Segment (z.B. ein Betriebssystemmodul) generell untersagt wird.



(Noch ein kleiner Nachtrag! Tatsächlich ist die VA in einem bestimmten Fall doch mit der PA gleichzusetzen. Dann nämlich, wenn der Paging-Mechanismus ausgeschaltet wurde. Das kann man durch Löschen des PG-Flags in Kontrollregister CR4 erreichen. Dann interpretiert der Prozessor alle linearen Adressen, also die VA, die Adressen im GDTR oder IDTR usw. als physikalische Adressen. Aber das soll nur der Form halber gesagt sein – denn das wird Ihnen vermutlich bei modernen Betriebssystemen niemals begegnen. Zusätzlich ein Wort der Warnung: Sie sollten an diesen Flags niemals herumspielen! Oder Sie riskieren das totale Chaos.)

18.1.7 Die Page – Abbildung der kleinsten Einheit des segmentierten physikalischen Speichers

Kommen wir nun noch zur Berechnung der »echten«, durch den Prozessor tatsächlich nutzbaren Adresse. Segmente haben wir als Möglichkeit kennengelernt, Programme zu strukturieren. Sie sind also eine Hilfe, um Software in klar abgegrenzte Strukturen zu gliedern. Analoges kann man auch mit der Hardware machen. Hier heißen die »Segmente« Seiten, engl. *Pages*. So, wie dieses Buch physikalisch aus

mehreren Seiten besteht, besteht der physikalische Speicher auch aus »Seiten«, und so, wie jedes Buch eine unterschiedliche Anzahl von Seiten hat, kann auch jeder Rechner eine unterschiedliche Anzahl von Pages haben, was sich in der unterschiedlichen RAM-Ausstattung manifestiert.

Eine Page kann 4 kByte, 2 MByte oder 4 MByte groß sein. Gesteuert wird die zum Einsatz kommende Seitengröße durch drei *Flags*. Es sind dies das *Paging-Flag* PG, Bit 31 im *Control Register* CR0, die *Page Size Extension* PSE, also Bit 4 in CR4, und die *Physical Address Extension* PAE, Bit 5 in CR4. Außerdem spielt noch das *Page Size-Flag* PS im Page-Directory-Entry PDE eine Rolle.

Die Anzahl von Flags und die etwas verwirrenden Beziehungen zwischen ihnen stellt die folgende Tabelle dar:

PG	PAE	Effekt	PSE	Effekt	PS	Effekt	Resultat
0	?	kein Effekt	?	kein Effekt	?	kein Effekt	kein Paging
1	0	32-Bit-Adressen	0	Size Extension nicht möglich	?	kein Effekt	4–kByte-Pages
			1	size extension möglich	0	keine Extension	4-kByte-Pages
					1	Extension	4-MByte-Pages
	1	36-Bit-Adressen	?	kein Effekt	0	keine Extension	4-kByte-Pages
					1	Extension	2-MByte-Pages

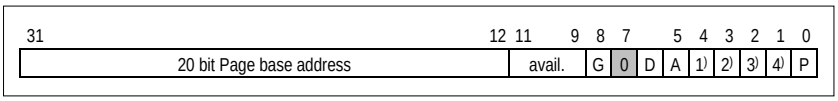
Der Normalfall dürfte wohl PG = 1, PAE = 0, PSE = 0 sein, womit der Inhalt von PS unerheblich ist und immer mit 4-kByte-Seiten gearbeitet wird. Bei den weiteren Betrachtungen wird das daher auch unterstellt.

Um nun eine Verbindung zwischen dem echten, physikalisch vorhandenen Speicher und dem virtuellen Adreßraum herzustellen, in dem sich die Programmierer mit ihren Segmenten bewegen, bleibt uns eine weitere, virtuelle Betrachtung nicht erspart: die Einteilung eines virtuellen Speichers diesmal nicht in Segmente, sondern in virtuelle Seiten. (Die dann in einem zweiten Schritt auf reale, physikalisch vorhandene Pages abgebildet werden können.) Gehen wir dazu zunächst einmal, rein virtuell, vom »Schlimmsten« aus, also daß unser Rechner tatsächlich 4 GByte physikalisch ansprechbaren Speicher hat. Nun teilen wir diesen Speicher auf. Da wir wissen, wie groß eine Page üblicherweise ist, können wir den Adreßraum in 1.048.576 Seiten à 4 kByte einteilen. Wir müssen also berücksichtigen, daß wir es im schlimmsten Fall tatsächlich mit dieser Anzahl physikalischer Seiten tun zu haben könnten.

Das wird aber in der Regel nicht der Fall sein. Sehr viel wahrscheinlicher ist, daß der Rechnerspeicher um einiges kleiner sein wird. 128 MByte ist schon einiges, aber eben sehr viel kleiner. Das aber bedeutet, daß wir eine Möglichkeit brauchen, die virtuellen Seiten auf physikalische abbilden zu können. Dazu verwenden wir wie für die Segmente einen Deskriptor für jede Seite, einen *Page-Deskriptor*. Dies ist eine Struktur, in der zu jeder virtuellen Seite Informationen abgelegt werden, so z.B., auf welche physikalische Seite sie abgebildet wurde, ob sie überhaupt verfügbar ist und ähnliches. Ganz nebenbei können wir dieser Struktur auch Informationen mitgeben, ob auf die Seite zugegriffen wurde, ob etwas hinsichtlich der Schutzkonzepte zu berücksichtigen ist usw. (Ich nehme es an dieser Stelle vorweg: den Begriff *Page*-Deskriptor gibt es nicht. Da wie die Segmentdeskriptoren auch diese »*Page Descriptors*« in Tabellen stehen, nannte man sie *Page Table Entry*. Anderer Name, gleiche Wirkung!)

Schauen wir uns also solch einen Page-Table-Entry einmal an:

Page-Table-Entry



1) PCD; 2) PWT; 3) U/S; 4) R/W

Denken Sie immer daran: der *Page-Table-Entry* beschreibt eine virtuelle Page und ihre Beziehung zu einer physikalischen! Die wichtigste Beziehung ist: gibt es eine »echte« Page, auf die die betrachtete virtuelle abgebildet werden kann? Und, wenn ja, wo steht sie? Hier: die Bits 31 bis 12 beherbergen eine 20-Bit-Adresse, die, mit dem Faktor 2^{12} = 4.096 skaliert (multipliziert), die 32-Bit-Basisadresse der zugehörigen physikalischen Page codiert. (Prozessorintern ist das recht elegant gelöst, indem die Adresse bei Bit 12 beginnt. Durch das Setzen der Bits 0 bis 11 des Tabelleneintrags auf »0« ist automatisch die korrekte 32-Bit-Basisadresse der Page verfügbar.) Pages können daher nur an 4-kByte-Grenzen im Speicher angesiedelt werden (was ja der Definition der Page entspricht!). Bit 8 ist das sogenannte Global Flag G, das eine »globale Page« anzeigt, wenn es gesetzt ist. Globale Pages werden, wenn das PGE-Flag im *Kontrollregister CR4* gesetzt ist, im Translation Lookaside Buffer nicht entfernt, wenn das Kontrollregister CR3 geladen oder ein Task-Switch durchgeführt wird. Dadurch wird ein häufiges Löschen/Nachladen im TLB bei oft benutzten *Pages* (die z.B. Systemteile beherbergen) verhindert.

(An dieser Stelle zwei Anmerkungen. Erstens: wie wir gesehen haben, können Pages auch andere Seitengrößen besitzen. Dann ändert sich natürlich auch entsprechend die Interpretation der 20-Bit-Adresse zur Umrechnung in die reale. Aber das soll hier nicht weiter interessieren. Wer

sich darüber informieren möchte, sei auf entsprechende weiterführende Literatur verwiesen. Zweitens: der Translation Lookaside Buffer wird auch nicht näher erklärt! Denken Sie einfach daran, daß der TLB einfach eine etwas aufgemotzte Prefetch Queue ist, die der Prozessor füllt, um unnötige Pausen beim Laden der Befehle aus dem Speicher zu verhindern. Ferner wird durch eine ausgetüftelte Verwaltung des TLB die Performance noch dadurch drastisch verbessert, daß mit dem TLB Voraussagen über eventuell notwendig werdende Sprünge gemacht werden können. Belassen wir es bei diesen Informationen – Sie werden vermutlich niemals in die Verlegenheit kommen, den TLB direkt manipulieren zu müssen. Falls doch, sei auch hier wieder auf weiterführende Literatur verwiesen.)

Bit 7 ist reserviert und auf Null gesetzt. Bit 6 heißt Dirty-Flag. Es zeigt im gesetzten Zustand an, daß in eine Seite geschrieben wurde. Bit 5, das Accessed-Bit A, zeigt an, ob auf die Page bereits zugegriffen wurde. Das Betriebssystem verwendet dieses Flag und das Dirty-Flag, um das Auslagern von Pages in und aus dem physikalischen Speicher zu verwalten.

Bit 4, das *Page-Level Cache Disable Flag*, wird benutzt, um das Cachen von Pages zu verhindern, bei denen dies keinen Nutzen hätte. Ist dieses Flag gesetzt, so wird die Page nicht gecacht. Bit 3, das *Page-Level Write-Through Flag*, kontrolliert, ob die individuellen Pages mittels »*Write-Through*« oder »*Write-Back*« über den Cache verwaltet werden. Bit 3 und 4 lassen also auf Page-Ebene eine Kontrolle der Arbeitsweise des Caches zu.

Bit 2, das User / supervisor-Flag, erlaubt, wenn gelöscht, die Nutzung von Supervisor-Privilegien im Rahmen der Schutzmechanismen auf Page-Ebene. Bit 1, das Read/Write-Flag, gibt an, ob die Pages nur gelesen (Bit 2 gelöscht) oder auch beschrieben werden dürfen.

Das Bit 0 (Present-*Flag*) signalisiert, ob die Page, auf die der Tabelleneintrag zeigt, zur Zeit im physikalischen Speicher vorhanden ist (Bit gesetzt) oder nicht. Falls das Bit gelöscht ist, so wird eine Page-Fault-Exception (#PF) generiert, falls der Prozessor versucht, auf die Page zuzugreifen. In diesem Fall hat das Betriebssystem durch Abfangen der #PF-Exception dafür zu sorgen, daß die entsprechende Page in den physikalischen Speicher geladen wird. Dies umfaßt

- das Laden der Seite von der Festplatte,
- das Eintragen der Basisadresse der Page in die Page-Table mit anschließendem Setzen des *Present-Flag*s,
- das »Invalidieren« der aktuellen Einträge im Translation Lookaside Buffer und
- die Rückkehr aus dem *Exception-Handler*, um das unterbrochene Programm erneut zu starten.

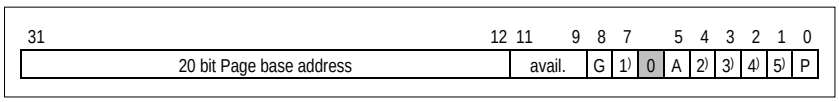
18.1.8 Die Page Table

Es werden nun wie im Falle der Segmentdeskriptoren solche *Page-Table-Entries* in einer sogenannten *Page-Table* zusammengefaßt. (Wen würde das bei diesem Namen auch wundern!) Machen wir uns ein paar Gedanken darüber, wie das erfolgen könnte. Wir wissen, daß wir 1.048.576 virtuelle Pages verwalten können müssen. Diese sollten strukturiert werden. Denn wir werden wohl nicht gleichzeitig auf alle Informationen aller Pages zurückgreifen müssen. Das bedeutet, daß wir die Verwaltung des Speichers genau so organisieren wollen wie den Speicher selbst: höchst dynamisch!

Unter Berücksichtigung der Page-Struktur fragen wir uns, wie viele Page-Table-Entries denn eine Page aufnehmen könnte. Ein Page-Table-Entry ist 4 Bytes groß, so daß eine 4-kByte-Seite 1.024 Einträge aufnehmen könnte. Dividieren wir nun die maximal mögliche Anzahl an Seiten durch eben diese »magische« Zahl, so erhalten wir wiederum 1.024.

Das heißt, daß wir 1.024 Page-Table-Entries zusammenfassen können und daraus analog zu dem Deskriptortabellen eine sogenannte Page-Table machen können. Diese Page-Table paßt haargenau auf eine einzelne Page. Das heißt aber noch mehr: daß wir theoretisch 1.024 solche Page-Tables haben können. Über die benötigen wir wieder einmal Informationen. Denn es wäre schon sehr einfallslos, allein 4 MByte »wertvollen« RAMs dadurch zu verschwenden, 1.024 Page-Tables zu verwalten, wo man vielleicht sogar nur einige wenige gleichzeitig braucht. (Immerhin kann man ja mit 32 Page-Tables $32 \cdot 1.024 = 32.768$ Pages à 4 kByte = 128 MByte RAM verwalten!) Ich mache es an dieser Stelle kurz: Es gibt analog der Page-Table-Entries *a*uch *Page-Directory-Entries*, die wie folgt aufgebaut sind:

Page-Directory-Entry



1) PS; 2) PCD; 3) PWT; 4) U/S; 5) R/W

Auch in diesem Fall beherbergen die Bits 31 bis 12 eine 20-Bit-Adresse, die, mit dem Faktor 2^{12} = 4.096 multipliziert, die 32-Bit-Basisadresse der zugehörigen Page-Table codiert. Page-Tables können daher, wie die Pages selbst auch, nur an 4-kByte-Grenzen im Speicher angesiedelt werden (und belegen somit eine eigene Page im Speicher). Bit 8 ist das sogenannte *Global* Flag *G*, das wie bei einem *Page-Entry* auch eine »globale Page« anzeigt, wenn es gesetzt ist. Globale Pages werden, wenn das *PGE-Flag* im Kontrollregister CR4 gesetzt ist, im

Translation Lookaside Buffer nicht entfernt, wenn das *Kontrollregister CR3* geladen oder ein *Task-Switch* durchgeführt wird. Dadurch wird ein häufiges Löschen/Nachladen im Translation Lookaside Buffer bei oft benutzten Pages (die z.B. Systemteile beherbergen) verhindert.

Bit 7 ist das *Page-Size-Flag* und signalisiert, wenn es auf 0 gesetzt ist, daß die Größe der verwalteten Page 4 kByte ist und die 20-Bit-Adresse auf eine Page-Table zeigt. Ist Bit 7 dagegen gesetzt, so sind die Pages 4 MByte groß, was sinnvoll sein kann, um größere, häufig benutzte Segmente wie z.B. Betriebssystemkerne vor einem Zerstückeln zu bewahren. In diesem Fall ist in den Bits 22 bis 31 eine 10-Bit-Adresse verzeichnet, die, mit dem Faktor 2^{22} = 4.194.304 multipliziert, die 32-Bit-Adresse der 4-MByte-Page codiert. Die Bits 12 bis 21 sind dann als reserviert zu betrachten.

Bit 6 ist reserviert und auf Null gesetzt. Bit 5, das *Accessed-Flag A*, zeigt an, ob auf die Page-Table bereits zugegriffen wurde. Das Betriebssystem verwendet dieses Flag, um das Auslagern von Pages in und aus dem physikalischen Speicher zu verwalten.

Bit 4, das *Page-Level Cache Disable Flag*, wird benutzt, um das Cachen von Page-Tables zu verhindern, bei denen dies keinen Nutzen hätte. Ist dieses Flag gesetzt, so wird die Tabelle nicht gecacht. Bit 3, das *Page-Level Write-Through Flag*, kontrolliert, ob die individuellen Tabellen mittels »*Write-Through*« oder »*Write-Back*« über den Cache verwaltet werden. Bit 3 und 4 lassen also auf Page-Ebene eine Kontrolle der Arbeitsweise des Caches zu.

Bit 2, das *User/Supervisor-Flag*, erlaubt, wenn gelöscht, die Nutzung von Supervisor-Privilegien im Rahmen der Schutzmechanismen auf Page-Ebene. Bit 1, das *Read/Write-Flag*, gibt an, ob die Tabelle nur gelesen (Bit gelöscht) oder auch beschrieben werden darf.

Bit 0 (*Present-Flag*) signalisiert, ob die Tabelle, auf die der Directory-Eintrag zeigt, zur Zeit im physikalischen Speicher vorhanden ist (Bit gesetzt) oder nicht. Falls das Bit gelöscht ist, so wird eine *Page-Fault-Exception (#PF)* generiert, falls der Prozessor versucht, auf die Tabelle zuzugreifen. In diesem Fall hat das Betriebssystem durch Abfangen der *#PF-Exception* dafür zu sorgen, daß die entsprechende Page in den physikalischen Speicher geladen wird. Dies umfaßt

- das Laden der Seite von der Festplatte,
- das Eintragen der Basisadresse der Page-Table in das Directory mit anschließendem Setzen des Present-Flags,
- das »Invalidieren« der aktuellen Einträge im Translation Lookaside Buffer und
- die Rückkehr aus dem *Exception-Handler*, um das unterbrochene Programm erneut zu starten.

18.1.9 Das Page Table Directory und sein Register

Das *Page [Table] Directory* ist nun endlich eine Struktur im Speicher, die die *Page-Directory-Entries* aufnimmt. Dieses »Inhaltsverzeichnis« für »Page-Tables« kann, wie gesagt, maximal 1.024 Einträge umfassen und ist, da jeder Eintrag 32 Bits breit ist, maximal 4 kByte groß.

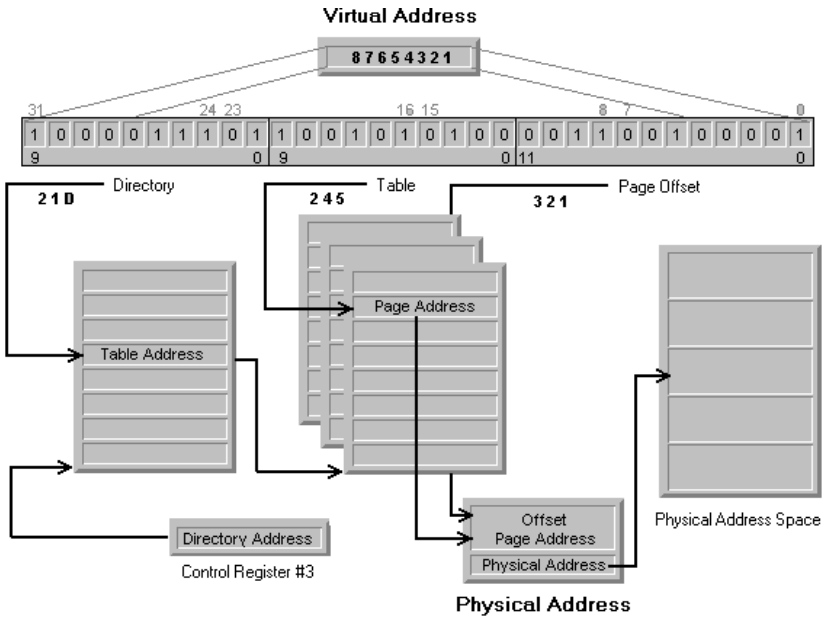
Nun haben wir das Pferd von hinten aufgezäumt. Wir haben ein Inhaltsverzeichnis, das uns Angaben über die Position und Eigenschaften von Tabellen gibt, die ihrerseits Angaben über Adressen und Eigenschaften von virtuellen Seiten beinhalten, die auf physikalische Seiten abgebildet werden können. Bloß: Wo, bitte, befindet sich dieses Inhaltsverzeichnis im Speicher?

Antwort: Analog zum GDTR gibt es ein Register, das die Adresse dieses Inhaltsverzeichnisses aufnimmt. Es ist das *Kontrollregister* 3 (CR3) des Prozessors, das auch auf den schönen Namen PDBR, *Page Directory Base Register*, hört.

18.1.10 Die physikalische Adresse

Kommen wir zu unserer ursprünglichen Fragestellung zurück. Nun ist es eigentlich nicht weiter schwer zu verstehen, was bei der Berechnung der PA aus der VA passiert. Analog zu den Gegebenheiten bei der logischen Adresse wird der 32-Bit-Wert der virtuellen Adresse VA als Offsetanteil und als Selektoranteil aufgefaßt. Der Selektor besteht aus zwei Teilen: Einer zeigt in das Page-Directory auf eine Page-Table, der andere auf eine bestimmte Seite in dieser Page-Table. Der dort stehende Eintrag sagt uns neben verschiedenen Dingen über Zugriffserlaubnis etc., welche physikalische Seite die virtuelle Adresse repräsentiert und ob die überhaupt verfügbar ist.

Zusammengefaßt wird also die PA berechnet, indem der Prozessor zunächst das PDBR ausliest, um festzustellen, an welcher Stelle im RAM die Adresse des Inhaltsverzeichnisses liegt. Mit dieser Adresse und einem Zeiger in das Inhaltsverzeichnis, der Teil der 32 Bit großen virtuellen Adresse ist, berechnet er die Adresse eines Tabelleneintrags, in dem die verfügbaren Seiten verzeichnet sind. Einen weiteren Teil der 32 Bit großen virtuellen Adresse, der ebenfalls als Zeiger aufgefaßt wird, verwendet er anschließend, um die physikalische Basisadresse der Seite zu erhalten, in der das gewünschte Segment liegt. Zusammen mit dem Rest der 32 Bit großen virtuellen Adresse kann dann die echte physikalische Adresse berechnet werden. Das folgende Schaubild macht das noch einmal deutlich:



Die Bits 31 bis 22 der 32 Bit breiten virtuellen Adresse werden als 10-Bit-Index in das Page-Table-Directory interpretiert. Der an der indizierten Stelle stehende Eintrag ist ein sogenannter Page-Directory-Entry, der neben einer 20-Bit-Adresse weitere Informationen enthält, die dem Paging-Mechanismus dienen (wir werden darauf zurückkommen). Diese 20-Bit-Adresse ist die Basisadresse der benötigten Page-Table. Der Index in diese Tabelle wird aus den Bits 31 bis 12 der VA gebildet. An entsprechender Stelle steht nun wiederum eine 32-Bit-Adresse, der Page-Table-Entry, der ähnlich wie der Page-Directory-Entry aufgebaut ist. Auch hier bilden die Bits 31 bis 12 eine Basisadresse, die in diesem Fall jedoch diejenige der physikalisch ansprechbaren Page darstellt, zu der die Bits 11 bis 0 der VA den Offset bilden.

18.1.11 Der Paging-Mechanismus – ein Beispiel

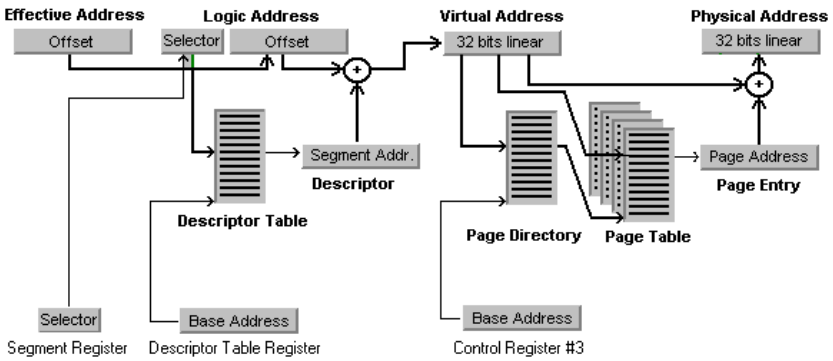
Auch an dieser Stelle stellt sich die Frage: Muß die Berechnung der PA aus der VA tatsächlich so kompliziert sein? Die Antwort lautet auch in diesem Falle: ja! Während man zu den guten, alten DOS-Zeiten noch das Problem hatte, mehr physikalischen RAM zur Verfügung zu haben, als man ohne Verrenkungen mit 16-Bit-Programmen

ansprechen konnte, hat man auch heute in einer Zeit billigen Speichers noch lange nicht die Möglichkeit, auch nur annähernd soviel Speicher nutzen zu können, wie man ansprechen könnte. Das Auslagern und Nachladen von Programmen und Programmteilen macht also eine komplexe Speicherverwaltung mit den geschilderten Mechanismen notwendig. Verschärft wird das Ganze noch durch das Multitasking, auf das weiter unten noch eingegangen werden wird.

Daß man für die softwareseitige Zerstückelung in Segmente und die hardwareseitige in Pages nicht analoge Mechanismen nutzen kann, ist offensichtlich. So kann die Hardware nur in bestimmten, diskreten Mengen mit RAM ausgestattet werden. So gibt es 4-MByte-, 8-MByte-, 16-MByte-, 32-MByte- und 64-MByte-Speichermodule. Diese lassen sich sehr einfach in 4-kByte-Pages einteilen: 1024, 2048, 4096, 8192 und 16.384 Seiten umfassen diese Module, also Potenzen von 2. Was liegt näher, als diesem Sachverhalt Rechnung zu tragen, indem eine Page-Table genau 1024 Pages aufnehmen kann? Der Speicherausbau mit den oben genannten Modulen findet damit seine Abbildung in 1, 2, 4, 8 und 16 Page-Tables! Und, welch Wunder, wenn man den theoretisch nutzbaren physikalischen Speicher von 4 GByte durch eben diese 1024 Pages à 4 kByte teilt, kommen genau 1024 Page-Tables heraus, die theoretisch verfügbar sein müssen und die auf sehr elegante Weise gerade die 1024 Page-Table-Entries des Pages-Table-Directo*rys* ausmachen. Der Paging-Mechanismus macht also durchaus Sinn.

Nicht anders verhält es sich mit der Speichersegmentierung. Es macht eben mehr Sinn, alle Daten, die sinnvollerweise zusammen verwaltet werden sollen (weil sie z.B. die Daten sind, die Ihr Programm benötigt), in einem eigenen Segment zu verwalten – unabhängig davon, wie groß die Datenstruktur in Bytes ausgedrückt ist. So können die Schutzkonzepte tatsächlich selektiv und spezifisch auf genau die Gesamtheit von Daten (auch Code sei hier darunter verstanden!) angewendet werden, die es betrifft! Das aber hat zur Folge, daß im Falle der Speichersegmentierung eine starre Einteilung wie beim RAM keinen Sinn macht. Eben dieser Gegensatz, nämlich variable Strukturen auf starre abzubilden, kombiniert mit dem Problem, daß eventuell weniger physikalischer Speicher verfügbar ist, als benötigt wird, macht dieses komplexe Verfahren der Umrechnung einer effektiven Adresse in eine physikalisch nutzbare notwendig.

Zusammengefaßt und verdeutlicht werden soll die »nervenaufreibende« Prozedur, um die Sie sich zum Glück nicht wirklich und höchstens in Ausnahmefällen kümmern müssen, an folgendem Schaubild:



Betrachten wir nun einmal ein Beispiel, wie der Paging-Mechanismus in der Praxis abläuft. Gegeben sei ein Programm, das stolze 3,22 MByte groß sein soll, genauer: 3.377.779 Bytes. Es greift auf insgesamt 15,17 MByte (16.003.498 Bytes) Daten zurück. Der Einfachheit halber besitzt es keinen Stack. Das Programm läßt sich also wie folgt beschreiben:

- Das Codesegment umfaßt die virtuelle Adresse 0 bis $00338A73.
- Das Datensegment beginnt an der virtuellen Adresse $00340000 und reicht bis $012831AA.

Wir wissen, daß der physikalische Speicher in Pages mit 4096 Bytes aufgeteilt ist. Der virtuelle Adreßraum muß also auf diese physikalisch vorhandenen Seiten in 4-kByte-Einheiten (4.096 Bytes) aufgeteilt werden. Das Codesegment reicht somit von Seite 0 bis Seite 824 (3.377.779 / 4.096 = 824,65 = aufgerundet 825; da Zählung bei 0 beginnt: 824!). Das Datensegment beginnt auf Seite 831 und reicht bis Seite 4.739.

Da die Seiten in Tabellen mit maximal 1.024 Einträgen verzeichnet werden, läßt sich folgende Beziehung erstellen:

Virtuelle Adresse	Virtuelle Page	Tabelle	Tabelleneintrag	Offset
$00000000	0	0	0	0
$00338A73	824 (=$338)	0	824 (=$338)	2.675 (=$A73)
$00340000	831 (=$33F)	0	831 (=$33F)	0
$012831AA	4.739 (=$1283)	4	643 (=$283)	426 (=$1AA)

Das bedeutet im Klartext: Die Einträge 0 bis 824 der Page-Table 0 sind für das Codesegment reserviert. Es umfaßt die Adressen 0 bis $00338000. Die Einträge 825 bis 831 sind leer bzw. werden zur Zeit nicht benutzt. Die Einträge 831 bis 1024 der Tabelle 0, die gesamten

Tabellen 1, 2 und 3 sowie die Einträge 0 bis 643 der Tabelle 4 repräsentieren das Datensegment bzw. dessen virtuellen Adreßraum. Übrigens: Es wurde hier von einer virtuellen Page gesprochen, da wir ja zunächst rein virtuell den virtuellen Adreßraum in 4.096-kByte-Blöcke, die virtuellen Seiten, aufgeteilt haben.

An dieser Stelle haben wir also den virtuellen Adreßraum in Pages strukturiert, deren Adressen in Tabellen eingetragen wurden. Nun müssen wir noch die Beziehung zum physikalischen Adreßraum herstellen. Dazu nehmen wir an, daß der Computer mit 8 MByte RAM ausgestattet sei – für heutige Verhältnisse wahrhaft deutlich zu wenig – aber eben auch zu wenig, um 3,2 MByte Programm und 15,2 MByte Daten gleichzeitig aufnehmen zu können. Das bedeutet, daß der Paging-Mechanismus ins Spiel kommen muß. Nehmen wir also weiter an, daß davon die »unteren« 1 MByte (z.B. wegen Real-Mode-Treiber, DOS-Bereich etc.) nicht benutzt werden können – um so schlimmer. Das heißt, daß das Programm zwar in den RAM paßt, nicht aber komplett mit Daten. Ein schön konstruiertes Beispiel!

Zunächst wird also das Programm in den Speicher ab 1 MByte geladen. Die virtuelle Adresse 0, mit der das Codesegment beginnt, liegt also an der physikalischen Adresse $00100000. Folglich wird in der Tabelle 0 als Page-Entry 0 die Adresse $00100000 (genauer: deren Bits 31 bis 12, da Pages nur an Page-Grenzen beginnen = 4.096 = 2^{12}) eingetragen. Page-Entry 1 dieser Tabelle bekommt dann die Adresse $00101000 (die Bits 31 bis 12), Page-Entry 2 $00102000 usw., bis mit dem Eintrag an Position 824 und der Adresse $00438000 das Ende des Codesegments erreicht ist. Die Einträge 825 bis 830 werden als leer markiert. In Eintrag 831 wird die physikalische Adresse der virtuellen Adresse 0 des Datensegments eingetragen. In unserem Fall folgt das Datensegment unmittelbar auf das Codesegment an Adresse $00439000, so daß der Eintrag 831 die Bits 31 bis 12 dieser Adresse zugewiesen bekommt.

Die 8 MByte RAM entsprechen 2.048 Pages. Daher werden die Einträge der Tabellen 0 und 1 vollständig mit den weiteren Adressen des Datensegments versehen, soweit es noch in den Speicher paßt. Da diese Seiten tatsächlich im Speicher vorhanden sind, wird Bit 0 des Eintrags, das »Present-Bit«, gesetzt. Die restlichen Einträge (die gesamten Tabellen 2 und 3 sowie die Einträge 0 bis 643 der Tabelle 4) erhalten ein gelöschtes »Present-Bit«. Nun wird das Programm gestartet.

Solange sowohl das Programm (das ja vollständig in den Speicher paßt) als auch dessen Daten nur in dem tatsächlich physikalisch vorhandenen Bereich bleiben, läuft alles reibungslos und ungehindert.

Nun aber versucht das Programm z.B., auf ein Datum an der virtuellen Adresse $01283100 zuzugreifen. Der Eintrag in Tabelle 4 an Position 643 hat ein gelöschtes »Present-Bit«. Ein Zugriff auf eine solchermaßen markierte Seite endet in einer *Page-Fault-Exception* (#PF).

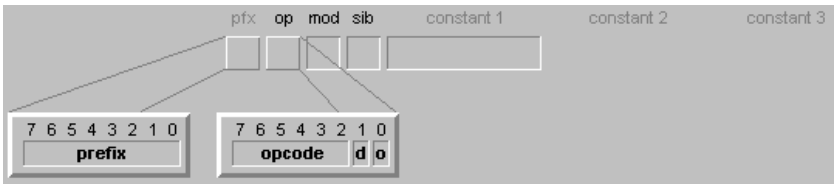
Nun ist das Betriebssystem gefragt! Es muß einen *Handler* für diese *Exception* haben. Dieser Handler ist nun dafür verantwortlich, daß die Seite mit dem gewünschten Datum geladen wird. Dazu muß eine bestehende Seite, auf die bisher sehr wenig zugegriffen wurde, entfernt werden. Nehmen wir an, es sei Seite 123. Das Betriebssystem markiert daher die Seite durch Löschen des »Present-Bits« in Eintrag 123 in Tabelle 0 als »nicht vorhanden«, lädt die Seite mit dem gewünschten Datum von der Platte und trägt in Tabelle 4 an Position 643 die Adresse $0007B000, also die physikalische Adresse ein, die die physikalische Seite 123 repräsentiert. Nach dem Markieren der Seite als »vorhanden« ist die Arbeit des Handlers erledigt, er kehrt in das unterbrochene Programm zurück. Dort wird der Befehl, der zur #PF führte, wiederholt. Nachdem die Seite jetzt tatsächlich vorhanden ist, greift das Programm auf die Adresse zu, die in Tabelle 4, Eintrag 643 steht: $0007B000 + Offset $100 und wird fündig.

Allerdings wird nun im Anschluß ausgerechnet an die Programmadresse $0007BC83 verzweigt. Ein Betrachten des Eintrags 123 in Tabelle 0 zeigt dem Prozessor, daß diese Seite nicht vorhanden ist – er löst wiederum eine #PF aus. Der Exception-Handler findet nun, daß die Seite 1.602 schon lange nicht mehr benutzt wurde. Also markiert er in Tabelle 1 den Eintrag 578 als »nicht vorhanden« und lädt von der Festplatte die (vorhin ausgelagerte) Seite mit der ehemaligen physikalischen Adresse $0007B000 an Adresse $00642000. In Tabelle 0 wird an Eintrag 123 genau diese Adresse (bzw. deren Bits 31 bis 12) eingetragen und als »vorhanden« markiert. Nach Rückkehr aus dem Exception-Handler kann dann wieder auf die gewünschte Adresse zugegriffen und die Programmausführung fortgesetzt werden.

Wo aber ist in unserem Beispiel das Page-Table-Directory? Kurz gesagt: wir haben es unterschlagen. Weil nämlich jede Page-Table die Adressen von 1024 Pages aufnehmen kann, wir aber in unserem Beispiel nur vier brauchten, haben wir ganz implizit immer mit dem Eintrag »0« des Page-Table Directory gearbeitet. Denn dieser ist ja für die Page-Tables 0 bis 1023 zuständig.

18.2 Die Berechnung von Adressen Teil 2 – Praxis: Adressen aus der Sicht des Assemblerprogrammierers

18.2.1 Präfix und OpCode



Sobald der Prozessor mit Daten umgehen muß, hat in der weitaus größten Zahl der Fälle ein Datentransfer zwischen den Registern des Prozessors und dem peripheren Speicher zu erfolgen. Hierzu besitzt der Befehlssatz der Intel-Prozessoren machtvolle Befehle, die entweder nur Daten zwischen Register und RAM verschieben oder eventuell zusätzliche arithmetische oder logische Operationen mit solchen Daten durchführen.

Alle Befehle dieser Art enthalten in ihrer Befehlsabfolge nicht nur den eigentlichen Opcode, also die Information, welche Operation zu erfolgen hat, sondern darüber hinaus noch zusätzliche Angaben, die

- im Opcode selbst codiert sein können, wie das Bit *Destination (d)*, das die Richtung des Transfers (zum Prozessor/in den RAM) festlegt, und das Bit *Operand Size (o)*, das Angaben zur Größe des zu behandelnden Datums macht. Ist $d = 0$, so ist das immer beteiligte Register die Quelle der Operation, wird also im Befehl als zweiter Operand angegeben. Bei $d = 1$ ist das Register das Ziel der Operation und somit erster Operand. Ist das *Operand-Size-Bit* $o = 0$, so werden 8-Bit-Daten, also Bytes, übertragen. Bei $o = 1$ handelt es sich um Standarddaten. Was nun ein Standarddatum ist, entscheidet die Umgebung, in der das Programm abläuft. Werden zur Adressierung 16-Bit-Adressen verwendet, so sind 16-Bit-Daten, also Worte, Standarddaten. Bei Nutzung von 32-Bit-Adressen sind es dagegen 32-Bit-Daten oder Doppelworte.
- als zusätzliche Bytes dem Opcode folgen; je nach Umgebung können hier bis zu zwei Bytes die Art und Weise codieren, wie die sogenannte effektive Adresse berechnet wird, mit der der Datentransfer zu erfolgen hat. Diese Bytes nennt man MOD- und SIB-Byte. Wir kommen darauf zurück. Schließlich können auch Konstanten folgen, wie z.B. Adressen von Speicherstellen oder »echte« Konstanten.

- als zusätzliche Bytes dem Opcode vorangestellt sein können. Diese Präfixe geben Auskunft über bestimmte Sondersituationen bei der Datenmanipulation. So kann (zumindest ab dem 80386) auch in 16-Bit-Umgebungen mit 32-Bit-Registern als Quelle oder Ziel einer Operation gearbeitet werden. Hier muß dann dem Prozessor kundgetan werden, daß er nicht Standarddaten übertragen soll, sondern eben 32-Bit-Daten. Dies erfolgt durch Voranstellen des Präfixes OPSIZE. Andererseits kann auch ein 32-Bit-Register eine Adresse enthalten, die in 16-Bit-Umgebungen zur indirekten Adressierung dient. Hierbei soll der Prozessor also Nicht-Standardadressen verwenden, was durch Voranstellen des Präfixes ADRSIZE signalisiert wird. Schließlich kann es sein, daß sich Adressen nicht auf Standardsegmente beziehen sollen. Um dies zu erzwingen, dienen die Segment-Override-Präfixe.

Solange der Datentransfer nicht zwischen zwei Registern intern im Prozessor erfolgt (was als Spezialfall der allgemeinen Adressierung angesehen wird), müssen also alle Befehle eine Quell- oder Zieladresse codiert bekommen, die Auskunft darüber gibt, welche Speicherstelle adressiert werden soll. Diese sogenannte logische Adresse besteht aus zwei Teilen, die gemäß der Speicherverwaltung in eine Segmentadresse und eine sogenannte effektive Adresse aufgeteilt wird, die einen Offset auf die Segmentadresse darstellt:

logische Adresse (LA) = Segment : effektive Adresse (EA)

wobei mit *Segment* die Adresse gemeint ist, an der das dazugehörige Segment beginnt. Details, wie diese Adressen dann vom Prozessor in physikalische Adressen umgerechnet werden, und wie, vor allem im Protected-Mode, der Speicherzugriff erfolgt, haben wir im letzten Abschnitt kennengelernt.

Der Begriff der effektiven Adresse wird nicht immer eindeutig benutzt. So wird manchmal unter der effektiven Adresse die Angabe einer vollständigen, eindeutigen Adresse verstanden, während andererseits häufig lediglich der sogenannte Offsetanteil dieser Adresse gemeint ist. In diesem Kontext wird die Nomenklatur des CPU-Herstellers Intel übernommen, der mit dem Befehl LEA (*Load Effective Address*) die Möglichkeit geschaffen hat, eine effektive Adresse in ein Register zu laden. Dieser Befehl verwendet lediglich den Offsetanteil zu einem Segment, das mittels eines Segmentregisters näher beschrieben wird. Daher wird hier als effektive Adresse ein 16- oder 32-Bit-Zeiger bezeichnet, der den Offset zu einem gegebenen Segment darstellt.

Die effektive Adresse kann auf zwei Arten bestimmt werden:

1. Durch direkte Angabe einer Konstanten, die den Offset selbst repräsentiert. So bewirkt z.B. der Befehl MOV EAX, DWordVar, daß der Assembler/Compiler die Stelle der Variablen *DWordVar* im Datensegment sucht, den Abstand zur Segmentbasis (den Offset) berechnet und den resultierenden Wert als Konstante in die Befehlsfolge für den MOV-Befehl einbaut. Als »direkt« bezeichnet man diesen Weg, da der Prozessor zur Laufzeit des Programms die Adresse nicht erst berechnen muß.

   effektive Adresse = Offset

2. Durch indirekte Adressierung, bei der der Prozessor die Adresse zur Laufzeit bestimmen muß. Hierfür benötigt er Informationen, die sich üblicherweise in Registern befinden und zur Berechnung eines Offsets herangezogen werden.

   effektive Adresse = Basis + (skalierter) Index + Displacement

Bei dieser Art der Bestimmung der effektiven Adresse werden eine sogenannte Basis, ein Index und ein *Displacement* verwendet. Die Basisadresse könnte z.B. die Adresse einer Tabelle mit Adressen sein, aus der der index-te Eintrag gelesen und als effektive Adresse verwendet wird. Beide Adressen befinden sich in Registern des Prozessors. Ergänzt werden kann diese Adreßberechnung durch die Addition einer Konstanten, des sogenannten *Displacements*.

In 16-Bit-Umgebungen, also Programmumgebungen, in denen 16-Bit-Adressen verwendet werden, gibt es nur zwei Register, die als Basis in Frage kommen: BP und BX. Auch als Indexregister gibt es nur zwei, nämlich SI und DI. Ferner ist eine (in der Formel angedeutete) Skalierung des Index nicht möglich, so daß nur eingeschränkte Möglichkeiten an Registerkombinationen bestehen. Welche Register nun, und ob überhaupt, beteiligt sind, regelt das sogenannte MOD-Byte, das dem Opcode folgt. In 32-Bit-Umgebungen sind prinzipiell alle Register als Basis verwendbar. Darüber hinaus sind auch alle Register als Index erlaubt, so daß die Möglichkeiten indirekter Adressierung in diesen Umgebungen deutlich erweitert sind. So ist klar, daß das MOD-Byte gegebenenfalls nicht zur Codierung der Verhältnisse ausreicht und ein weiteres Byte, das sogenannte SIB-Byte, erforderlich wird. Dadurch ändert sich auch die Definition des MOD-Bytes selbst.

Dem Opcode können bis zu vier sogenannte Präfix-Bytes vorangestellt sein: OPSIZE, ADRSIZE, ein Segment-Override-Präfix und LOCK.

Das Präfix OPSIZE, ein Byte mit dem Wert $66, wird immer dann verwendet, wenn die Größe der als Operanden verwendeten Daten nicht dem für den aktuellen Kontext gegebenen Standard entspricht. So sind in 16-Bit-Umgebungen (d.h. in Umgebungen, in denen 16-Bit-Adressen verwendet werden) 16-Bit-Daten (Worte) Standard. Befehle, die solche Daten manipulieren, benötigen das Präfix OPSIZE daher nicht. Soll jedoch in solchen Umgebungen z.B. auf die 32-Bit-Register zugegriffen werden, die ab dem 80386 zur Verfügung stehen, so muß dem Prozessor mitgeteilt werden, daß nicht die Standard-16-Bit-Register zu benutzen sind, sondern eben die neuen 32-Bit-Register. Diese Information wird durch das der eigentlichen Befehlsfolge vorangestellte Präfix OPSIZE übermittelt. In 32-Bit-Umgebungen sind 32-Bit-Daten und somit auch 32-Bit-Register Standard. Somit ist in diesen Umgebungen OPSIZE nur dann eingebunden, wenn 16-Bit-Register, also nicht die Standardregister, eingesetzt werden sollen. Es ist also de facto der entgegengesetzte Fall zum Sachverhalt in 16-Bit-Umgebungen.

Analoges gilt für das Präfix ADRSIZE, ein Byte mit dem Wert $67. Es wird immer dann verwendet, wenn die Größe der verwendeten Adressen (gemeint ist die effektive Adresse, also der Offsetanteil einer vollständigen Adresse) nicht dem für den aktuellen Kontext gegebenen Standard entspricht. In 16-Bit-Umgebungen werden, wie der Name schon suggeriert, 16-Bit-Adressen verwendet, die in 16-Bit-Registern verwaltet werden können. Jedoch kann es interessant sein, ab Prozessoren vom Typ 80386 von den erweiterten Möglichkeiten indirekter Adressierung Gebrauch zu machen. Dann jedoch werden aufgrund der 32 Bit breiten Register, die die Adressen halten, 32-Bit-Adressen verwendet, die in 16-Bit-Umgebungen nicht Standard sind. In diesem Fall stellt der Assembler der Befehlsfolge das Präfix ADRSIZE voran, das signalisiert, daß Nicht-Standardadressen benutzt werden. Analoges erfolgt in 32-Bit-Umgebungen bei der Verwendung von 16-Bit-Adressen, die hier nicht Standard sind. Es ist also de facto der entgegengesetzte Fall zum Sachverhalt in 16-Bit-Umgebungen.

ADRSIZE und OPSIZE werden niemals direkt vom Programmierer verwendet! Vielmehr sorgt der Assembler selbständig für das Einstreuen, wenn die Angaben im Assemblerquelltext dies erforderlich machen. So lautet die vom Assembler erzeugte Befehlsfolge für den Befehl MOV ESI, [EAX+EDI] in 32-Bit-Umgebungen 8B 34 38, in 16-Bit-Umgebungen dagegen 67 66 8B 34 38 (beachten Sie bitte, daß auch das Präfix OPSIZE verwendet wird, da Nicht-Standard-32-Bit-Daten übertragen werden. Falls nur 16-Bit-Daten manipuliert werden, wie in MOV SI, [EAX+EDI], so lautet die Folge 67 8B 34 38). Demgegenüber

ist MOV CX, [BX] in 16-Bit-Umgebungen mit 8B 4E 00 codiert, während in 32-Bit-Umgebungen 67 66 8B 4E 00 verwendet wird (auch hier entfällt OPSIZE, wenn die Standard-32-Bit-Daten übertragen werden sollen: MOV ECX, [BP] wird als 67 8B 4E 00 codiert). Analog lautet die vom Assembler erzeugte Befehlsfolge für den Befehl MOV CX, DX in 16-Bit-Umgebungen 8C BA, in 32-Bit-Umgebungen dagegen 66 8C BA. Demgegenüber ist MOV ECX, EDX in 32-Bit-Umgebungen mit 8C BA codiert, während in 16-Bit-Umgebungen 66 8C BA verwendet wird.

Das Präfix LOCK, $F0, setzt das LOCK-Signal des Prozessors während der Ausführung des sich anschließenden Befehls. Durch dieses Signal wird in Multiprozessorsystemen erreicht, daß der das LOCK-Signal setzende Prozessor den alleinigen Zugriff auf den Speicher besitzt, der von den Prozessoren geteilt wird. Dieses Präfix arbeitet nicht mit jedem Befehl zusammen.

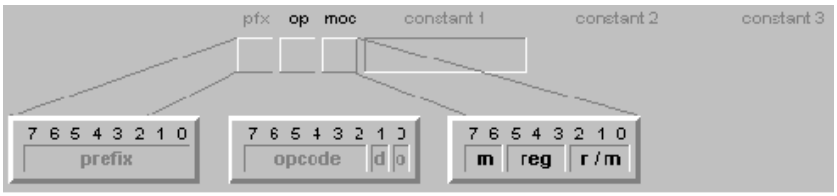
Jeder Befehl, der mit Zugriff auf den RAM zu tun hat, verwendet neben der effektiven Adresse auch ein Segmentregister, damit die zum eigentlichen Zugriff notwendige logische Adresse berechnet werden kann. Je nach Art der Bestimmung der effektiven Adresse kommen hierbei Standardsegmente zum Tragen.

So bezieht der Prozessor automatisch bei Angabe einer direkten Adresse (z.B. über Variablennamen) die effektive Adresse auf das Datensegment, dessen Selektor sich im DS-Register des Prozessors befindet. Stackmanipulationen beziehen sich auf das Stacksegment, dessen Selektor im SS-Register verzeichnet ist, Sprünge im Programmablauf beziehen sich auf das im CS-Register referenzierte Codesegment.

Bei den *Segment-Override-Präfixen* CS:, DS:, ES:, FS:, GS: und SS: handelt es sich nun um Informationsbytes, die dem Prozessor signalisieren, daß zur Berechnung der logischen Adresse bei einem Datenzugriff nicht das Standardsegmentregister verwendet werden soll. Soll z.B. anstelle der (normalen) indirekten Adressierung über das BP-Register (das standardmäßig mit dem in SS referenzierten Stacksegment verknüpft ist) der in BP stehende Offset in Relation zum Datensegment gesehen werden, so muß mittels des Segment-Override-Präfix DS: das im DS-Register referenzierte Segment zum Bezugssegment gemacht werden.

Die Präfixe werden im Assemblerbefehl unmittelbar vor die Adreßangabe gestellt, für die das gewünschte Segment verwendet werden soll. Beispiele: MOV AX, CS:[BX+DI], XCHG FS:[SI], DL, ADD DS:[BP], 01234h. In den assemblierten Befehlsfolgen steht das entsprechende Präfix dann unmittelbar vor dem Opcode der Folge.

18.2.2 Das MOD-Byte in 16-Bit-Umgebungen



Zur Berechnung der effektiven Adresse in 16-Bit-Umgebungen wird zusätzlich zum Opcode und eventuellen Präfixen ein weiteres Byte notwendig. Dieses Byte heißt MOD-Byte nach dem Namen der Bits 7 und 6 dieses *Bytes*, dem sog. *mod*-Feld (im folgenden mit *m* abgekürzt). Es steuert, welche Register/Register- oder Register/Memory-Kombinationen benutzt werden sollen und ob der Speicherzugriff durch direkte oder indirekte Adressierung erfolgen soll. In 32-Bit-Umgebungen kann aufgrund der wesentlich erweiterten Adressierungsmöglichkeiten über das MOD-Byte hinaus ein weiteres Byte, das SIB-Byte notwendig werden.

Das MOD-Byte läßt sich in drei Felder aufteilen: das *mod*-Feld (Bits 7 und 6), das *reg*-Feld (Bits 5 bis 3) und das *r/m*-Feld (Bits 2 bis 0).

Die Bedeutung der einzelnen Felder ist wie folgt:

- Das *m*-Feld gibt den Modus der Adressierung an:

00 Indirekte Adressierung mittels Indexregister (und evtl. Basisregister). Genauere Spezifizierung durch das *r/m*-Feld.

01 Wie *m* = 00; zusätzliche Addition eines 8-Bit-*Displacements*, das als Konstante dem Befehl folgt.

10 Wie *m* = 01; das *Displacement* wird als 16-Bit-Konstante angegeben.

11 Der zweite Operand ist ein Register.

Ausnahme: bei direkter Adressierung unter Angabe einer 16-Bit-Adresse ist *m* = 00 und *r/m* = 110.

- Das *reg*-Feld spezifiziert das (erste oder einzige) beteiligte Register des Prozessors.

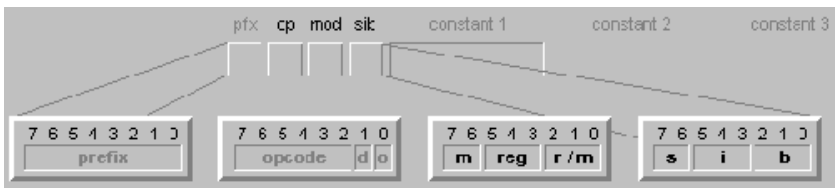
000 EAX/AX/AL	010 EDX/DX/DL	100 ESP/SP/AH	110 ESI/SI/DH
001 ECX/CX/CL	011 EBX/BX/BL	101 EBP/BP/CH	111 EDI/DI/BH

- Das *r/m*-Feld spezifiziert die verwendeten Index- und Basisregister:

000 [BX+SI]	010 [BP+SI]	100 [SI]	110 [BP]/direkt
001 [BX+DI]	011 [BP+DI]	101 [DI]	111 [BX]

Ausnahme: wenn *m* = 00 und *r/m* = 110 ist, kommt die direkte Adressierung mittels vollständiger 16-Bit-Adresse zum Einsatz. Soll lediglich [BP] als Basisregister (ohne Index) verwendet werden (*r/m* = 110), so kann aufgrund der Ausnahme für die direkte Adressierung *m* nicht den Wert 00 annehmen. In diesem Fall wird *m* = 01 gesetzt, was jedoch die Angabe eines *Displacements* nötig macht, das auf 0 gesetzt wird. Falls *m* = 11 ist, hat *r/m* die gleiche Bedeutung wie *reg*. Das Standardsegment bei der Verwendung von [BP], [BP+SI] und [BP+DI] ist SS, andernfalls DS.

18.2.3 Die MOD- und SIB-Bytes in 32-Bit-Umgebungen



Zur Berechnung der effektiven Adresse in 32-Bit-Umgebungen wird zusätzlich zum Opcode und eventuellen Präfixen, wie in 16-Bit-Umgebungen auch, ein weiteres Byte notwendig. Dieses Byte heißt MOD-Byte nach dem Namen der Bits 7 und 6 dieses *Bytes*, dem sog. *mod*-Feld (im folgenden mit *m* abgekürzt). Es steuert hier wie dort, welche Register/Register- oder Register/Memory-Kombinationen benutzt werden sollen und ob der Speicherzugriff durch direkte oder indirekte Adressierung erfolgen soll. Darüber hinaus kann in 32-Bit-Umgebungen ein weiteres Byte, das SIB-Byte notwendig werden, welches aufgrund der eingeschränkten Möglichkeiten der indirekten Adressierung in 16-Bit-Umgebungen nicht benötigt wird.

Das MOD-Byte läßt sich auch hier in drei Felder aufteilen, die eine zu 16-Bit-Umgebungen analoge Bedeutung haben: das *m*-Feld (Bits 7 und 6), das *reg*-Feld (Bits 5 bis 3) und das *r/m*-Feld (Bits 2 bis 0).

Das neue SIB-Byte hat einen ähnlichen Aufbau mit den Feldern *s* (Bit 7 und 6), *i* (Bits 5 bis 3) und *b* (Bits 2 bis 0). Die Berechnung der effektiven Adresse in 32-Bit-Umgebungen zeichnet sich gegenüber derjenigen in 16-Bit-Umgebungen dadurch aus, daß

1. jedes der acht Allzweckregister zur indirekten Adressierung herangezogen werden kann, also nicht nur BX, BP, SI und DI.
2. jedes dieser Register als Basis und/oder Index benutzt werden kann und
3. jedes Indexregister mit den Werten 1, 2, 4 und 8 skaliert werden kann!

- Das *m*-Feld gibt den Modus der Adressierung an:

00 Indirekte Adressierung mittels Indexregister (und evtl. Basisregister). Die genauere Spezifizierung erfolgt durch das *r/m*-Feld.

01 Wie *m* = 00; zusätzliche Addition eines 8-Bit-*Displacements*, das als Konstante dem Befehl folgt.

10 Wie *m* = 01; das *Displacement* wird als 32-Bit-Konstante angegeben.

11 Der zweite Operand ist ein Register.

Ausnahme: bei direkter Adressierung unter Angabe einer 32-Bit-Adresse ist *m* = 00 und *r/m* = 101.

- Das *reg*-Feld spezifiziert das (erste oder einzige) beteiligte Register des Prozessors.

000 EAX/AX/AL	*010 EDX/DX/DL*	*100 ESP/SP/AH*	*110 ESI/SI/DH*
001 ECX/CX/CL	*011 EBX/BX/BL*	*101 EBP/BP/CH*	*111 EDI/DI/BH*

- Das *r/m*-Feld spezifiziert die verwendeten Index- und Basisregister:

000 [EAX]	*010 [EDX]*	*100 [SIB-Byte!]*	*110 [ESI]*
001 [ECX]	*011 [EBX]*	*101 [EBP] / direkt*	*111 [EDI]*

Ausnahme: wenn *m* = 00 und *r/m* = 101 ist, kommt die direkte Adressierung mittels vollständiger 32-Bit-Adresse zum Einsatz. Falls *m* = 11 ist, hat *r/m* die gleiche Bedeutung wie *reg*.

- Das s-Feld gibt den Skalierungsfaktor an (Faktor = 2s):

00 Faktor 1	*01 Faktor 2*	*10 Faktor 4*	*11 Faktor 8*

- Das b-Feld spezifiziert das verwendete Basisregister.

000 [EAX]	*010 [EDX]*	*100 [ESP]*	*110 [ESI]*
001 [ECX]	*011 [EBX]*	*101 [EBP] / direkt*	*111 [EDI]*

Ausnahme: falls m = 00 ist, spezifiziert b = 101 die Verwendung einer 32-Bit-Konstanten.

- Das i-Feld spezifiziert das ggf. verwendete Indexregister:

000 [EAX]	*010 [EDX]*	*100 keines*	*110 [ESI]*
001 [ECX]	*011 [EBX]*	*101 [EBP]*	*111 [EDI]*

18.3 Die Berechnung von Adressen Teil 3 – praktische Beispiele

Im folgenden sind tabellarisch einige Beispiele zur Bestimmung des OpCodes bzw. der OpCodesequenzen angegeben, die im Rahmen der Adressierung auftreten können. Die Erläuterungen wurden so knapp wie möglich im Telegrammstil gehalten. Beachten Sie bitte die Unterschiede bei gleichen Befehlen in unterschiedlichen Umgebungen oder bei gleichen Instruktionen mit unterschiedlich breiten Parametern.

MOV CX, DX (16-Bit-Umgebung)
kein SIB-Byte (16-Bit-Umgebung, direkte Adressierung); Ziel ist Register (*d* = 1);
16-Bit-Daten (CX; *o* = 1; Standard: kein OPSIZE); Quelle ist Register (*mod* = 11);
Ziel = *reg* = CX = 001; Quelle = *r/m* = DX = 010; vollständig definiert!

prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	100010	1	1	11	001	010	-	-	-	-	-
		8B				CA						

MOV ECX, EDX (16-Bit-Umgebung)
kein SIB-Byte (16-Bit-Umgebung, direkte Adressierung); Ziel ist Register (*d* = 1);
32-Bit-Daten (ECX; *o* = 1; nicht Standard: OPSIZE); Quelle ist Register (*mod* = 11);
Ziel = *reg* = ECX = 001; Quelle = *r/m* = EDX = 010; vollständig definiert!

prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	OPSI	100010	1	1	11	001	010	-	-	-	-	-
	66	8B				CA						

MOV ECX, EDX (32-Bit-Umgebung)
kein SIB-Byte (direkte Adressierung!); Ziel ist Register (*d* = 1);
32-Bit-Daten (ECX; *o* = 1; Standard: kein OPSIZE); Quelle ist Register (*mod* = 11);
Ziel = *reg* = ECX = 001; Quelle = *r/m* = EDX = 010; vollständig definiert!

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	100010	1	1	11	001	010	-	-	-	-	-
		8B				CA						

ADD CH, ByteVar (16-Bit-Umgebung)
direkte Adressierung mit Variablenadresse (kein SIB-Byte; *mod* = 00; *r/m* = 100);
Ziel ist Register (d = 1); Datengröße durch CH mit 8 Bit festgelegt (o = 0; kein OPSIZE);
Register = *reg* = 101; die Variablenadresse benutzt das Standardsegment (kein *Segment-Override-Präfix*), zu dem der Assembler/Compiler die relative Lage der Variablen bestimmt und als Konstante (Offset) an die Befehlsfolge anhängt.

prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	000000	1	0	00	101	100	-	-	-	Offset	-
		02				2E					xx xx	

XOR WordVar, BP (32-Bit-Umgebung)
direkte Adressierung (kein SIB-Byte, *mod* = 00, *r/m* = 101); Register ist Quelle (d = 0) und 16 Bit breit (nicht Standard: OPSIZE, o = 1). Auch in diesem Fall wird der Offset vom Assembler/Compiler berechnet.

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	OPSI	001100	0	1	00	101	101	-	-	-	Offset	-
	66	31				2D					xxxxxxxx	

SUB ES:[BX]+08h, AL (16-Bit-Umgebung)
kein SIB-Byte (16-Bit-Basis); indirekte Adressierung mit einem 8-Bit-Displacement (»+08h«; *mod* = 01); 8-Bit-Register ist Quelle (d = 0; o = 0); Quelle = *reg* = AL = 000, Ziel = *r/m* = [BX] = 111; Segment-Override des Standardsegments zu [BX] (= DS): Präfix ES.

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	ES:	001010	0	0	01	000	111	-	-	-	disp	-
	26	28				47					08	

SUB [BP], AL (16-Bit-Umgebung)
kein SIB-Byte (16-Bit-Basis); indirekte Adressierung nur mit Basis, ohne Index (*mod* = 01, da *mod* = 00 mit *r/m* = 110 für [BP] für die direkte Adressierung reserviert ist);
8-Bit-Register ist Quelle (d = 0; o = 0); Quelle = *reg* = AL = 000, Ziel = *r/m* = [BP] = 110;
*Displacemen*t = 0, da nicht vorhanden, muß wegen *mod* = 01, aber angegeben werden.

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	001010	0	0	01	000	110	-	-	-	disp	-
		28				46					00	

SUB [BP], AL (32-Bit-Umgebung)
kein SIB-Byte (16-Bit-Basis); indirekte Adressierung nur mit Basis, ohne Index (*mod* = 01, da *mod* = 00 mit *r/m* = 110 für [BP] für die direkte Adressierung reserviert ist); 8-Bit-Register ist Quelle (d = 0; o = 0, kein OPSIZE!); Quelle = *reg* = AL = 000, Ziel = *r/m* = [BP] = 110; ADRSIZE notwendig, da in einem 16-Bit-Register eine Nicht-Standard-16-Bit-Adresse gehalten wird; *Displacement* = 0, da nicht vorhanden, muß wegen *mod* = 01 aber angegeben werden.

prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	ADR	001010	0	0	01	000	110	-	-	-	disp	-
	67	28				46					00	

SUB [EBP], AX (16-Bit-Umgebung)
indirekte Adressierung nur mit Basis, ohne Index (kein SIB-Byte; *mod* = 01, da *mod* = 00 mit *r/m* = 101 ([EBP]!) für die direkte Adressierung reserviert ist); 16-Bit-Register ist Quelle (d = 0; o = 1; Standard, daher kein OPSIZE); Quelle = *reg* = AX = 000, Ziel = *r/m* = [EBP] = 101; da ein 32-Bit-Register die Zieladresse enthält, wird ADRSIZE notwendig; *Displacement* = 0, da nicht vorhanden, muß wegen *mod* = 01 aber angegeben werden.

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	ADR	001010	0	1	01	000	101	-	-	-	disp	-
	67	29				45					00	

SUB [EBP], EAX (16-Bit-Umgebung)
indirekte Adressierung nur mit Basis, ohne Index (kein SIB-Byte; *mod* = 01, da *mod* = 00 mit *r/m* = 101 ([EBP]!) für die direkte Adressierung reserviert ist); 32-Bit-Register ist Quelle (d = 0; o = 1; kein Standard, daher OPSIZE); Quelle = *reg* = AX = 000, Ziel = *r/m* = [EBP] = 101; da ein 32-Bit-Register die Zieladresse enthält, wird ADRSIZE notwendig; *Displacement* = 0, da nicht vorhanden, muß wegen *mod* = 01 aber angegeben werden.

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
ADR	OPSI	001010	0	1	01	000	101	-	-	-	disp	-
67	66	29				45					00	

AND EDI, CS:[BP+DI+1234h] (16-Bit-Umgebung)
kein SIB-Byte (16-Bit-Basis); indirekte Adressierung über [BP+DI] (*r/m* = 011) mit 16-Bit-Displacement (*mod* = 10); 32-Bit-Register ist Ziel (d = 1; keine Standarddaten: OPSIZE und damit o = 1; *reg* = 111); der in [BP+DI] stehende Offset bezieht sich nicht auf das Standardsegment, sondern auf CS, somit Segment-Override CS:; 16-Bit-*Displacement* als 16-Bit-Konstante (nach Intel-Konvention mit LSB vor MSB!).

prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
CS	OPSI	001000	1	1	10	111	011	-	-	-	disp	-
2E	66	23				BB					34 12	

OR EDX, [EAX+12345678h] (32-Bit-Umgebung)
Indirekte Adressierung mittels Basis und *Displacement*, kein Index, keine Skalierung (kein SIB-Byte!); 32-Bit-Displacement (*mod* = 10). 32-Bit-Register ist Ziel (d = 1; o = 1; kein OPSIZE, *reg* = 010);Quelle = *r/m* = 000. Die Konstante wird nach Intel-Konvention LSB<Glossar> an höchster und MSB<Glossar> an niedrigster Adresse angegeben!

prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	000010	1	1	10	010	000	-	-	-	disp	
		0B				90					78563412	

SBB EBP, [EAX*4] (32-Bit-Umgebung)
Indirekte Adressierung mit skaliertem Index (SIB-Byte, *mod* = 00, *r/m* = 100), keine Basis (*b* = 101, *Displacement* muß explizit angegeben werden und ist hier somit 0, weil nicht vorhanden); Skalierungsfaktor 4 (*s* = 10); Index = *i* = 000. Register = *reg* = EBP = 101 ist Ziel (d = 1), es werden Standarddaten verwendet (o = 1, kein OPSIZE).

prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	000110	1	1	00	101	100	10	000	101	disp	-
		1B				2C			85		00000000	

MOV ESI, [EAX+EDI] (32-Bit-Umgebung)
Indirekte Adressierung mit Basis und Index ohne Displacement (SIB-Byte; *mod* = 00, *r/m* = 100); 32-Bit-Register ESI = *reg* = 110 ist Ziel (d = 1, Standarddaten: o = 1, kein OPSIZE); Basis ist EAX, Index EDI (*b* = 000, *i* = 111); Skalierungsfaktor nicht angegeben und damit standardmäßig 1 (*s* = 00).

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	100010	1	1	00	110	100	00	111	000	-	-
		8B				34			38			

MOV [EAX+ECX*8]+12345678h, EDX (32-Bit-Umgebung)
Indirekte Adressierung mit Basis, skaliertem Index und 32-Bit-*Displacement* (*mod* = 10, *r/m* = 100); 32-Bit-Register (o = 1, kein OPSIZE) EDX = *reg* = 010 ist Quelle (d = 0). Basis = EAX = *b* = 000, Index = ECX = *i* = 001; Skalierungsfaktor = 8 = *s* = 11.
Die Befehlsfolge ist somit harmloser, als es die Assemblerzeile vermuten läßt.

Prfx1	prfx2	code	d	o	mod	reg	r/m	s	i	b	const1	const2
-	-	100010	0	1	10	010	100	11	001	000	disp	-
		89				94			D8		78563412	

Tip

Die neue Art der indirekten Adressierung läßt sich recht angenehm für eine schnelle und unproblematische Art der Multiplikation mit 2, 3, 4, 5, 8 und 9 benutzen:

MOV EAX, [EAX*s] bzw. MOV EAX, [EAX+EAX*s], wobei s der Skalierungsfaktor 2, 4 oder 8 ist.

19 Multitasking

Eine der bedeutendsten Errungenschaften moderner Prozessoren ist ihre Fähigkeit, nicht nur ein bestimmtes Programm ablaufen zu lassen, wie man es vom Real-Mode her gewohnt war: sobald die Textverarbeitung aktiviert war, konnte man nicht schnell unterbrechen, um eine kurze Rechnung durchzuführen, deren Ergebnis man dann in den bearbeiteten Text integrieren konnte. Zunächst mußte das Dokument gesichert werden, die Textverarbeitung beendet, das Rechenprogramm gestartet, die Berechnung durchgeführt, das Ergebnis von Hand notiert und das Programm beendet werden. Nach dem Laden der Textverarbeitung mußte dann die ursprüngliche Datei wieder geladen werden, die Stelle gesucht, an der das Berechnungsergebnis eingefügt werden sollte, um dann von Hand eben dieses Ergebnis einzugeben.

Unter den heutigen Bedingungen ist das etwas anders. Heute ist es absolut üblich, zwischen Programmen beliebig hin und herzuschalten, Daten über eine sog. Zwischenablage von einem Programm in ein anderes zu übertragen und mehrere Dinge gleichzeitig zu tun. Das gipfelt darin, daß der Prozessor parallel eine Datei aus dem Internet herunterladen kann, während er in einem Fenster einen Krimi im Fernsehen zeigt, die Textdatei mit dem Buchmanuskript von 1200 Seiten formatiert und für den Druck vorbereitet, ja sogar druckt. Der Anwender durchforstet in der Zwischenzeit kurz eine Datenbank nach einem bestimmten Eintrag.

Für jeden dieser Vorgänge benötigt man unter DOS einen eigenen Rechner – jeder hat seine Aufgabe, seinen Task. Unter Windows und anderen Betriebssystemen ist das nicht nötig – sie sind, wie man sagt, multitaskingfähig.

Falls man mehrere Programme gleichzeitig ausführen möchte, muß, falls man sie nicht auf mehrere Prozessoren aufteilen kann, ein Mechanismus bemüht werden, den man üblicherweise als Multitasking bezeichnet. In ihm nimmt ein Task die Aufgaben wahr, die unter Be-

triebssystemen, die nicht zum Multitasking fähig sind, ein Programm innehatte. Multitasking ist somit eine (hardware-unterstützte!) Fähigkeit des Betriebssystems.

19.1.1 Der Task

Beim Multitasking wird der aktuelle Task (»Programm«) eine gewisse Zeit ausgeführt und dann »eingefroren«. An seiner Stelle wird nun ein anderer Task ausgeführt, der nach einer gewissen Zeit ebenfalls unterbrochen wird. Nun kann, wenn nur zwei Tasks laufen, der erste wieder ausgeführt werden. Andernfalls wird eben der nächste Task ausgeführt, so lange, bis der erste wieder an der Reihe ist. Die sehr kurze Zeiteinheit (»Zeitscheibe«), die jedem Task zur Bearbeitung zur Verfügung gestellt wird, und die häufige Frequenz, mit der alle Tasks gewechselt werden, erwecken den Eindruck »gleichzeitiger« Bearbeitung durch nur einen Prozessor.

Zur Laufzeit eines Tasks befindet sich dieser jedoch »allein« im Speicher, das heißt, »es gibt nur diesen einen!«. Das bedeutet, daß jeder Task die Mechanismen der Speicherverwaltung, inklusive des Paging, für sich nutzen kann. Um dies alles bewerkstelligen zu können, verwaltet das Betriebssystem daher Informationen für jeden einzelnen Task und wechselt rhythmisch die zur Speicherverwaltung notwendigen Angaben in den speziellen Registern aus. Man nennt dies Task-Switching.

Die wesentlichen Angaben zu einem Task, sein *Task-State* (TS), werden in eigenen Segmenten, den *Task-State-Segmenten* (TSS), gespeichert. Jeder Task hat ein eigenes TSS und, wie bei Segmenten üblich, einen *Task-State-Segmentdeskriptor*, der das Segment beschreibt und die Schutzmechanismen des Betriebssystems unterstützt. Dieser Deskriptor steht in der *globalen Deskriptortabelle* (GDT). (Steht er nur dort, oder kann er auch in der lokalen Deskriptortabelle stehen? Denken Sie einmal nach!) Auf den aktuell ablaufenden Task zeigt ein Selektor, der im *Task-Register* gespeichert wird und auf den entsprechenden TSS-Deskriptor in der GDT zeigt.

Stellt sich abschließend noch die Frage, was denn nun genau ein Task ist – nicht strukturell, sondern definitionsmäßig. Bislang haben wir ja als Task ein »Programm« bezeichnet und Multitasking mit dem quasigleichzeitigen Ausführen mehrerer Programme erklärt. Stimmt das? Ja – aber Tasks können auch andere Dinge sein. Ein Task kann z.B. der Handler eines Interrupts oder einer Exception sein. Denn analog zu

den *Interrupt-Handlern* unter DOS sind das in der Regel keine vollständigen Programme, auch wenn sie routinemäßig ausgeführt werden. Oder es können Betriebssystemroutinen sein, die parallel zu »Anwendungen«, wie »Programme« heute heißen, durchgeführt werden und werden müssen, so z.B. das ganz profane Weiterzählen des Zeitgebers von Windows. Auch in diesem Fall ist der Task kein vollständiges Programm, sondern eine Routine. Also: ein Task ist alles das, was sinnvollerweise eigenständig aufgerufen werden kann, um bestimmte Funktionen zu erledigen.

(Hier die Lösung der eben gestellten Frage, ob die TSS auch in der LDT stehen können. Nein, können sie nicht! Denn die LDT ist ja so etwas wie die Task-eigene, private GDT eines Tasks. Somit kann die jeweils aktuelle LDT anders aufgebaut sein als die vorhergehende. Da der Task-Switch aber ein vom Task unabhängiger Mechanismus ist und sein muß, darf das Betriebssystem nicht auf Task-abhängige Datenstrukturen zurückgreifen. Oder es müßte in jeder LDT die Information zu allen möglichen Tasks verzeichnet sein. Dann kann man sie gleich in die GDT legen! Aber es gibt noch einen anderen Grund: könnte man die TSS auch in den LDTs ablegen, bestünde die Gefahr des rekursiven Aufrufs von Tasks, was prinzipiell nicht möglich ist und sein darf.)

19.1.2 Der Task-State

Zurück zur Struktur! Wodurch wird nun ein Task eindeutig festgelegt? Was also beschreibt einen Task? Welches sind daher die Informationen, die in einem TSS abgelegt werden müssen? Die Antwort ist einfach: alles das, was der Prozessor bei der Bearbeitung eines Tasks benutzen kann und auch benutzt! Im einzelnen sind das:

- der aktuelle Adreßraum des Tasks. Dieser wird durch die Segmente definiert, die der Task nutzt. Deren Beschreiber, die Selektoren, sind in den Segmentregistern CS, DS, ES, FS, GS und SS verzeichnet. Da auf keine andere Weise Speicher angesprochen werden kann (vgl. das Kapitel »Die Berechnung von Adressen«), ist der verfügbare Adreßraum durch die Inhalte der Segmentregister eindeutig bestimmt.
- der Inhalt der Allzweck-, Index- und Basisregister sowie des EFlags-Registers; denn ein Task-Switch ist ja, von der Warte eines tasks aus betrachtet, so etwas wie »Schicksal«: Er kommt, man weiß es genau – bloß nicht genau, wann. Daher kann ein Task nicht prophylaktisch wichtige Registerinhalte sichern – das muß im Rahmen des Task-Switches erfolgen.

- der Inhalt des Instruction-Pointer-Registers EIP mit der Adresse des nächsten auszuführenden Befehls; auch hier gilt: niemand kann vorhersehen, an welcher Programmstelle ein Task unterbrochen wird.
- der Inhalt des PDBR. Das *Page-Directory-Base-Register* enthält die Adresse des *Page-Directory* und somit Informationen über die geladenen Pages während des Ablaufs des Tasks; wir haben ja gesehen, daß jedem Task der gesamte Adreßraum von (theoretisch) 4 GByte zur Verfügung steht, weshalb der Paging-Mechanismus bei jedem Task greift – und bei jedem Task anders.
- der Inhalt des Task-Registers; na ja – das dürfte klar sein! Denn das Task-Register enthält ja Angaben zum jeweils aktuellen Task. Aber darauf kommen wir noch zu sprechen.
- der Inhalt des Local-Descriptor-Table-Registers mit Adresse der lokalen Deskriptortabelle, die zu dem aktuellen Task gehört; auch das dürfte inzwischen klar sein.
- die Basisadresse der I/O Map;
- die Zeiger auf die Stacks von Privileg Level 0 bis 2; diese Stacks sind für die Zusammenarbeit mit dem Betriebssystem unerläßlich.
- die Verknüpfung zum vorangegangenen Task (»nested tasks«); soweit vorhanden!

Wie Sie sehen, sind das eine Menge, aber alles sinnvolle Informationen, die in der Lage sind, einen Task vollständig zu beschreiben.

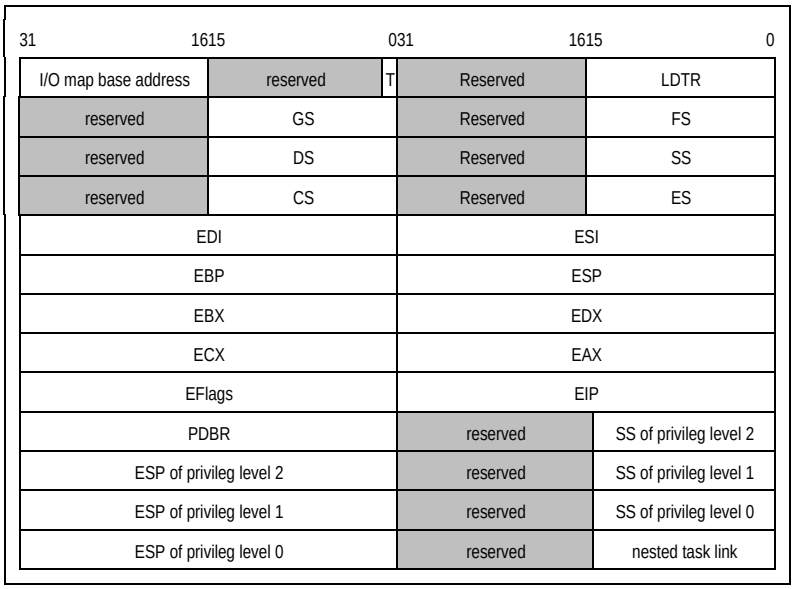
Diese Informationen bilden also den Task-State (TS), den Zustand eines Tasks zu einem bestimmten Zeitpunkt. Letzeres ist wichtig! Ein TS ist nichts Konstantes, wie es – in gewissem Rahmen – eine GDT oder die LDTs sind. Ein TS ändert sich, während der Code eines Tasks abgearbeitet wird - und wenn die Änderungen lediglich in anderen Inhalten des Program Counters CS:EIP begründet sind.

Das aber bedeutet, das der begriff Task-State nur im Rahmen eines Task-Switches Bedeutung hat. Dann nämlich, wenn der aktuelle Zustand eines Tasks »eingefroren« und gesichert bzw. geladen und »aufgetaut« wird – durch den Task-Switch.

19.1.3 Das Task-State-Segment

All die Informationen zu einem Task, also den Task-State, kann man als Datenstruktur darstellen, die selbstverständlich wieder in ein Segment gesteckt wird: das Task-State-Segment. Hier ist sein Aufbau:

32-Bit-Task-State-Segment



31 – 16	15 – 0		
I/O map base address	reserved	T	+96
reserved	GS	FS	+88
reserved	DS	SS	+80
reserved	CS	ES	+72
EDI		ESI	+64
EBP		ESP	+56
EBX		EDX	+48
ECX		EAX	+40
EFlags		EIP	+32
PDBR		reserved	+24
ESP of privileg level 2		reserved	+16
ESP of privileg level 1		reserved	+8
ESP of privileg level 0		reserved	+0

31 – 16	15 – 0	
LDT selector	DS	+40
SS	CS	+36
ES	DI	+32
SI	BP	+28
SP	BX	+24
DX	CX	+20
AX	Flags	+16
IP	ss level 0	+12
sp level 2	ss level 1	+8
sp level 1	ss level 0	+4
sp level 0	nested task link	+0

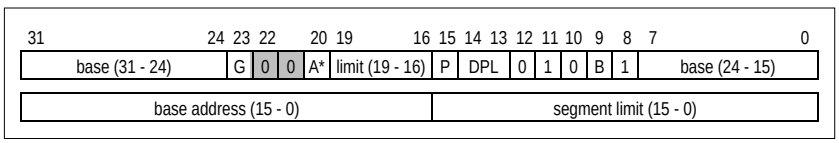
(Zum Vergleich wurden die beiden Tabellen in gleicher Weise aufgebaut.) Sie werden sowohl im Falle des 32-Bit-TSS wie auch beim 16-Bit-TSS keine Überraschungen erleben, was den Aufbau des TSS betrifft: Das TSS ist le-

diglich eine Struktur, in der an bestimmten, wohldefinierten Stellen all die Informationen abgelegt werden, die wir eben betrachtet haben. Lediglich auf drei Punkte soll hingewiesen werden: Adressen in den Privilegstufen 0 bis 2, gespeichert an den Offsets 4 bis 28, sind logische Adressen des jeweiligen Stacks. Zweitens: Durch ein gesetztes T-Flag (Debug-Trap-Flag) wird eine *Debug-Exception* generiert, wenn ein Task-Switch erfolgt. Drittens: der Nested Task Link ist eine Rücksprungadresse. Sie und das Nested-Task-Flag NT im EFlags-Register werden verwendet, wenn ein Task als »Unterprogramm« und nicht als eigenständiger Task aufgerufen wurde.

19.1.4 Der Task-State-Segmentdeskriptor

Da der Task-State in einem eigenen Segment, dem Task-State-Segment, verzeichnet ist, muß die Information über das Vorhandensein eines solchen Segments in der globalen Deskriptortabelle (GDT) verzeichnet sein. Genau dazu dient der Task-State-Segment Descriptor:

Task-State-Segment Descriptor



A* = AVL

Wie an Bit 12 (System-Bit) des zweiten Doppelworts (Dwords) erkennbar ist, beschreibt der TSS-Deskriptor ein Systemsegment (S = 0). Die Bits 31 bis 16 des ersten DWords sowie 31 bis 24 und 7 bis 0 des zweiten DWords bestimmen die Basisadresse des TSS, dessen Größe findet sich in den Bits 15 bis 0 des ersten DWords sowie in den Bits 19 bis 16 des zweiten. Ferner gehört hierzu auch Bit 23 des zweiten DWords, das sogenannte *Granularity-Bit G*: bei gelöschtem Bit *G* sind die Zellen 1 Byte groß, bei gesetztem 2^{12} = 4.096 Byte. Auf diese Weise lassen sich Segmentgrößen von 1 MByte (*G* = 0; von 0 bis 1 MByte in 1-Byte-Schritten) bis 4 GByte (*G* = 1; von 4 kByte bis 4 GByte in 4-kByte-Schritten) verwalten.

Das Feld DPL (*Define Privileg Level*) spielt für die Schutzkonzeption, das Bit *P*, *Segment Present*, für den Paging-Mechanismus im Protected-Mode eine Rolle. Das *AVL* (*Available*) Bit kann von der Software (Betriebssystem) für eigene Zwecke benutzt werden. Das Busy-Flag B informiert, ob der Task, den der TSS-Deskriptor beschreibt, gerade aktiv (busy) ist, also ausgeführt wird, oder auf die Ausführung wartet. Bei gesetztem Busy-Flag ist der Task aktiv, andernfalls nicht.

19.1.5 Das Task-Register des Prozessors

Welcher Task ist nun im Moment aktiv? Anders ausgedrückt: Welches aller in der GDT verzeichneten TSS ist das aktuelle? Auskunft darüber gibt das Task-Register (TR) des Prozessors:

Task-Register

In den 16-Bit-Anteil des TR wird mittels des Befehls LTR ein Selektor eingelesen. Dieser Selektor zeigt auf einen Task-State-Segmentdeskriptor in der globalen Deskriptortabelle, der das gewünschte Task-State-Segment genauer beschreibt. Dieser Deskriptor, d.h. die enthaltene Basisadresse des Segmentes, dessen Größe sowie Informationen zu Zugriffsrechten etc., wird nun beim Laden des Selektors automatisch in den »unsichtbaren« Cache des Registers geladen. Auf diese Weise müssen nicht bei jedem Zugriff auf das TSS die GDT ausgelesen und die entsprechenden Attribute bestimmt und Adressen berechnet werden. Der Prozeß erfolgt ganz analog zur Belegung eines Segmentregisters mit einem Segmentselektor. Bei einem Task-Switch im Rahmen des Multitasking wird der Inhalt dieses Registers verändert.

19.1.6 Task Switches

Ich will nun nicht mehr tiefer ins Detail gehen, was das Multitasking des Betriebssystems betrifft. An dieser Stelle will ich lediglich schematisch zusammenfassen, was bei den als Task-Switches bezeichneten Wechseln des aktuellen Tasks erfolgt.

Gehen wir von dem Zustand aus, daß ein aktueller Task läuft und das Betriebssystem nun einem anderen Task zur Ausführung verhilft. Wie geht das? Im Prinzip ganz einfach durch einen banalen JMP- oder CALL-Befehl. Die für den Task-Switch verantwortliche Routine des Betriebssystems ruft einen Task genau so auf, wie ein Programm seine Routinen aufruft. Tasks könnte man somit als »Unterprogramme« des Betriebssystems bezeichnen.

Doch es gibt einen wesentlichen Unterschied. Während ein Programm sein Unterprogramm aufruft, indem eine bestimmte Codestelle angesprungen wird, die genau bekannt ist, ist das bei einem Task-Switch nicht der Fall. Denn wie wir eben gesehen haben, ist ja der Inhalt des Program Counter, also der anzuspringende Punkt im Task, von der

Zeit abhängig. Also kann keine feste Adresse übergeben werden. Doch wir haben ja das TSS, in dem die notwendige Information steht. Diese TSS steht in der GDT. Also müßten wir dem CALL- oder JMP-Befehl eigentlich nur den Selektor übergeben, der auf das richtige TSS in der GDT zeigt – und genau das erfolgt auch!

Im Rahmen des dann stattfindenden Task-Switches erfolgt der Reihe nach folgendes:

- Überprüfung der Zugriffsberechtigungen (Schutzmechanismen). Das bedeutet, daß der Prozessor zunächst einmal prüft, ob der Task überhaupt ausgeführt werden darf. Nun macht das zunächst keinen Sinn! Denn wer, wenn nicht das Betriebssystem, das den Task-Switch durchführt, darf denn überhaupt die Tasks umschalten? Und wer hätte denn mehr Privilegien als das Betriebssystem? Aber wir werden noch sehen, daß diese Prüfung sinnvoll ist. Denn es gibt auch Situationen, in denen nicht das Betriebssystem einen Task-Switch veranlaßt.
- Verschiedene Prüfungen, auf die an dieser Stelle nicht näher eingegangen werden soll. Es handelt sich um Überprüfungen auf Zulässigkeit eines Task-Switches im Rahmen der Schutzkonzepte des Protected-Mode, auf Verfügbarkeit und Validität der Inhalte des TSS etc.
- Prüfung, ob alter und neuer Task sowie alle Segmentdeskriptoren, die am Task-Switch beteiligt sind, im Systemspeicher vorhanden sind.
- Sicherung des Task-State des alten Tasks im Task-State-Segment. Hierzu liest der Prozessor das Task-Register aus und benutzt den darin enthaltenen Selektor als Index in die Deskriptortabelle (GDT oder LDT, je nach Angabe in Selektor). In das Task-State Segment, das durch den korrespondierenden Deskriptor bezeichnet wird, trägt er dann den Task-State des abzubrechenden Tasks ein.

An dieser Stelle hat sich der Prozessor zum Task-Switch »verpflichtet«. Hat sich bis hierher ein »Unrecoverable Error« ergeben, wird der Task-Switch nicht durchgeführt, und die entsprechenden Exceptions werden generiert. Ab diesem Punkt führt der Prozessor den Task-Switch durch! Fehler, die ab hier auftreten, führen zu einer Exception, die unmittelbar vor der ersten Instruktion des neuen Tasks generiert wird. Die nächsten Schritte sind:

- Laden des Task-Registers mit dem Segment-Selektor des neuen Tasks, der dem JMP- oder CALL-Befehl übergebenen wurde. Das Busy-Flag im Task-State-Segmentdescriptor des neuen Tasks wird gesetzt sowie das TS-Flag im Kontrollregister CR0 (= »Task-Switched«: der Task wurde umgeschaltet).

- Das Task-Register weist nun auf das neue Task-State-Segment, aus dem die notwendigen Informationen über den neuen Task ausgelesen und in die entsprechenden Prozessorregister geladen werden.

Das war's. Eigentlich ganz einfach, aber wie immer steckt die Tücke im Detail!

Doch kommen wir noch einmal auf die Frage zurück, wer Task-Switches überhaupt veranlassen darf. Das Betriebssystem – sicher! Denn sonst hätten wir kein Multitaskingsystem. Aber unter bestimmten Bedingungen können auch Anwendungen Task-Switches veranlassen. Denn wir haben weiter oben ja festgestellt, daß ein Task nicht nur ein Programm ist, sondern auch aus (Betriebssystem-) Routinen, *Interrupt-* und *Exception-Handlern* bestehen kann. Diese müssen prinzipiell auch von anderen Programmen aufrufbar sein. Was jetzt wieder die Problematik des Betriebssystems auf den Plan ruft, »fremde« Programme oder Programmteile voneinander abgrenzen zu können und zu müssen. Gut – wir haben gesehen, daß das Betriebssystem immer im Adreßraum des aktuellen tasks eingeblendet ist: An Adressen oberhalb von 2 GByte. Aber es könnte ja auch notwendig werden, fremde »Programme« oder Teile von ihnen anspringen zu können, die dann nicht unbedingt Teil des Betriebssystems sind und selbst die »unteren« 2 GByte belegen. Genau dann haben wir den Konflikt mit dem Protected-Mode. Denn der ist ja dafür zuständig, Prozesse gegeneinander so abzuschotten, daß sie eben nicht aufeinander zugreifen können.

Die Lösung heißt »Türen und Tore«. Der Adreßraum eines Tasks hat Türen, durch die jemand anderes ihn betreten kann. Man nennt sie Gates. Sie lassen den kontrollierten und kontrollierbaren Zugang »Fremder« zu. Wir werden im nächsten Kapitel noch etwas näher darauf eingehen. Halten wir daher für den Augenblick und in Hinblick auf einen Task-Switch fest: Die Umschaltungen von einem Task auf den anderen können auf vier verschiedenen Wegen erfolgen:

- Ausführen eines direkten oder indirekten Sprungs mittels JMP oder Aufruf als Unterprogramm durch CALL. Das Argument der beiden Befehle sind Selektoren, die auf einen TSS-Deskriptor in der GDT verweisen.
- Ausführen eines Sprungs oder Aufruf als Unterprogramm, wobei als Argument der Befehle ein Selektor übergeben wird, der auf einen *Task-Gate-Deskriptor* in der aktuellen *lokalen* Deskriptortabelle (LDT) zeigt.

- Behandlung eines Interrupts oder einer Exception, wobei der dazugehörige Eintrag in der Interrupt-Deskriptortabelle (IDT) ein Task-Gate-Deskriptor ist.
- Rückkehr aus dem aktuellen Task mittels IRET, falls es sich um »Nested Tasks« handelt und das NT-Flag in EFlags gesetzt ist.

Was in jedem Fall identisch ist, ist das Verhalten des Betriebssystemteils, das für den eigentlichen Switch verantwortlich ist – und das wir weiter oben kurz skizziert haben.

19.2 Gates

19.2.1 Tore zu fremdem Terrain

Wie wir eben gesehen haben, kann es sehr wohl Gründe geben, daß verschiedene Tasks irgendwie miteinander in Verbindung treten können. Wir haben bereits angedeutet, daß das über Gates geregelt werden kann. Der Begriff Gates bezeichnet quasi ein Tor zu einem fremden Prozeß.

Abhängig von der Funktion, die solche Tore haben sollen, gibt es vier verschiedene Gate-Typen:

- Call-Gates; dieser Gate-Typ wird verwendet, wenn segmentfremde Codeteile angesprungen werden sollen und bildet den im Real-Mode einfachen Programmaufruf mittels CALL oder JMP nach. Call-Gates werden durch Call-Gate-Deskriptoren beschrieben.
- Task-Gates; diesen Gate-Typ haben wir im letzten Abschnitt kennengelernt. Er wird verwendet, um im Rahmen des Multitasking andere Tasks ausführen zu lassen, also eigenständige Prozesse, die über Routinen hinausgehen.
- Interrupt-Gates dienen zur Ausführung von Routinen im Rahmen einer Interrupt- oder Exception-Behandlung.
- Trap-Gates spielen beim Debuggen eines Programms eine Rolle.

Was ist nun ein Gate genau? Was hat man sich darunter vorzustellen? Dazu müssen wir zunächst einmal feststellen, was wir alles brauchen, um auf ein Segment allgemein, auf ein fremdes Segment insbesondere zurückgreifen zu können. Zunächst brauchen wir eine Einsprungadresse. Egal, ob ein Prozeß, eine Routine oder ein *Handler* aufgerufen werden soll, es muß bekannt sein, an welcher Stelle im Segment mit der Ausführung begonnen werden soll. Das Sprungziel muß also bekannt sein. Das wiederum bedeutet, daß das Segment bekannt sein muß, das angesprungen

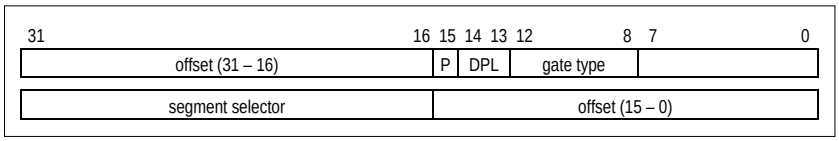
wird, und innerhalb dieses Segments ein Offset. Segmente werden, so wissen wir, über Selektoren gewählt. Und der Offset ist einfach eine 32-Bit-Adresse.

Zweitens gibt es offensichtlich unterschiedliche Gate-Typen. Das bedeutet, daß wir Informationen über den Typ des Gates haben müssen. Außerdem müssen wir – nicht zu vergessen – die Schutzprinzipien berücksichtigen: Welche Privilegien muß ein Prozeß haben, um auf das Gate zugreifen zu dürfen? Mit anderen Worten: Ein Gate ist eigentlich nichts anderes als eine Struktur, die beschreibt, an welcher Stelle unter welchen Voraussetzungen ein fremdes Codesegment angesprungen werden darf – und kann.

19.2.2 Gate-Deskriptoren

Wie wir bislang an hinreichend vielen Beispielen gesehen haben, ist immer dann, wenn irgend etwas näher beschrieben werden soll, ein Deskriptor im Spiel. So auch in diesem Fall. Es gibt Gate-Deskriptoren. Gate-Deskriptoren haben folgende allgemeine Struktur:

Gate-Deskriptor

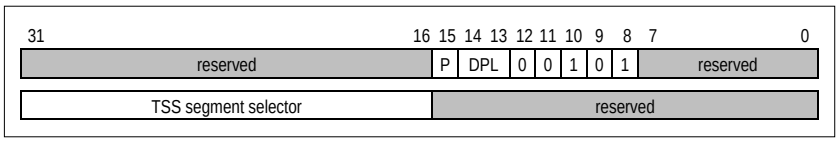


Das Feld DPL (*Define Privileg Level*), die Bits 13 und 14 des zweiten DWords, dient zur Angabe, welche *Privilegebene* das Segment besitzt, und ermöglicht so eine Überprüfung auf legitime Zugriffe auf das Segment im Protected-Mode. Das Bit *P*, *Segment Present*, zeigt an, ob das Segment zur Zeit verfügbar ist ($P = 1$) oder nicht (z.B. weil das Segment als nachladbar gekennzeichnet ist und zwecks Bereitstellung von RAM aus diesem gelöscht worden war). Falls $P = 0$ ist, generiert der Prozessor eine *»Segment Not Present«-Exception*, falls ein Selektor in ein Segmentregister geladen wird, der auf den vorliegenden Deskriptor zeigt. Hierdurch kann im Rahmen des *Exception-Handlers* dafür Sorge getragen werden, das betreffende Segment verfügbar zu machen. Sollte P gelöscht sein, so sind die Bits 31 bis 0 des ersten DWords sowie die Bits 31-16 und 7 bis 0 des zweiten DWords für die Software verfügbar. Betriebssysteme können hier z.B. verzeichnen, woher das »nicht anwesende« Segment geladen werden kann.

Das Feld Gate Type gibt an, um welchen Gate-Typ es sich bei dem betrachteten handelt. Es gibt, wie gesagt, vier Arten von Gates:

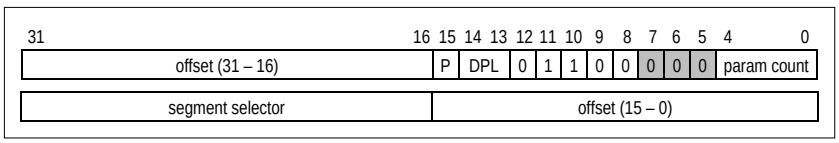
Task-Gates (00101) dienen zum Task-Switch in Multitasking-Umgebungen. ACHTUNG: in Task-Gates sind die Bits 31 bis 16 des zweiten DWords und die Bits 15 bis 0 des ersten DWords reserviert, da sie nicht auf eine Einsprungadresse im Segment zeigen (können; denn der jeweilige Task wird ja nicht an bestimmten, genau definierten Stellen unterbrochen. Also muß im Task-Segment selbst der jeweilige Programmzeiger verzeichnet sein!). Vielmehr zeigt der Segmentselector auf ein Task-State-Segment (TSS), in dem alle zur Ausführung eines Tasks notwendigen Informationen verzeichnet sind.

Task-Gate-Deskriptor



Call-Gates (01100); mit Call-Gates können Routinen in unterschiedlichen Schutzstufen aufgerufen werden. Die Bits 7 bis 5 des zweiten DWords sind auf 0 gesetzt, die Bits 4 bis 0 desselben DWords repräsentieren einen Zähler:

Call-Gate-Deskriptor

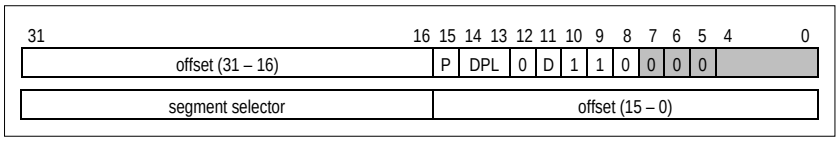


Dieser Zähler heißt im Falle der Call-Gates *ParamCount*. Falls über dieses Gate ein Task-Switch erfolgt, so gibt er an, wie viele Parameter vom Stack des rufenden auf den Stack des zu rufenden Programmteils kopiert werden sollen.

Die Bits 31 bis 16 des ersten DWords des Call-Gate-Deskriptors beherbergen den Selektor, der auf den Segmentdeskriptor des Codesegments verweist, das nun angesprungen werden soll. Die Bits 15 bis 0 des ersten DWords sowie die Bits 31 bis 16 des zweiten bilden eine lineare 32-Bit-Adresse, die den Offset des Einsprungpunkts in das Segment repräsentiert.

Interrupt-Gates (01110 und 0011) werden benutzt, falls ein Interrupt zu bedienen ist. Die Bits 7 bis 5 des zweiten DWords sind auf 0 gesetzt, die Bits 4 bis 0 desselben DWords sind reserviert.

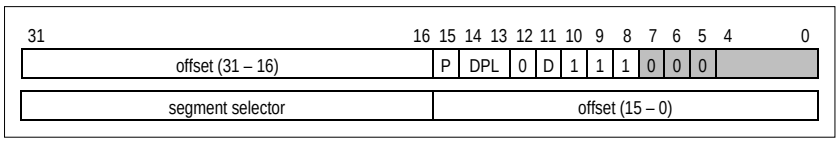
Interrupt-Deskriptor



Wie bei Call-Gates auch beherbergen die Bits 31 bis 16 des ersten DWords des Call-Gate-Deskriptors den Selektor, der auf den Segmentdeskriptor des Codesegments verweist, das nun angesprungen werden soll. Die Bits 15 bis 0 des ersten DWords sowie die Bits 31 bis 16 des zweiten bilden ebenfalls eine lineare 32-Bit-Adresse, die den Offset des Einsprungpunkts in das Segment repräsentiert.

Trap-Gates (01111 und 00111) werden beim Debuggen von Programmen benutzt. Auch hier sind die Bits 7 bis 5 des zweiten DWords auf 0 gesetzt, die Bits 4 bis 0 desselben DWords reserviert.

Trap-Gate-Deskriptor



Auch hier besitzen die restlichen Felder die Informationen über den Einsprungpunkt in das anzuspringende Segment, wie bei Call-Gates und Interrupt-Gates auch.

19.2.3 Das rechte Tor zur rechten Zeit

Warum gibt es vier unterschiedliche Gates, was unterscheidet die Gates voneinander? Bei der Betrachtung der Strukturen eben sollten zumindest ein paar Unterschiede deutlich hervorgetreten sein. Task-Gates sind geringfügig anders aufgebaut als die anderen drei Typen: es gibt keine Einsprungadresse! Denn wie wir von der Besprechung der Tasks her gesehen haben, ist der EIP in Tasks zeitabhängig und alles andere als konstant. Da aber ein Gate nichts anderes als ein Zeiger auf ein Segment ist, der etwas anders aufgebaut ist als ein Selektor, zeigt ein Task-Gate auf ein TSS – und damit indirekt auf eine Einsprungadresse. Task-Gates sind also nicht wesentlich mehr als ein Selektor, jedoch einer, der in einer Deskriptortabelle verzeichnet ist.

Auch Call-Gates unterscheiden sich geringfügig von den anderen Gates. So kann bei diesem Gate als einzigem eine bestimmte Information über Parameter angegeben werden: im Feld ParamCount. Wozu? Üblicherweise sollen ja mittels des CALL-Befehls Routinen aufgerufen werden. Diese Routinen sollen aber häufig bestimmte Werte als Parameter übergeben bekommen. Wie wir wissen, werden solche Parameter über den Stack an die aufgerufene Routine übergeben. Soweit zu den Verhältnissen im Real-Mode. Im Protected-Mode dagegen hat jeder Task seinen eigenen Stack! Folglich gibt es nur zwei Möglichkeiten: im Protected-Mode muß es zwei Arten des Unterprogrammaufrufs geben: eine für Routinen im gleichen Segment, bei dem dann die Parameterübergabe wie gewohnt über den Stack läuft, und eine für »Interprozeßaufrufe«, wo die Übergabe irgendwie anders zu erfolgen hätte. Oder man geht den einfacheren und offensichtlicheren Weg: Man kopiert beim Task-Switch einfach die Parameter, die der CALL-Befehl auf den Stack des »alten« Tasks gelegt hat, auf den Stack des neuen – und kann dann so weitermachen wie gewohnt. Genau dies wird bei Call-Gates realisiert: das Feld ParamCount gibt an, wie viele Doppelworte (im Falle eines 32-Bit-Call-Gates) oder Worte (im Falle eines 16-Bit-Call-Gates) vom Stack des rufenden auf den Stack des gerufenen Tasks kopiert werden sollen. Auf Call-Gates gehen wir bei der Besprechung der Schutzkonzepte noch einmal ein.

Interrupt-Gates brauchen keine Parameter zu übergeben. Sie definieren auch eine genau bekannte, konstante Einsprungadresse. Darum unterscheiden sie sich wiederum von den vorangegangenen Gates. Ebenso wie Trap-Gates. Nun könnte man auf die Idee kommen, doch beide Gates über ein Call-Gate zu realisieren, bei dem einfach das Feld ParamCount auf »0« gesetzt wird. Aber Interrupt- und Trap-Gates verhalten sich zu Call-Gates genau so wie ein INT-Befehl zum CALL-Befehl: bestimmte Flags spielen eine bestimmte Rolle. Im Falle der Interrupt- und Trap-Gates sind es die Flags TF und IF des EFlags-Registers. So legen beide Gate-Typen die Flags auf den Stack des rufenden Tasks und löschen anschließend das TF-Flag. Das Löschen dieses Flags sorgt dafür, daß im Falle eines Interrupts die Befehle des *Handlers* im Trap-Mode ausgeführt werden. Interrupt-Gates gehen noch einen Schritt weiter: Sie löschen das IF-Flag in EFlags und verhindern dadurch, daß die Abarbeitung eines Interrupt-Handlers durch einen neuen Interrupt unterbrochen werden kann. Umgekehrt wird bei der Rückkehr aus dem Handler das IF-Flag wieder gesetzt. Wenn Sie so wollen, realisiert also der Aufruf eines Interrupt-Gates einen Real-Mode-Interrupt im Protected-Mode, während ein Trap-Gate im Rahmen eines Handlers für den INT3 genutzt werden kann, also speziell für den Trap-Mode zugeschnitten wurde. Auf Interrupt- und Trap-Gates gehen wir im Rahmen der Besprechung von Interrupts und Exceptions im nächsten Kapitel noch einmal ein.

19.3 Interrupts und Exceptions

Zu den guten, alten DOS-Zeiten waren Interrupts ein wesentlicher Bestandteil eines jeden Programms, das etwas auf sich hielt. So gab es kaum Programme, die nicht in irgendeiner Weise selbst Hand an die Interruptvektoren gelegt und sie verbogen haben. Aus diesem Grunde ist auch im Anhang ein Kapitel diesen DOS-Interrupts gewidmet. Wenn man sich einmal an die kleinen, unwesentlichen Unterschiede bei der Nutzung von Interrupts gewöhnt hatte, war ihre Verwendung mittels INT xx und IRET genauso einfach wie der Aufruf eines Unterprogramms mit CALL und RET. Die Eingriffe in die Interrupttabelle und das Verbiegen von Interrupts auf eigene Programme waren ein Kinderspiel – was sicherlich nicht zu einer Verbesserung der friedlichen Koexistenz unterschiedlicher Module beigetragen hat.

Schon unter Windows 3.x war die Nutzung von Interrupts wesentlich eingeschränkt. Unter diesem Betriebssystem war das DPMI, das DOS Protected-Mode interface, dafür zuständig, bestimmte Interrupts, die das Betriebssystem bediente, an dieses Betriebssystem weiterzureichen, andere dagegen an DOS und/oder BIOS durchzugeben. Mit Hilfsmitteln wie GlobalDOSAlloc, dem simulate Real-Mode DPMI-Interrupt $31, Funktion $300 oder über DOS3Call waren die meisten der früheren DOS-Interrupts irgendwie verfügbar. Mit der DPMI-Speicherverwaltung war auch eine »friedliche« Koexistenz bei Zugriffen auf BIOS- und DOS-Datenbereiche möglich. Dennoch hatte das Betriebssystem eine Kontrollfunktion: Eine uneingeschränkte Nutzung der Interrupts wie unter DOS war schon nicht mehr möglich.

Unter Windows 9x und NT dagegen sieht es, was Interrupts betrifft, düster aus. Nicht nur, daß man nicht mehr unter »Umgehung« des Betriebssystems in die Interrupttabelle eingreifen kann (die es in etwas anderer Form immer noch gibt, auch wenn sie nicht mehr da liegt, wo sie früher gelegen hat). Auch der Aufruf von Interrupts wird vom System kontrolliert und gegebenenfalls unterbunden! Das aber bedeutet zweierlei: Das Betriebssystem muß erheblich mehr Funktionen dokumentiert zur Verfügung stellen, die die diversen Interrupts nutzen. So ist z.B. eine Konsequenz dieser eingeschränkten Nutzung die Implementation von Windows-Hooks. Andererseits muß der Anwender lernen, sich mit dem zufriedenzugeben, was das Betriebssystem bietet, auch wenn es noch so viele Verrenkungen bedeutet. (Was einige Hartgesottene dennoch nicht von der Nutzung undokumentierter Windowsfunktionen und anderen Vergewaltigungen des Betriebssystems abhält. Manchmal sogar zurecht!)

Mit den modernen Betriebssystemen hat sich auch ein weiterer Begriff durchgesetzt: die Exceptions. Was es mit Interrupts und Exceptions unter Windows 9x/NT auf sich hat, erklären wir im folgenden:

19.3.1 Interrupts

Interrupts sind Nötigungen! Durch einen Interrupt wird das Betriebssystem dazu genötigt, die »normale« Befriedigung seiner eigenen Bedürfnisse und das Ausführen von einem oder mehreren Programmen im Rahmen des Multitasking zu unterbrechen und die Verantwortung an eine bestimmte Routine oder gar ein bestimmtes Programm abzugeben, das man üblicherweise als Handler bezeichnet und das auf die Nötigung reagieren soll. Interrupts erfolgen immer dann, wenn ein Ereignis die unmittelbare Aufmerksamkeit des Prozessors benötigt. Solche Ereignisse können vielfältiger Natur sein: Das Drücken einer Taste der Tastatur, eine Mausbewegung, ein über ein Modem eintreffendes Byte, das seiner Bearbeitung harrt, oder die Nachricht eines Bausteins, daß etwas passiert ist. Allen Interruptquellen ist gemein, daß sie unvorhergesehen und unregelmäßig auftreten, also zu nicht festgelegten Zeiten oder Intervallen, und daß sie üblicherweise keine Verzögerung der Bearbeitung erlauben.

Interrupts sind also keine Ausnahmeerscheinungen in der Hinsicht, daß sie auf Fehler aufmerksam machen wollen oder darauf reagieren würden. Das Drücken einer Taste, das Bewegen der Maus, das Empfangen oder erfolgreiche Senden eines Datums sind keine Fehlerzustände.

Diese Art der Interrupts nennt man Hardware-Interrupts, da die Hardware sie auslöst: Tastatur oder Maus, Timerbaustein oder Schnittstelle, Netzwerkkarte oder Festplatte. Man kennt auch, wie unter DOS, Software-Interrupts. Das sind Interrupts, die von der Software, also von Programmen, ausgelöst werden. Die prominentesten sind die Bereichsprüfung (INT 05, ausgelöst durch den BOUND-Befehl), die Überlaufprüfung (INTO bzw. INT 04) oder der Breakpoint im Rahmen des Debuggens (INT3 bzw. INT 03).

Intel hat die Interruptvektoren 0 bis 31 auch im Protected-Mode (analog zum Real-Mode) für Hardware-Interrupts reserviert und die Interruptvektoren 32 bis 255 zur Nutzung freigegeben. (Was nicht zwingend bedeutet, daß sie auch für Sie zur Verfügung stehen! Immerhin hat das Betriebssystem auch Interessen und – vor allem im Protected-Mode – eigene Vorstellungen davon, was dem Anwender erlaubt ist und was nicht.)

19.3.2 Exceptions

Kommen wir kurz zum Begriff Exception. Exceptions sind Ausnahmesituationen. Solche Situationen können eintreten, wenn ein Programm auf Daten zuzugreifen versucht, die ihm verboten sind, oder wenn Adressen angesprungen werden sollen, die es nicht gibt, oder aber ein Befehl auf ein Datum angewendet werden soll, für das er nicht gilt – denken Sie an das Laden eines Byte-Registers mit einem DWord oder die Division durch 0. Kurz: Eine Exception ist ein Zustand, der eigentlich gar nicht eintreten dürfte, aber nun einmal eingetreten ist und der irgendwie aus der Welt geschafft werden muß.

Was ist nun der Unterschied zu Interrupts? Zunächst einmal der Grund, der zur Auslösung geführt hat. Bei Interrupts ist, wie gesagt, kein Fehler die Ursache, sondern ein entweder hardwarebedingtes oder softwaregesteuertes Ereignis. Bei Exceptions ist die Ursache immer ein Fehler. Von dieser Definition einmal abgesehen, gibt es keine weiteren Unterschiede. Denn für jeden Interrupt wie für jede Exception gibt es eine als Handler bezeichnete Routine, ja sogar ganze Tasks, die die Quelle irgendwie ruhigstellen soll. Auch der Mechanismus, mit dem diese Handler aufgerufen werden, ist der gleiche: über Interrupts. Somit ist die Unterscheidung eigentlich unnötig – und sie wird von vielen auch gar nicht angestellt.

Dennoch hat sich für die Interrupts mit den Nummern 0 bis 18 die Bezeichnung Exception eingebürgert. Denn diese Interruptnummern bezeichnen Handler, die Ausnahmesituationen regeln sollen. Die Interrupts im engeren Sinne des Wortes schließen sich daran an – sie werden vom Betriebssystem verwaltet und reagieren auf die »Nötigungen«, die im Rahmen des Betriebes des Computers eben so anfallen.

Gemeinsam ist beiden Arten, wie gesagt, die Arbeitsweise der Handler sowie deren Aufruf. Und dazu gibt es im Protected-Mode wieder einmal eine Tabelle, wie sollte es auch anders sein!

19.3.3 Die Interrupt-Deskriptortabelle (IDT)

Die Interrupt-Deskriptortabelle ist eine Struktur, die Informationen zu den Handlern der möglichen Interruptquellen beinhaltet. Es gibt bei den Intel-Prozessoren prinzipiell 256 mögliche Interruptquellen, auf die in geeigneter Weise reagiert werden muß, so daß die IDT maximal 256 Einträge haben kann (aber nicht notwendigerweise haben muß).

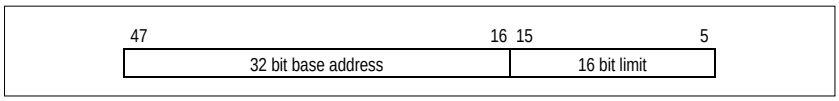
Welche Informationen enthält nun die IDT? Nun – Interrupts und Exceptions sind Ereignisse, die unabhängig von tasks eintreten können und eintreten. Somit gehören sie nicht zu einem Task, wenn sie auch in der Regel im Umfeld eines tasks gehandelt werden. Das aber bedeutet, daß sie in eigenen Segmenten untergebracht sind. Und wann immer im Protected-Mode unterschiedliche Codesegmente eine Rolle spielen, spielen auch die Schutzkonzepte eine Rolle. Das heißt zwangsläufig: Ein Zugriff kann nur sehr kontrolliert erfolgen. Solche kontrollierten Zugriffe haben wir mit den Gates kennengelernt.

Nun sind aber Gates (korrekter gesagt: Gate-Deskriptoren) nichts anderes als Zeiger auf Deskriptoren – erweitert um ein paar Informationen, die die Schutzkonzepte unterstützen. Also genau das, was man im Rahmen der Interruptbehandlung braucht. Daher wird es vermutlich niemanden wundern, daß die IDT eine Tabelle von Gates ist – für jeden Interrupt, der Verwendung findet, ein Gate. Schön sauber nach der Nummer des dazugehörenden Interrupts geordnet. Und wohin zeigen die Zeiger der Gate-Deskriptoren? Na, auf Segmente! Da diese Segmente, genauso wie beim Task-Switch, global und immer verfügbar sein müssen, müssen auch sie in der GDT angesiedelt werden.

Ganz unterschiedliche Gates kommen zum Einsatz. So werden wohl viele Interrupts von Interrupt-Gates Nutzen ziehen – die automatische Flagbehandlung macht sie eben prädestiniert. Wenn jedoch ein Interrupt-Handler in seiner Arbeit selbst von Interrupts unterbrochen werden darf (und vielleicht sogar aus verschieidenen Gründen sollte!), so können auch Trap-Gates zum Einsatz kommen. Sie unterscheiden sich ja nur unwesentlich von Interrupt-Gates. Wenn im Rahmen von SoftwareInterrupts vielleicht sogar ganze Tasks ausgeführt werden sollen, so kann auch ein Task-Gate angesprochen werden. Nur ein Gate – das Call-Gate – kann und darf nicht verwendet werden und wenn man es genauer betrachtet, sogar zu Recht!

Da es nur 256 mögliche Interruptquellen und somit maximal 256 Gate-Deskriptoren gibt, die in die IDT eingetragen werden, ist diese maximal 2 kByte groß. Wie die GDT auch, ist die IDT kein Segment und besitzt daher keinen Deskriptor. Anders als bei der GDT ist jedoch bei der IDT der erste Eintrag ein gültiger Deskriptor auf einen existenten Handler: den des Interrupts 0. Die IDT wird analog zur GDT mit dem Befehl LIDT bei Booten des Systems geladen und kann mittels SIDT gesichert werden. Das IDTR ist analog zum GDTR aufgebaut und besitzt wie dieses einen Cache, der den Zugriff auf die IDT beschleunigt.

19.3.4 Das Interrupt-Deskriptortabellenregister des Prozessors



Das Interrupt-Deskriptortabelle Register (IDTR) umfaßt wie das GDTR eine 32-Bit-Adresse sowie einen 16-Bit-Wert. Beide Angaben geben die Lage im Speicher und die Größe der sogenannten Interrupt-Deskriptortabelle wieder. Das Laden des IDTR erfolgt mit dem Befehl LIDT.

Was bedeutet das nun, zusammengefaßt, für den Interrupt bzw. die Exception? Der Prozessor erhält die Information, daß ein Ereignis eingetroffen ist, das den Aufruf des Handlers von Interrupt $xx notwendig macht. Also skaliert der Prozessor die Interruptnummer $xx mit acht, da jeder Gate-Deskriptor 8 Bytes groß ist. Aus dem IDTR liest der die Basisadresse der IDT, addiert die skalierte Interruptnummer als Offset in die Tabelle dazu und entnimmt dieser Stelle die Informationen, die zur Nutzung des korrespondierenden Gates notwendig sind. Dies ist unter anderem der Selektor, der auf den Deskriptor in der GDT zeigt. Diesen nutzt er (mit acht skaliert) ebenfalls wieder als Offset zur Basisadresse der GDT, die er im GDTR findet. An der Stelle steht nun endlich der Deskriptor, der das Segment näher spezifiziert, in dem der Handler residiert.

Handelt es sich um einen ganzen Task, der eingesetzt werden soll, so steht in der GDT ein TSS-Deskriptor. Dann muß in der IDT an der entsprechenden Stelle ein Task-Gate Deskriptor stehen. Ist es jedoch ein Interrupt- oder ein Trap-Gate Deskriptor, so zeigt der auf einen CodeSegmentdeskriptor in der GDT. Im ersten Fall wird somit ein Task-Switch veranlaßt, in den letzten beiden Fällen der Aufruf einer Routine, nicht ohne vorher – wie bei Interrupts üblich – den Stack mit wichtigen Informationen belegt zu haben. Aber das hatten wir bereits alles.

19.3.5 Nutzung von Interrupt und Exceptions

Was heißt das nun alles für uns Programmierer? Zweierlei: es ist vieles beim alten geblieben, aber vieles auch nicht. Alt, d.h. mit den Bedingungen im Real-Mode vergleichbar, ist das Konzept der Interruptvektoren und einer Interrupttabelle. Neu dagegen ist, wie diese Tabelle genutzt wird. Das bedeutet wiederum, daß es bei der Programmierung einer Routine, die lediglich mit einem IRET anstelle

eines RETs abgeschlossen wird, sowie mit dem Eintrag der Adresse dieser Routine in die richtige Position einer Tabelle mit genau bekannter Adresse nicht getan ist.

Ich möchte es an dieser Stelle auch mit der Beschreibung von Interrupts und Exceptions bewendet sein lassen. Denn auf die wirklich interessanten Interrupt-Handler, wie z.B. den für die Tastatur, Maus etc., mit dem man unter DOS vieles anstellen konnte (nicht nur Unsinn!), haben Sie Dank des Betriebssystems eh keinen Zugriff. Die Interrupts, auf die Sie Zugriff haben, sind für Sie wahrscheinlich wenig interessant. Und selbst Interrupts nutzen, um die Möglichkeiten des Betriebssystems zu erweitern, brauchen Sie auch nicht – sie können ja Libraries schreiben, die Sie direkt ins Betriebssystem einbauen. Schließlich kommen Sie auch ohne direkte Interruptnutzung, nämlich über das Betriebssystem, an die weitaus größte Anzahl von Informationen, die über die Interrupts verfügbar gemacht werden können. Und wenn Sie Handler für die Exception »Division durch 0«, für den BOUND-Befehl oder einen Überlaufhandler schreiben wollen, so sei auf weiterführende Literatur zu diesem Thema verwiesen.

19.4 Schutzmechanismen

Wir haben bereits an verschiedenen Stellen über die Schutzkonzepte des Protected-Mode gesprochen. Also sprechen wir sie jetzt einmal genauer an.

Die Schutzkonzeption der Intel-Prozessoren im Protected-Mode greift auf verschiedenen Ebenen. So können bei dem Versuch, auf ein Segment zuzugreifen, folgende Schutzmechanismen eingeschaltet werden:

- Überprüfung von Segmentgrenzen
- Überprüfung des Typs von Segmenten
- Überprüfung der Zugriffsprivilegien für Segmente
- Schutzmechanismen auf Page-Ebene

19.4.1 Die Prüfung der Segmentgrößen

Eine Säule des Protected-Mode und seiner Schutzkonzeption ist die Prüfung der Segmentgrenzen eines Segments. Hierdurch wird gewährleistet, daß auf Daten und Code außerhalb des eigenen Segments nicht unkontrolliert zugegriffen werden kann, wie dies im Real-Mode sehr leicht möglich ist, da Segmente dort lediglich eine Aufteilung des physikalischen Speichers in Strukturen mit 64 kByte Größe sind. Ob diese 64 kByte vollständig von einem Programm benutzt werden oder nicht, kann nicht festgestellt werden – oder besser gesagt: Es wird nicht festgestellt.

Die auf Segmenten aufbauenden Schutzkonzeptionen werden mit dem Umschalten des Prozessors in den Protected-Mode automatisch eingeschaltet. Sobald sich der Prozessor im Protected-Mode befindet, gibt es keine Möglichkeit mehr, diesen Schutzmechanismus auszuschalten – es erfolgt grundsätzlich eine Überprüfung auf die Validität einer Adresse.

Zur Prüfung der Segmentgrenzen im Protected-Mode muß neben der Adresse der Segmentbasis als »untere« Grenze des Segments auch eine »obere« Grenze angegeben werden können. Diese obere Grenze errechnet der Prozessor aus der Segmentbasis und der Größe des Segments. Beide Informationen sind in den Deskriptoren eines Segments verzeichnet. Ein Zugriff auf Adressen, die außerhalb der Segmentgrenzen liegen, ahndet der Prozessor mit einer #GP-Exception.

Das Granularity-Bit G im Segmentdeskriptor steuert die Berechnung der oberen Grenze. Ist dieses Bit gelöscht (»Byte Granularity«), so wird die Segmentgröße (das »Limit-Feld« im Deskriptor) in Bytes interpretiert. Der maximale Wert für ein Limit von $FFFFF (20 Bits) resultiert dann in einer Segmentgröße von 1 MByte. Da die Zählung mit 0 beginnt, beträgt der letzte adressierbare Offset zur Basisadresse eines Segments somit $FFFFF. Im Falle eines gesetzten G-Bits dagegen wird der Eintrag des Limit-Feldes mit 2^{12} (= 4.096; entspricht einer Page) skaliert. Das minimale Limit (Eintrag »0«) resultiert dann in der »unteren« Grenze $1000 (= 4 kByte), das maximale in der »oberen« Grenze $FFFFFFFF (4 GByte). Bei gesetztem G-Bit werden somit Offsets zur Segmentbasis 0 bis $FFF (= 4095) nicht geprüft, sie sind immer gültig! Die maximale Segmentgröße beträgt hier 4 GByte.

Bei Datensegmenten spielt noch das Bit E aus den Deskriptoren eine Rolle. Während bei »normalen« Segmenten die Segment**basis** »festliegt« und die Segmentgröße zu dieser Basis addiert wird, um die Segmentspitze zu erhalten, ist bei Segmenten, bei denen das E-Bit gesetzt ist, sogenannten »Expand-down Data Segments« – z.B. Stacksegmenten, die Segment**spitze** »fest«. Das heißt, daß das Datensegment seine größte Ausdehnung hat, wenn das Limit-Feld den Wert 0 enthält, seine minimale beim Wert $FFFFF.

Überprüfungen der Segmentgrenzen erfolgen auch, wenn das GDTR, IDTR, LDTR oder TR geladen wird. Die 16-Bit-Limits der GDTR und IDTR beschreiben genauso wie die 16-Bit-Limits der LDTR oder TR die maximale Größe, die die Tabellen, auf die die entsprechenden Register zeigen, besitzen. Auf diese Weise wird verhindert, daß unzulässige Zugriffe auf Deskriptoren außerhalb der Deskriptortabellen bzw. Datenstrukturen möglich sind.

Auch der Programmierer kann, wie schon weiter oben gesagt, mittels LSL vor einem eventuellen härteren Check des Prozessors prüfen, wie groß das spezifizierte Segment ist und ob ein Zeiger vielleicht nicht in das Segment zeigt.

19.4.2 Die Prüfung der Segmenttypen

An zwei Stellen finden sich Informationen darüber, um was für ein Segment es sich bei einem betrachteten handelt:

- das System/Application-Flag S im zweiten Doppelwort des Segmentdeskriptors und
- das Type-Feld, die Bits 11 bis 8 im zweiten Doppelwort des Segmentdeskriptors.

Diese Informationen können vom Prozessor vielfältig und bei verschiedenen Aktionen geprüft werden. So kann z.B. bei Befehlen, die zum Laden von Tabellen verwendet werden (LLDT, LTR, etc.) oder bei Befehlen, die Zugriffsrechte ändern (LAR), überprüft werden, ob das Segment, das geladen werden soll, den »richtigen« Typ besitzt. So besitzt ein Segment, das eine LDT enthält, als Typ den Wert »0010« bei gesetztem S-Flag. Werden diese Bedingungen während des Versuchs, mittels LLDT eine »neue« LDT zu laden, nicht vorgefunden, erzeugt der Prozessor eine Exception.

Analog kann z.B. beim Zugriff auf den Speicher mittels eines Segment Override Prefix festgestellt werden, ob das Segment ein Application-Segment ist (S-Flag gesetzt) und tatsächlich ein Datensegment beschreibt (Typ »0xxx«). Beim Aufruf von Routinen mittels JMP oder CALL und der Verwendung von Call-Gates kann mittels S-Flag und Typ geprüft werden, ob tatsächlich in ein Codesegment verzweigt werden soll (Typ = »1xxx«). Darüber hinaus lassen sich die Bit-Kombinationen in diesem Type-Feld zu weiteren Prüfungen verwenden (z.B. read only/read-write bei Daten).

Aber auch der Programmierer kann, wenigstens in geringem Umfang, einige Prüfungen vornehmen. Mittels VERR und VERW kann er beispielsweise prüfen, ob ein Segment, das durch den im Operanden übergebenen Selektor markiert worden ist, für Lese- und/oder Schreibzugriffe frei ist oder nicht.

19.4.3 Zugriffsprivilegien

Ein wichtiges Prinzip bei der Schutzkonzeption im Protected-Mode ist auch die Vergabe von Privilegien. Intel-Prozessoren kennen vier Privilegstufen, die mit den Ziffern 0 bis 3 bezeichnet sind. Die höchste Privilegstufe hat die Nummer 0, die niedrigste die Nummer 3.

Diese Privilegstufen stellen Sicherheitsbereiche dar. Der »äußerste« Bereich ist praktisch ungesichert – in ihm tummeln sich üblicherweise die Anwendungen. Der »innerste« Bereich ist der »Hochsicherheitstrakt«. Zu ihm haben nur ganz ausgewählte »Personen« Zutritt, denn in ihm befindet sich der Betriebssystemkern. Dazwischen können sich noch zwei weitere Stufen mit zu- bzw. abnehmenden Privilegien befinden, die aber bei den verschiedenen Windows-Betriebssystemen (Windows 3.xx, 95/98 und auch NT) nicht genutzt werden.

Privilegstufen ersetzen nicht die anderen Schutzkonzepte und setzen sie auch nicht (teilweise) außer Kraft. Sie sind ein weiteres Prinzip, für größtmögliche Sicherheit zu sorgen.

Es ist nicht ganz einfach, darzustellen, was im Rahmen der Überprüfung der Privilegien abläuft. Versuchen wir daher, uns heranzutasten.

Bei der ersten Annäherung an die Privilegprüfung spielen zunächst einmal zwei Begriffe eine wesentliche Rolle:

Der Current Privileg Level (CPL). Damit wird die Privilegstufe bezeichnet, die das augenblicklich laufende Programm besitzt. Der CPL wird vom Betriebssystem in den Bits 1 und 0 des CS- und des SS-Registers gespeichert, sobald ein Programm oder Task gestartet oder auf einen anderen Task umgeschaltet wird. Somit beinhaltet das CS bzw. SS-Register immer den CPL des aktuellen Tasks.

Der Descriptor Privileg Level (DPL). Hierbei handelt es sich um die Bits 14 und 13 des zweiten DWords eines Segmentdeskriptors. Der DPL gibt an, welche Privilegien für einen Zugriff auf dieses Segment gefordert werden, ab welcher Privilegstufe somit ein Zugriff erlaubt ist. So ist der DPL die niedrigste Privilegstufe (numerisch: die größte Zahl!), die Zugriff auf das Segment erhält: mehr Privilegien bedeuten eine höhere Privilegstufe (numerisch: kleinere Werte) und haben dann in jedem Fall Zugriffserlaubnis.

Der CPL beschreibt also, mit anderen Worten, die Privilegien, die ein Programm oder Task hat, während der DPL die Privilegien beschreibt, die gegeben sein müssen, wenn ein solches Programm oder Task andere Segmente nutzen möchte. Ein Vergleich beider Privilegstufen entscheidet also über die Zugriffserlaubnis. So einfach ist das.

Ein Beispiel: Nehmen wir ein von Ihnen geschriebenes Programm. In der Regel wird das eine Anwendung sein, die somit auf der niedrigsten Privilegstufe abläuft. Das Betriebssystem wird somit das CPL-Feld im CS- und SS-Register auf 3 setzen, sobald Sie Ihre Anwendung starten. Diese Anwendung greift nun auf ein Datensegment zu. Da es

sich um ein Datensegment einer Anwendung handelt, wird das Betriebssystem das DPL-Feld im Datensegmentdeskriptor ebenfalls auf 3 setzen – es gehört ja zu Ihrer Anwendung. Falls nun Ihr Programm auf seine Daten zugreifen möchte, wird CPL = 3 mit DPL = 3 verglichen. Da beide Werte gleich sind, darf Ihr Programm auf seine eigenen Daten zurückgreifen.

Nun möchten Sie aber auf Daten in Segmenten zurückgreifen, die Sie nach Ansicht des Betriebssystems gar nichts angehen, z.B. auf irgendwelche Systemdaten oder -einstellungen. Da es sich hier um Daten des Betriebssystems handelt und dieses mit höchster Priorität (CPL = 0) abläuft, wird auch das entsprechende Datensegment einen DPL von 0 haben. Das ist kein Problem für das Betriebssystem, denn mit CPL = 0 und DPL = 0 hat das Betriebssystem Zugriff. Es ist aber ein Problem für Sie: Denn mit CPL = 3 liegen Sie wertmäßig höher als DPL = 0. Mit anderen Worten: Ihre Privilegien reichen nicht! Der Zugriff auf die Daten ist Ihnen somit untersagt. Ziemlich ungerecht ist dagegen der umgekehrte Fall: Aufgrund der höchsten Priorität hat das Betriebssystem immer Zugriff auf Ihre Daten, da DPL = 3 > als CPL = 0 ist. (Es gibt eben immer welche, die gleicher als andere sind!)

Das ärgert Sie, und Sie versuchen, das zu umgehen! Sie knobeln und detektieren wie Sherlock und finden eine Lösung, wie Sie glauben. So finden Sie Routinen des Betriebssystems, die auf die gewünschten Daten zugreifen können. Diese Routinen nutzt das Betriebssystem selbst, weshalb die Übergabekonventionen genau definiert sind. In Ihrer Genialität haben Sie diese Konventionen herausgefunden. So brauchen Sie nur über den Stack die Adressen von Variablen anzugeben und die richtige Routine aufzurufen.

Dabei stoßen Sie auf ein Problem. Sie haben es nicht mehr mit einem bloßen Zugriff auf Daten zu tun, Sie rufen nun fremden Code auf. Das ist ein weiteres Ereignis, das die Schutzkonzeption auf den Plan ruft. Aber lassen wir das fürs Erste außen vor, und tun wir so, als würde uns das keine Probleme bereiten. Ihre Vermutung ist nun, daß durch den Aufruf der Betriebssystemroutine nicht mehr Ihr Anwendungsprogramm auf das Datensegment zugreift, sondern das Betriebssystem in Form der Routine. Soll heißen, Sie vermuten, daß CPL beim Zugriff auf das Datensegment 0 ist. Das ist auch der Fall! Also: CPL = 0, DPL = 0, keine Probleme! Sie bekommen, quasi über einen Strohmann, doch noch, was Sie wollen. Keine Probleme? Doch! Denn daß Sie so vorgehen würden, haben sich die Entwickler der Prozessoren auch gedacht – und dem einen Riegel vorgeschoben. Dieser Riegel heißt RPL und ARPL.

Unter RPL versteht man den Requestor Privileg Level. Es sind die Bits 1 und 0 des Selektors, der auf das Segment (besser auf dessen Deskriptor in einer der Deskriptortabellen) zeigt, auf das zugegriffen werden soll. In dieses Feld wird eingetragen, welche Privilegstufe der Requestor de facto hat. Üblicherweise steht hier der gleiche Inhalt wie in CPL – was auch logisch erscheint. Aber der RPL kann mit dem Befehl ARPL, adjust RPL, verändert werden. Das tut das Betriebssystem liebend gern. Es verwendet dazu Angaben, die ebenfalls über den Stack übergeben werden, nämlich Rücksprungadressen. Denn die Idee dahinter ist, daß jeder, der Parameter über den Stack übergibt, ja auch die Ergebnisse sehen will, also kundtun muß, wohin es nach der Routine wieder geht. Dadurch wird der Requestor als das entlarvt, was er ist: ein Strohmann.

Durch den Befehl ARPL wird im übergebenen Selektor das RPL-Feld, das von der aufgerufenen Betriebssystemroutine mit dem CPL = 0 der Routine belegt wird, mit dem CPL = 3 in der Rücksprungadresse zum Anwendungsprogramm »überschrieben«. Das bedeutet, daß bei einem Zugriff auf die so heiß ersehnten Daten des Betriebssystems nun der RPL = 3 mit dem DPL = 0 verglichen wird, obwohl eine Betriebssystemroutine auf eigentlich erlaubte Daten zugreift. Resultat: Der Zugriff wird dennoch verweigert, es wird eine #GP ausgelöst.

Was lernen wir daraus? Es ist äußerst schwer, im Rahmen des Protected-Mode mit seinen Schutzkonzepten Zugriff auf Dinge zu bekommen, die einem verboten sind und sein sollten. Bei der Überprüfung der Privilegien spielen also nicht nur der CPL des Rufers und der DPL des Gerufenen eine Rolle, sondern auch der RPL. Läßt der RPL eines Selektors nur den Zugriff auf einer niedrigeren Privilegstufe zu als der CPL, so hat der RPL Vorrang gegenüber dem CPL. Umgekehrt hat der CPL Vorrang, wenn es weniger Privilegien signalisiert als der RPL. Summa: Es gelten grundsätzlich die niedrigeren Privilegstufen aus CPL und RPL. Sie werden gegen den DPL geprüft.

Doch nun noch ein Wort zum Aufruf fremden Programmcodes. Soweit es sich nicht um einen Task-Switch handelt, stehen einem dazu zwei Möglichkeiten offen: die Angabe einer bestimmten Sprungzieladresse als Operand eines CALL- oder JMP-Befehls oder die Angabe eines Call-Gate-Deskriptors als Operand für ein CALL oder JMP. (Beachten Sie hierbei bitte, daß es sich jeweils um FAR-Ziele handeln muß. Near Jumps oder Calls, bei denen keine Segmentangabe erfolgt, bleiben grundsätzlich innerhalb des Segments, auch wenn dies 4 GByte groß sein sollte. Daher unterbleibt in solchen Fällen eine Privilegienüberprüfung!)

Auch bei diesen Programmverzweigungen spielen der CPL des rufenden Codesegments, der DPL des gerufenen Segments sowie der RPL des Selektors, der im Rahmen des FAR-CALLs oder -JMPs übergeben wird, eine Rolle. Die Mechanismen ähneln denen, die für einen Datenzugriff bereits geschildert wurden. In dem Moment, in dem entweder CPL oder RPL eine geringere Privilegstufe angeben als der DPL des gerufenen Segments, wird der Sprung verweigert. Somit hätten wir bereits in dem Beispiel oben das Problem, daß die von uns ausspionierte Betriebssystemroutine von uns gar nicht angesprungen werden kann.

Es gibt aber noch einen weiteren Punkt, der eine Rolle spielt. Das ist das C-Flag im Segmentdeskriptor des Codesegments. Ist dieses gesetzt, so ist das anzuspringende Segment ein Conforming-Segment. Dann kehrt sich alles um! Denn bei »anpassungsfähigen« Segmenten wird ein #GP nur dann erzeugt, wenn der CPL kleiner als der DPL ist. Das heißt: alle Module, deren Privilegien niedriger oder gleich sind als der DPL dürfen, alle anderen dürfen nicht zugreifen! Ein merkwürdiges Verhalten. Denn es besagt, daß der, der höhere Privilegien hat, wie z.B. das Betriebssystem, keinen Zugriff erhält, der aber, der niedrigere oder die gleichen Privilegien hat, also ein Anwendungsprogramm, erhält Zugriff. Der Sinn dahinter ist, daß Conforming-Segmente für Bibliotheken oder Exception-Handler eingesetzt werden können, die zwar für ein Anwendungsprogramm eine Rolle spielen, nicht aber für das Betriebssystem. Das gilt zum Beispiel für Bibliotheken mit mathematischen Funktionen. Durch die Nutzung von Conforming-Segmenten sind diese Routinen dann Teil des Betriebssystems, können aber von Anwendungsprogrammen genutzt werden. Bleibt zu fragen, ob das vielleicht ein Hintertürchen sein könnte, das Betriebssystem zu überlisten. Denn immerhin könnten wir ja auf diese Weise ein Betriebssystemmodul erstellen, hätten also Zugriff auf einen CPL = 0.

Leider aber klappt auch das nicht! Denn der CPL/RPL des Rufers bleibt erhalten, wenn in ein Conforming-Segment gesprungen wird. Das bedeutet, daß selbst innerhalb einer Conforming-Routine (Betriebssystemroutine) nirgendwo ein CPL/RPL = 0 auftritt, also Betriebssystemprivilegien nicht erfüllt sind. Keine Chance, keine Hintertür!

Ich möchte an dieser Stelle nicht tiefer in die Schutzkonzepte und Privilegstufen eindringen, die Prüfungen bei Gates näher erklären oder auf die Konzepte bei Task-Switches eingehen und Sie daher an dieser Stelle auf weiterführende Literatur verweisen, falls Sie hier Wissensbedarf haben sollten.

19.4.4 Schutzmechanismen auf Page-Ebene

Auch auf der »niedrigsten« Ebene, der Ebene der Abbildung virtueller Speicherräume auf reale – dem Paging-Mechanismus also – greifen bestimmte Schutzmechanismen. So gibt es die Unterscheidung zwischen einen Supervisor und einem User. Dies ist hier nicht persönlich zu nehmen: als Supervisor werden Code und Daten verstanden, die grundsätzlich alle Zugriffsrechte haben – das Betriebssystem also. Ein User ist dann eine Anwendung.

Auf der Page-Ebene gibt es zwei Möglichkeiten des Schutzes, die der Prozessor beide zur Verfügung stellt:

- eingeschränkte Nutzung von Seiten in Abhängigkeit des Status (Supervisor oder User) und
- Markierung von bestimmten Seiten mit dem Attribut *read only*.

Der Prozessor überprüft bei einem Zugriffsversuch diese beiden Bedingungen und erzeugt eine Page-Fault-Exception (#PF) in dem Fall, in dem ein Zugriff verboten ist.

Wenn der Prozessor auf einer Privilegstufe mit CPL < 3 arbeitet, ist der Supervisor-Mode eingeschaltet. Andernfalls befindet er sich im User-Mode, was einen eingeschränkten Zugriff bedeutet: Es kann nur noch auf Seiten zugegriffen werden, bei denen das Bit 2 der Page-Table-Entries bzw. des Page-Directory-Entries, das sog. U/S-Flag (*User/Supervisor*) gesetzt ist und somit signalisiert, daß die Seite auch für User frei ist. Ist dagegen Bit 2 gelöscht, so werden für einen erfolgreichen Zugriff die Privilegien eines Supervisors (CPL < 3) gefordert. Andererseits gestattet der Prozessor im Supervisor-Mode grundsätzlich den Zugriff auf alle Seiten, unabhängig davon, ob das U/S-Flag gesetzt oder gelöscht ist.

Der Supervisor-Mode ist auch beim zweiten Mechanismus involviert. Ist das Bit 1 in einem Page-Table-Entry oder einem Page-Directory-Entry, das R/W-Flag, gelöscht, so bedeutet das normalerweise, daß die betreffende Seite nur gelesen werden darf (*read only*). Andernfalls darf sie auch beschrieben werden (*read/write*). Diese Unterscheidung erfolgt aber wiederum nur im User-Mode, also bei einem CPL = 3. Im Supervisor-Mode kann in jedem Fall lesend und schreibend auf alle Seiten zugegriffen werden – selbst auf schreibgeschützte.

Ab dem 80486 gibt es allerdings auch eine Möglichkeit, einen schreibenden Zugriff auf read only Pages im Supervisor-Mode zu verhindern. Sobald Bit 16 in CR0, das sog. Write-Protection-Flag, gesetzt wird, unterbindet der Prozessor einen schreibenden Zugriff auf schreibgeschützte Seiten – allerdings lediglich auf solche, die über das U/S-Flag als User-Page gekennzeichnet sind.

19.4.5 Kommunikation mit der Peripherie: Die Ports

Wie sieht es eigentlich mit der Kommunikation mit der Außenwelt aus? Die Frage sei an dieser Stelle erlaubt. Denn was wir bisher an Schutzkonzepten kennengelernt haben, betrifft lediglich den Adreßraum des Prozessors, der der Verwaltung des Speichers dient. So besitzen Segmente in diesem Adreßraum ein Feld, das die Privilegien angibt, die von jemandem gefordert werden, der darauf zurückgreifen will: den DPL. Ob dieser Jemand diese Privilegien erfüllt, wird ebenfalls über solche Felder geklärt: CPL und RPL.

Gibt es also etwas Vergleichbares für den Adreßraum, der der Kommunikation mit der Peripherie dient? Unterliegen auch die Ports solchen Prüfungen? Ja! Der Protected-Mode hätte seinen Namen zu Unrecht, würde er nicht auch für diesen wesentlichen Bereich Schutzkonzepte vorweisen. Sie funktionieren sogar absolut gleich.

Doch zunächst einmal müssen wir eine kleine Fallunterscheidung machen. Wie wir von weiter oben wissen, gibt es ja zwei Arten von Ports: die »echten« Ports, also die Adressen, die der Prozessor über seine »Peripherieadreßleitungen« ansprechen kann, und die Ports für den Memory-Mapped-Input/Output, die in Wahrheit Adressen im Adreßraum des Speichers sind. Das bedeutet, daß solche Memory-Mapped-Ports eigentlich auf die gleiche Weise geschützt sein sollten wie ganz normaler Speicher. Das sind sie auch: Für Memory-Mapped-I/O gilt bis zum letzten i-Tüpfelchen das, was wir bislang festgestellt haben: DPL, CPL und RPL lassen grüßen.

Was die anderen Ports betrifft, gibt es zwei Möglichkeiten. Zum einen gibt es ein DPL-Pendant. Das heißt IOPL, Input/Output Privileg Level, und ist ein Feld im EFlags-Register. Das aber bedeutet, daß diese Privilegien für alle Ports gelten! Denn es gibt nur ein EFlags-Register, damit nur ein IOPL-Feld, und das ist unabhängig von irgendwelchen anderen Strukturen. Mit dem IOPL läßt sich somit ein genereller Zugriff auf alle Ports regeln: Ist der IOPL numerisch kleiner als der CPL (fordert er also mehr Privilegien als zur Zeit bestehen), so wird der Zugriff auf den Adreßraum der Ports generell verboten. (Übrigens: einen RPL gibt es hier nicht – Portzugriffe erfolgen nicht über Selektoren, in denen das RPL-Feld verzeichnet ist.)

Es gibt nur ein EFlags-Register und nur ein IOPL-Feld. Diese Beschränkung heißt jedoch nicht, daß es für alle Tasks die gleichen Zugriffsbeschränkungen gibt! Denn jeder Task hat eine eigene Kopie des Inhalts des EFlags-Registers, die beim Task-Switch entsprechend verwendet wird: entweder gesichert (alter Task) oder geladen (neuer Task). Die Zugriffe auf die Ports sind also auf Task-Ebene geregelt.

Wer nun kann das IOPL-Feld setzen, somit also Zugriffe auf die Ports beschränken? Im ersten Moment könnte man annehmen, daß dieses Schutzkonzept Augenwischerei ist. Denn immerhin haben wir mit den Befehlen PUSHFD und POPFD ja theoretisch die Möglichkeit, den Inhalt des EFlags-Registers zunächst auszulesen, dann ggf. das IOPL-Feld zu verändern und anschließend wieder in das EFlags-Register zurückzuschreiben. Theoretisch ja – aber praktisch nicht. Was auch gut so ist, denn andernfalls brauchte man ja die ganzen Schutzmechanismen nicht! Der Grund dafür ist, daß der POPF-Befehl in seinen verschiedenen Variationen nur dann das IOPL-Feld im EFlags-Register verändert, wenn der IOPL nicht kleiner als der CPL ist. Das bedeutet: Wann immer z.B. ein IOPL = 1 von irgend jemandem einmal in ein IOPL-Feld eingetragen wurde, haben nur noch Routinen die Möglichkeit, dieses IOPL-Feld zu verändern, die unter CPL = 0 oder 1 laufen, nicht aber Anwendungen, die Sie in der Regel schreiben. Denn die laufen mit CPL = 3 ab. Nun raten Sie einmal, wer die IOPL-Felder in der Regel besetzt!

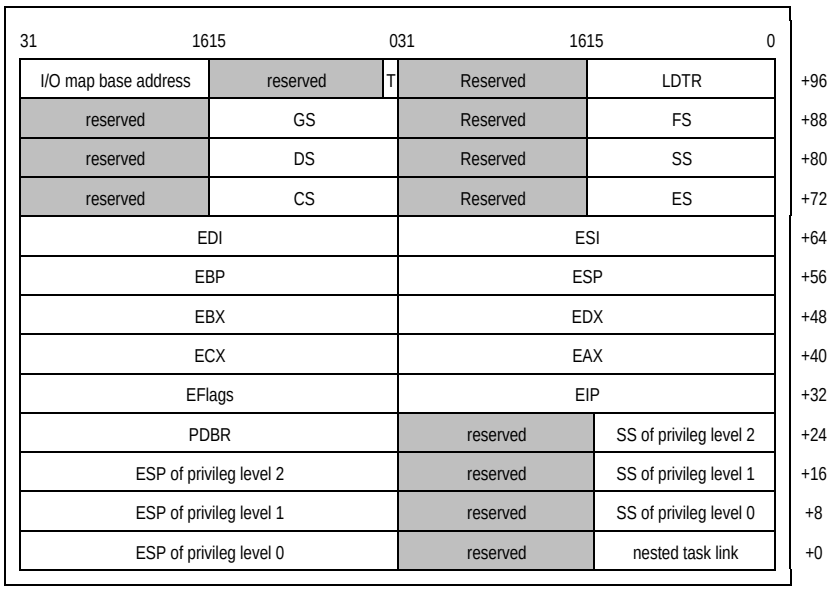
Auch der andere Weg, das EFlags-Register über den IRET-Befehl zu manipulieren, scheitert. Auch IRET ändert wie POPF/POPFD den Inhalt des IOPL-Feldes nur, wenn die aktuellen Privilegien des Tasks mindestens die Forderungen des IOPL erfüllen. Andernfalls bleibt einfach der bisherige IOPL erhalten. (Übrigens wird auch das IF-Flag auf diese Weise geschützt. STI/CLI/POPF und IRET manipulieren nur dann das IF-Flag, wenn IOPL ≤ CPL ist)

Einen Haken hat diese Art der Schutzkonzeption nun aber doch! Auf diese Weise kann nur eine generelle Blockade/Freigabe der Ports erfolgen. Also: entweder alle oder keiner. Das ist sicherlich nicht immer erwünscht. Im Gegenteil: Es dürfte die Ausnahme sein. So sollte es eine Möglichkeit geben, auch gezielt einzelne Ports vor Zugriffen zu schützen und andere eben nicht. Diese Möglichkeit gibt es tatsächlich. Sie wird durch eine Erweiterung des TSS ermöglicht.

Diese Erweiterung besteht in einer Tabelle, die sich an die Strukturen der TSS anschließt, die Sie bisher kennengelernt haben.

Wenn Sie nun die letzten beiden Bytes im TSS näher betrachten, werden Sie ahnen, wie dieses Schutzkonzept aussehen könnte: Es gibt wieder einmal eine Tabelle, in der die zu schützenden Ports verzeichnet sind. Die Adresse dieser Tabelle steht im Feld I/O Map Base Address der TSS.

32-Bit-Task-State-Segment

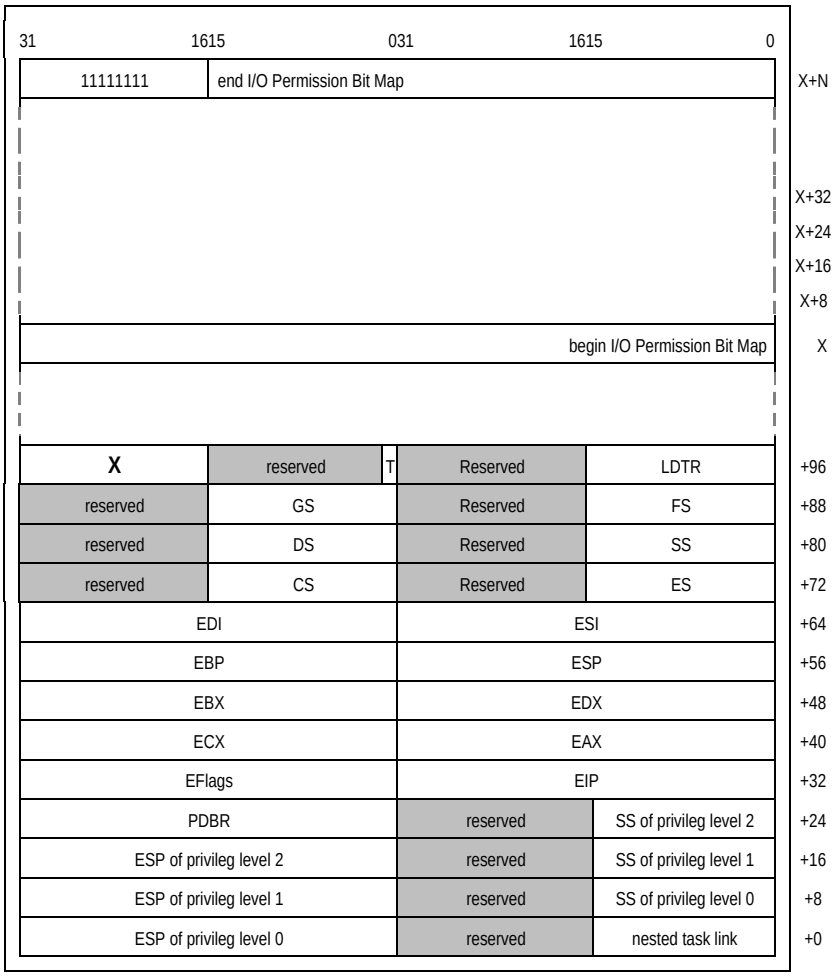


31 … 16	15 … 0		31 … 16	15 … 0	
I/O map base address	reserved	T	Reserved	LDTR	+96
reserved	GS		Reserved	FS	+88
reserved	DS		Reserved	SS	+80
reserved	CS		Reserved	ES	+72
EDI			ESI		+64
EBP			ESP		+56
EBX			EDX		+48
ECX			EAX		+40
EFlags			EIP		+32
PDBR			reserved	SS of privileg level 2	+24
ESP of privileg level 2			reserved	SS of privileg level 1	+16
ESP of privileg level 1			reserved	SS of privileg level 0	+8
ESP of privileg level 0			reserved	nested task link	+0

Die Tabelle mit dem schönen Namen I/O Permission Bit Map steht allerdings nicht irgendwo im Speicher, denn dann dürfte sie nicht nur eine 16-Bit-Adresse haben – oder müßte über Selektoren angesprochen werden. Dann aber würde wieder das Schutzkonzept für Segmente greifen, was in diesem Fall Unsinn wäre! Denn warum sollte die Information, ob auf etwas zugegriffen werden darf, schützenswert sein? Nein, diese Information muß frei für jeden verfügbar sein, der das wissen will. Also ist die Adresse lediglich ein Offset zu einer bekannten Basisadresse. Die ist die Basisadresse des TSS selbst. Das heißt: die I/O-Map schließt sich direkt an die TSS an, besser: ist Teil der (erweiterten) TSS, und zwar an Offset »I/O Map Base Address«. Das bedeutet, daß die Segmentgrenze des TSS, also das Feld *Limit* im Descriptor, genau diese Erweiterung berücksichtigen muß. (Also im Klartext: Wenn die TSS eine I/O Permission Bit Map hat, so zeigt das Feld *Limit* im Deskriptor für die TSS auf das Ende der I/O Permission Bit Map. Zeigt es dagegen auf das Feld I/O Map Base Address, gibt es keine I/O Permission Bit Map.)

Mit diesem Wissen können wir nun das TSS wie folgt darstellen:

32-Bit-Task-State-Segment



Das TSS wurde um die I/O Permission Bit Map erweitert. Zwischen dem Feld I/O Map Base Address, das im oberen Schaubild den Wert »X« hat, und dem Beginn der Bit Map können weitere Informationen stehen. Ab dem in I/O Map Base Address stehenden Offset beginnt ein Bitfeld, in dem jedes Bit einen Port repräsentiert. Abgeschlossen wird die Tabelle durch ein Byte, in dem alle Bits gesetzt sind. Zugriff auf einen Port wird nun nur in dem Falle erlaubt, daß das korrespondierende Bit in der Tabelle gelöscht ist. Übrigens: Wir haben bereits angesprochen, daß Ports auch eine bestimmte Größe haben können.

Das bedeutet, daß es für einen Port je nach seiner Größe mehrere Permission-Bits geben kann. So hat z.B. ein DWord-Port an Adresse $0100 insgesamt vier Permission-Bits: Bit $0100, $0101, $0102 und $0103. Zugriff erlaubt ist nur in dem Fall, in dem alle betroffenen Permission-Bits gelöscht sind. Ist auch nur ein einziges gesetzt, so wird der Zugriff nicht gestattet.

Das Bitfeld muß keine feste Größe haben. Falls nur die ersten 24 Ports individuell geregelt werden sollen, werden nur drei Byte ($3 \cdot 8 = 24$) erforderlich. Die restlichen Ports werden bei einer Prüfung dann so behandelt, als wären die sie repräsentierenden, nicht in der I/O Map vorhandenen Bits gesetzt, ein Zugriff also schlichtweg verboten. Es gibt nur drei Bedingungen an die Bit Map: Sie muß immer bei Port 0 beginnen und kontinuierlich bis zu dem Port gehen, der als letzter freigegeben werden soll. Auch wenn nur der letzte Port interessieren sollte! Denn die Nummer des freizugebenden Ports wird aus der Position im Bitfeld berechnet. Weiter: der Wert für die Basisadresse der Tabelle, der in das Feld I/O Map Base Address eingetragen wird, darf nicht größer als $DFFF sein! Schließlich muß die Tabelle mit einem $FF-Byte abgeschlossen werden.

Wie und wann erfolgt nun der Zugriffsschutz auf die Ports? Auf Ports kann nur mit wenigen Befehlen zugegriffen werden: IN/OUT, INS/OUTS und deren Abkömmlinge. Diese Befehle prüfen als erstes, ob der CPL kleiner oder gleich dem IOPL ist. Ist das nicht der Fall, sind alle Ports geschützt, ein Zugriff wird generell verweigert, und ein Zugriffsversuch durch eine #GP-Exception geahndet. Andernfalls prüfen sie, ob es eine I/O Permission Map im TSS gibt. Wenn nein, wird der Zugriff ebenfalls generell untersagt. Wenn aber doch, so werden alle zu einem Port gehörenden Bits dieser Tabelle geprüft. Ist auch nur eines der betrachteten Bits gesetzt, wird wiederum ein Zugriffsversuch mit einer #GP-Exception bestraft. Andernfalls erfolgt der Zugriff.

19.4.6 Weitere Schutzmechanismen

Es gibt noch weitere Mechanismen, die den Protected-Mode zu dem werden lassen, was er ist: ein geschützter Modus. Einige möchte ich im folgenden aufzählen:

- Privilegierte Befehle. Nicht alle Prozessorbefehle können in allen Situationen benutzt werden. So gibt es sogenannte privilegierte Befehle, die Sie nur dann einsetzen können, wenn Sie Module für die höchste Privilegstufe schreiben: für Betriebssystemmodule. Denn bei diesen Befehlen prüft der Prozessor, ob der CPL = 0 ist. Ist das nicht der Fall, wird eine #GP ausgelöst. Diese Befehle sind

Befehle, die in irgendeiner Weise in die Schutzkonzepte des Protected-Mode eingreifen könnten: LGDT, LLDT, LTR, LIDT, MOV mit den Control Registern oder den Debug-Registern als Operand, LMSW, CLTS, INVD, WBINVD, INVDPG, HLT, RDMSR, WRMSR, RDPMC, RDTSC. Für die letzten beiden Befehle gibt es eine Ausnahme: Falls das PCE- und das TSD-Flag im Kontrollregister 4 (CR4) gesetzt sind, können RDPMC und RDTSC bei allen CPL-Werten ausgelesen werden.

- Protected-Mode-Befehle. LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW und ARPL können nur im Protected-Mode ausgeführt werden. Im Real-Mode oder im virtuellen 8086-Modus aufgerufen, löst der Prozessor eine #DU (invalid opcode-)Exception aus.
- Zeigervalidierung. Generell werden Zeiger auf die gleiche Weise wie die eben besprochenen Zugriffe geprüft. So können Zeiger auf Datensegmente auch tatsächlich nur auf Daten-, nicht aber auf Code- oder Systemsegmente zeigen (Typ- und Privilegprüfung). Es wird geprüft, ob Zeiger Null sind oder ob ein lesender (Codesegmente) oder schreibender (read-only-Datensegmente) Zugriff auch erlaubt ist (Prüfung der Zugriffsrechte). Neben der Prüfung, ob Zeiger innerhalb der gültigen Grenzen bleiben, erfolgt auch eine Prüfung der Ausrichtung, ein sog. Alignment Check. So kann bei einem CPL = 3 (und nur dort!) durch Setzen des AM-Flags in CR0 und des AC-Flags in EFlags erreicht werden, daß der Prozessor Zeiger auf eine gültige Ausrichtung hin überprüft. Sollte dies nicht der Fall sein, wird eine #AC-Exception ausgelöst.

Teil 2

Arbeiten mit dem Assembler

20 Hallo, Welt!

Es ist schon fast so etwas wie Tradition geworden, in eine Programmiersprache mit einem kleinen Programm einzuführen, das sofort ein Aha-Erlebnis aufkommen läßt. Ein solches Programm zeigt seinen reibungslosen Ablauf dadurch an, daß es den Schriftzug »Hallo, Welt!« auf den Bildschirm zaubert. Auch wir wollen es so halten.

Starten Sie daher tpASM[7] oder einen anderen Editor, und öffnen Sie ein neues Fenster. Geben Sie dann folgende Zeilen ein, und achten Sie darauf, keinen Tippfehler zu machen.

```
DATA SEGMENT WORD 'DATA'
Msg DB 13,10,'Hallo, Welt!',7,13,10,'$'
DATA ENDS

CODE SEGMENT WORD 'CODE'
ASSUME CS:CODE, DS:DATA
Start:   mov   ax,DATA
         mov   ds,ax
         mov   dx,OFFSET Msg
         mov   ah,009h
         int   021h
         mov   ah,04Ch
         int   021h
CODE ENDS

END Start
```

Im folgenden werde ich mich darauf beschränken, die Anweisungen anzugeben, die tpASM benötigt, um das fertige Programm zu erzeugen.

Bevor wir nun den Quelltext etwas näher betrachten, bereiten wir uns zunächst ein Erfolgserlebnis! Gehen Sie zunächst in OPTIONS/SETTINGS... Kreuzen Sie unter PROGRAM EXECUTION den Eintrag PAUSE AFTER EXECUTION an. Dies können Sie entweder mit der Maus direkt oder via Tab und Space, die Cursortasten und Return. Aktivieren Sie dann einfach den Menüpunkt RUN/RUN. Es öffnet sich ein Fenster, in dem Sie angezeigt bekommen, wie der derzeitige Status ist.

---

[7] Wie Sie tpASM installieren und nutzen können, entnehmen Sie bitte der Datei README.TXT auf der beiliegenden Diskette.

Falls Sie keinen Tippfehler eingegeben haben, sollte die Assemblierung fehlerfrei erfolgen. Ist dies der Fall, wird das Programm automatisch gelinkt. Falls auch dies keinen Fehler erzeugt, so haben Sie ein lauffähiges Programm erzeugt, das nun aufgerufen wird. Sie sehen seine Bildschirmausgabe in der folgenden Abbildung.

Falls sich also der Schriftzug »Hallo, Welt!« mit einem kurzen Pieps auf dem Bildschirm hat darstellen lassen, so ist alles richtig verlaufen. Sie können dann durch einen Druck auf eine beliebige Taste in die Entwicklungsumgebung zurückgelangen! Falls das nicht funktioniert, tun Sie folgendes:

- Prüfen Sie, ob tpASM die richtigen Pfade für den Assembler und den *Linker* hat. Sie haben in diesem Fall von tpASM eine entsprechende Fehlermeldung erhalten, in der Ihnen mitgeteilt wird, daß der Assembler/Linker nicht gefunden werden konnte.

```
C:\>tpasm

tpASM  Version 2.1  Copyright (c) 1993 by Trutz Podschun


Hallo, Welt!
```

  Rufen Sie dann tpCONFIG auf, oder aktivieren Sie aus tpASM heraus den Menüpunkt OPTIONS/DIRECTORIES. Stellen Sie sicher, daß die eingegebenen Pfade vollständig sind und auch den Namen des jeweiligen Programms mit ».EXE«-Erweiterung beinhalten, also z.B. C:\TASM\TASM.EXE oder C:\LINKER\TLINK.EXE (man nennt dies unter DOS eine qualifizierte Pfadangabe).

- Beachten Sie bitte die Fehlermeldungen, die Ihnen tpASM anzeigt. Falls sich nämlich ein Tippfehler im Quelltext befindet, so kann ihn der Assembler gegebenenfalls nicht korrekt assemblieren. In diesem Falle sollte tpASM Ihnen dies in einem Meldungsfenster mitteilen und nach Bestätigung durch die Eingabe-Taste ein Fen-

ster öffnen, in dem die Fehlermeldungen des Assemblers sichtbar werden. Analoges gilt natürlich für den Linker, wenn der Assembler zwar ein korrektes OBJ-Modul erzeugen konnte, dies aber nicht durch den Linker in ein lauffähiges Programm gelinkt werden konnte.

- Sollte sich kein offensichtlicher Fehler finden lassen, so verwenden Sie das auf der beiliegenden CD-ROM mitgelieferte Programm HI_WORLD.ASM. Es ist der Quelltext, den Sie eingeben müssen. Falls tpASM HI_WORLD.ASM über den Menübefehl RUN/RUN korrekt ausführt, so haben Sie sich bei der Eingabe vertan. Vergleichen Sie in diesem Falle Ihren Quelltext mit HI_WORLD.ASM. Sie brauchen dabei alles, was hinter einem Semikolon steht, nicht abzutippen – dies ist nur Kommentar, der vom Assembler nicht berücksichtigt wird. Auf diese Weise sollten Sie den Fehler eliminieren können.
- Hilft auch dies nicht, so versuchen Sie zunächst, mit dem Menüpunkt ASSEMBLE/ASSEMBLE ein OBJ-Modul zu assemblieren. Es sollten hierbei keine Fehler auftreten. Falls doch, so stimmt etwas mit Ihrem Quelltext oder den Directory-Angaben des tpASM nicht. Sie müssen dann wie oben verfahren: Überprüfen Sie, ob tpASM die korrekten vollständigen Pfadangaben für den Assembler/Linker hat, und vergleichen Sie Ihren Quelltext mit HI_WORLD.ASM. Läßt sich kein Fehler ausmachen, sollten Sie sich spätestens jetzt fragen, ob Sie überhaupt einen Assembler/Linker auf der Platte installiert haben!

  Traten keine Fehler auf, so linken Sie das OBJ-Modul in einem zweiten Schritt über den Menüpunkt ASSEMBLE/LINK/LINK. Auch hier sollte nun alles klargehen.

Falls keine Fehler zu erkennen sind, sollte es keinen Grund geben, daß RUN/RUN kein Programm aufruft! Beenden Sie trotzdem tpASM. Suchen Sie in dem Verzeichnis, in dem sich die EXE-Dateien befinden sollten (dieses Verzeichnis haben Sie über tpCONFIG oder den Menüpunkt OPTIONS/DIRECTORIES eingegeben), nach Ihrer EXE-Datei. Sie muß existieren, wenn die beiden obigen Schritte keinen Fehler angezeigt haben und Sie den Linker nicht angewiesen haben, .COM-Dateien zu erzeugen! Rufen Sie nun einfach diese Datei aus DOS heraus wie jedes ausführbare Programm auf.

20.1 Die Segmente eines Assemblerprogramms

Beginnen wir nun mit der Erläuterung des Quelltextes. Beim ersten Anschauen sollten Ihnen mehrere Dinge auffallen:

- Einige der Worte im Quelltext bestehen aus Großbuchstaben, andere aus Kleinbuchstaben, wieder andere zeichnen sich durch die ganz normale Schreibweise aus.
- Es gibt sehr viele Begriffe, die Sie noch nicht kennen, weil sie bisher noch nicht erwähnt wurden! So z.B. »DB«, »ENDS« oder »SEGMENT«, um nur einige zu nennen.
- Einige der Ihnen noch nicht vorgestellten Begriffe sollten Ihnen dennoch zumindest bekannt vorkommen: »SEGMENT«, »CODE«, »DATA« z.B.
- Ein ganzer Block allerdings sollte Ihnen nach dem ersten Teil dieses Buches sehr bekannt vorkommen: alles das, was hinter »Start:« steht und klein geschrieben ist! Dies sind die Assemblerbefehle, die wir in Teil 1 ausführlich beschrieben haben.

Alle im Quelltext groß geschriebenen Begriffe sind sogenannte Assembler*anweisungen*. Bitte verwechseln Sie dies nicht mit den Assembler*befehlen* aus Teil 1. Diese Assembleranweisungen dienen dazu, den Assembler in seiner Funktion zu steuern. Mit ihnen werden dem Assembler wichtige Informationen gegeben, die er unbedingt zu seiner Arbeit benötigt, die er aber nicht, wie die Assemblerbefehle, in ausführbaren Code assembliert.

Fangen wir, um dies zu verstehen, ganz vorn im Quelltext an. Die erste Zeile heißt:

SEGMENT

```
DATA SEGMENT WORD 'DATA'
```

Diese Zeile sagt dem Assembler, daß im folgenden ein Segment erzeugt werden soll. Das erste Wort dieser Zeile, *DATA*, ist ein reserviertes Wort, das Sie in Assemblertexten nur dann verwenden dürfen, wenn Sie das Segment meinen, das Daten aufnehmen soll. Auch *SEGMENT* ist ein reserviertes Wort, das dem Assembler sagt, daß im folgenden ein neues Segment definiert wird. Mit *DATA SEGMENT* definieren Sie somit ein Datensegment.

Doch wozu? Schließlich wollen wir ja einfach nur den Text »Hallo, Welt!« auf dem Bildschirm ausgeben. Richtig! Wie die Hochsprachen unterscheidet auch der Assembler den reinen ausführbaren Code, der den Prozessor zur Aktivität bringt, von Daten, die dieser Code nutzt. Wenn wir also einen Text auf dem Bildschirm ausgeben wollen, so muß der Code, der dies tut, eben den Text als Datum vorfinden. Alle

Daten eines Programms aber werden, wie in Hochsprachen auch, in einem eigenen Bereich des Speichers, also einem eigenständigen Segment, gehalten – dem Datensegment.

Nachdem wir dieses Datensegment erzeugt haben, können wir die Daten angeben. Vorher aber wird dem Assembler noch mitgeteilt, daß das Segment auf jeden Fall an geraden Adressen beginnen oder »an Wortadressen ausgerichtet sein« soll, wie man sagt. Dies erfolgt über das reservierte Wort WORD direkt hinter der Segmentdefinition. Einen Namen hat das Segment auch: DATA, was nicht verwunderlich ist, da es ja das Datensegment DATA ist.

Halten wir fest, daß eine Segmentdefinition die allgemeine Form

Segmenttyp SEGMENT Ausrichtung Name

hat. Abgeschlossen wird die Definition eines Segments mit dem ebenfalls reservierten Wort *ENDS*, was man als *End of Segment* lesen sollte. Dies heißt, daß alles, was zwischen diesen beiden Zeilen steht, in das spezifizierte Segment gebracht wird.

ACHTUNG

ENDS darf nicht allein im Raum schweben, sondern muß auf die Angabe des Segmenttyps folgen! Ein Segment wird somit allgemein wie folgt abgeschlossen:

Segmenttyp ENDS

TIP

Vielleicht erkennen Sie jetzt, was es mit der unterschiedlichen Schreibweise auf sich hat. Dem Assembler ist nämlich eigentlich ganz egal, wie der Text geschrieben wird. Er unterscheidet, ganz analog zu DOS, nicht zwischen Groß- und Kleinschreibung. Dies können wir benutzen, um rein optisch die Assembler*anweisungen* von den Assembler*befehlen* zu unterscheiden – der Quelltext wird auf diese Weise lesbarer. Vereinbaren wir daher für den Rest des Buches, daß Assembleranweisungen immer groß, Assemblerbefehle immer klein geschrieben werden.

Zurück zum Datensegment! Mit

```
DATA SEGMENT WORD 'DATA'
Msg DB 13,10,'Hallo, Welt!',7,13,10,'$'
DATA ENDS
```

weisen wir den Assembler an, ein Datensegment zu erzeugen, das Daten enthält. Diese Datendefinition erfolgt ganz offensichtlich in der zweiten Zeile. Doch was dort steht, ist ungewöhnlich!

DB

Zunächst fällt wieder eine Assembleranweisung auf: *DB*. DB steht für *Define Byte* und sagt dem Assembler, daß nun Bytes folgen, die als Daten zu interpretieren sind. Dann folgen die Bytes: 13, 10 ... doch

dann? *'Hallo, Welt!'* – der Text, den wir ausgeben wollen. Aber nicht als Bytes, sondern als String, also tatsächlich als Textzeile! Das ist ungewöhnlich! Noch ungewöhnlicher ist: An den String schließen sich nochmals Datenbytes an, gefolgt von einem einzelnen Buchstaben, dem Dollarzeichen!

Wenn wir zunächst unberücksichtigt lassen, was es mit 10,13 und $ auf sich hat, läßt sich folgendes feststellen:

HINWEIS Daten können durch Angabe ihres Zahlenwertes oder als ASCII-Zeichen eingegeben werden. ASCII-Zeichen werden vom Assembler in Zahlenwerte übersetzt und müssen dazu in einzelnen Anführungszeichen (') stehen, mehrere Daten werden in einer Zeile durch Kommata getrennt. Mehrere aufeinanderfolgende ASCII-Zeichen müssen nicht einzeln angegeben werden, sondern dürfen als String definiert werden. Die Anweisung

```
DB 10,13, 'Hallo, Welt!', 7, 13, 10, '$'
```

ist also identisch mit der Anweisung

```
DB 10, 13, 'H', 'a', 'l', 'l', 'o', ',', ' ', 'W', 'e', 'l', 't',
'!', 7, 13, 10, '$'
```

und diese wiederum mit:

```
DB 10,13,72,97,108,108,111,44,32,87,101,108,116,33,7,13,10,36
```

Das ist es, was der Assembler in allen drei Fällen erzeugt.

Wie lange dürfen Datenbytes bzw. Strings oder Buchstaben folgen? Dies führt zu einer allgemeingültigen Einschränkung:

ACHTUNG Anweisungen und Befehle dürfen in Assembler eine Zeile nicht überschreiten!

Falls also eine Zeile nicht ausreicht, um alle Daten darzustellen, so muß die Dateneingabe in der nächsten Zeile fortgesetzt werden, der dann jedoch eine neue Anweisung voranzugehen hat! Folgender Quelltext ist also *verboten*:

```
DB 'Dies ist ein Beispielsstring, der so lang ist, daß er sich nicht
in einer Zeile darstellen läßt!'
```

Hier muß nach `er` der String beendet werden und eine neue DB-Anweisung in der nächsten Zeile folgen:

```
DB 'Dies ist ein Beispielstring, der so lang ist, daß er'
DB 'sich nicht in einer Zeile darstellen läßt!'
```

HINWEIS Halten wir also fest: In jeder Zeile des Quelltextes darf nur jeweils eine Assembleranweisung bzw. ein Assemblerbefehl stehen!

Dies ist jedoch kein großes Problem, da sich Assembleranweisungen nicht mehr im Assemblat wiederfinden, weil sie ja nur Anweisungen sind, wie der Assembler was zu interpretieren hat. Das wiederum heißt, daß auf das Leerzeichen hinter `er` in der ersten Datenzeile unmittelbar das `s` der zweiten Datenzeile folgt.

Labels

Doch wir sind mit unserem Datensegment noch nicht ganz fertig! Denn vor der DB-Anweisung, nach der die Daten angeführt wurden, steht noch ein Wort, `Msg`. Dieses Wort, in normaler Weise mit Groß- und Kleinbuchstaben geschrieben, ist ein sogenanntes *Label*. Labels dürfen, müssen aber nicht vor Datenanweisungen stehen. `Test DB 7` ordnet dem Datenbyte 7 das Label `Test` zu, `Msg DB 10,13,'Hallo, Welt!',7,10,13,'$'` der Folge von Bytes angefangen bei 10 bis '$' das Label `Msg`.

Doch wozu das Ganze? Dies ist eine sehr nützliche Eigenschaft des Assemblers, auf die Daten zurückgreifen zu können. Stellen Sie sich folgende Datendefinition vor:

```
DATA SEGMENT WORD 'DATA'
DB 'Dies ist die erste Zeile des Datensegments'
DB 'Und dies die zweite Zeile'
DB 'Ihr schließt sich die dritte Zeile an, '
DB 'gefolgt von der vierten.'
DB 'Sowie eine fünfte, in der das Byte ',135,' steht.'
DATA ENDS
```

Stellen Sie sich nun vor, das Programm müßte, aus welchen Gründen auch immer, ausgerechnet auf die Zahl 135 in der fünften Zeile zurückgreifen. Was also müßten *Sie* tun, um z.B. einem MOV-Befehl, mit dem Sie diese Zahl lesen können, die Stelle kundtzutun, an der sie steht? Abzählen, an welchem Byte vom Beginn des Datensegments sie steht. Dies wäre bei umfangreichen Daten sehr mühsam und fehleranfällig.

Also verpaßt man der Zahl 135 einfach ein Label:

```
DATA SEGMENT WORD 'DATA'
         DB 'Dies ist die erste Zeile des Datensegments'
         DB 'Und dies die zweite Zeile'
         DB 'Ihr schließt sich die dritte Zeile an, '
         DB 'gefolgt von der vierten.'
         DB 'Sowie eine fünfte, in der das isolierte Byte'
Wichtig  DB 135
         DB ' steht.'
DATA ENDS
```

Sie ist auf diese Weise ganz einfach unter dem Label ansprechbar. So würde z.B.

```
mov      ah,Wichtig
```

genau die Zahl 135 finden und in das AH-Register kopieren.

CODE

Ganz analog zum Datensegment wird ein Segment erzeugt, in dem ausführbarer Code steht:

```
CODE SEGMENT WORD 'CODE'
ASSUME CS:CODE, DS:DATA
Start:   mov   ax,DATA
         mov   ds,ax
         mov   dx,OFFSET Msg
         mov   ah,009h
         int   021h
         mov   ah,04Ch
         int   021h
CODE ENDS
```

Mit

```
CODE SEGMENT WORD 'CODE'
```

teilen wir dem Assembler mit, daß er ein Segment vom Typ CODE erzeugen soll. Auch dieses Segment soll auf Worte ausgerichtet werden und hat den Namen CODE. Abgeschlossen wird auch dieses Segment analog zum Datensegment:

```
CODE ENDS
```

Bevor wir uns im nächsten Abschnitt mit dem ausführbaren Code beschäftigen, sollte zuvor noch eine Assembleranweisung besprochen werden:

ASSUME

```
ASSUME CS:CODE, DS:DATA
```

Wie wir im ersten Teil des Buches bei der Besprechung der Register des 8086 gesehen haben, gibt es Segmentregister auf dem Chip. Diese Segmentregister, enthalten die Adressen von Segmenten, genauer gesagt, die Adressen der Segmentgrenze eines Segments.

Es gibt mit CS ein Segmentregister, das die Segmentgrenze des Segments enthält, in dem der ausführbare Code eines Programms steht. DS enthält die analoge Adresse für ein Datensegment. Eben haben wir gesehen, wie mit dem Assembler Segmente definiert werden können. Und wir haben gesehen, daß wir verschiedene Segmente unabhängig voneinander definieren können: ein Codesegment und ein Datenseg-

ment. Wer verbietet uns eigentlich, zusätzliche Segmente zu definieren? Und wer befiehlt uns, das Codesegment CODE und das Datensegment DATA zu nennen?

Antwort: Niemand! Daß das Codesegment CODE und das Datensegment DATA heißen, ist reine Tradition. Definieren wir z.B. das Datensegment über

```
SAETZLE SEGMENT WORD 'NEE_WIE_ISSES_SCHOEN'
```

und das Codesegment mit

```
MACHMAL SEGMENT WORD 'ISSES_WAHR'
```

so können wir sicherlich auch ein lauffähiges Programm erzeugen (Assemblieren und linken Sie den Quelltext HI_FUN.ASM auf der beiliegenden CD-ROM). Wenn dem aber so ist, woher soll der Assembler wissen, in welchem Segment der Code und in welchem die Daten stecken? Die Adresse welches Segments muß in CS geladen werden, damit das Programm ablaufen kann?

Genau das müssen wir dem Assembler mitteilen! Über ASSUME sagen wir ihm, daß sich im für das Codesegment zuständigen Segmentregister CS die Segmentnummer des Codesegments befindet, im einen Fall mit dem Typ CODE (ASSUME CS:CODE) oder im anderen Fall mit MACHMAL (ASSUME CS:MACHMAL). Ebenso können wir das Datensegment spezifizieren: ASSUME DS:DATA definiert als Inhalt des für das Datensegment zuständigen Segmentregisters DS die Segmentgrenze von DATA, bei ASSUME DS:SAETZLE die des Segments SAETZLE! Beide Anweisungen lassen sich natürlich auch kombinieren, was dann zu ASSUME CS:CODE, DS:DATA bzw. ASSUME CS:MACHMAL, DS:SAETZLE führt.

ACHTUNG

Mit der ASSUME-Anweisung verändern wir nicht den Inhalt der Register CS und DS. Dies muß noch gesondert erfolgen. ASSUME teilt dem Assembler lediglich mit, daß er zum Zeitpunkt der Assemblierung *annehmen* (nichts anderes heißt ja *assume*!) soll, daß in den angegebenen Registern die Segmentnummern der ebenfalls angegebenen Segmente liegen. Wer sie wann und wie dorthin bringt, ist damit noch nicht festgelegt.

TIP

Das heißt aber auch, daß uns niemand daran hindert, ein anderes Segment als das eigentliche Datensegment als Datensegment zu verwenden, etwa das Codesegment! So könnte dem Assembler mit ASSUME CS:CODE, DS:CODE mitgeteilt werden, daß sowohl der ausführbare Code als auch etwaige Daten im gleichen Segment liegen! Genau das werden wir etwas später auch einmal verwenden!

HINWEIS Wir werden dennoch als Segmenttypen weiterhin CODE und DATA verwenden! Dies hat mehrere Gründe. Zum einen läßt sich daraus schon deutlich erkennen, was das Segment enthält. Ferner werden wir mit Assembler nur sehr kleine lauffähige Programme erzeugen, hauptsächlich soll der Assembler ja zur Programmierung von Routinen dienen, die in Hochsprachen eingesetzt werden sollen. Diese Hochsprachen verwenden aber ganz bestimmte »Speichermodelle«, in denen eben die Segmente z.B. CODE und DATA heißen. Wir kommen noch darauf zurück.

Bleibt eigentlich nur noch die Frage, warum wir dem Assembler Code- und Datensegment (und eventuell auch ein Extra- und/oder Stacksegment) nennen müssen. Denn der Assembler »übersetzt« ja lediglich den Quelltext in maschinenlesbaren Opcode! Und wenn CS und DS (und gegebenenfalls auch ES und SS, ja vielleicht auch FS und GS ab dem 80386) sowieso noch gesondert mit der richtigen Adresse belegt werden müssen, ist der Sinn von ASSUME nur schwer zu verstehen.

Aber genau das Übersetzen ist der Punkt. Wir haben eben gesehen, daß Labels z.B. im Datensegment eine wertvolle Hilfe sind, um unnötiges Abzählen von Bytes im Quellcode zu verhindern, wenn wir einzelne Bytes ansprechen wollen. Denn einem MOV-Befehl z.B. muß eine Adresse als Operand übergeben werden, falls ein Speicheroperand verwendet werden soll. Die Angabe des Labelnamens als Operand nimmt es uns ab, die dazugehörige Adresse berechnen und als Operand eingeben zu müssen.

Das heißt jedoch, daß das Label in eine Adresse umgerechnet werden muß! Genau diese Arbeit leistet der Assembler. Wir wissen nun aber, daß Adressen immer aus einem Segment und einem Offset bestehen. Nun ist es für den Assembler ein Leichtes, den Offset zu bestimmen – er zählt einfach die Bytes bis zu dem Label, beginnend mit der Definition des Segments. Doch der Segmentanteil?

Normalerweise versucht man, mit einem MOV-Befehl Daten zu kopieren. Daten aber gehören in ein eigens dafür angelegtes Segment, das Datensegment. Also versucht der Assembler nun ganz logisch, als Segmentanteil der zu berechnenden Adresse die Adresse dieses Segments zu verwenden. Doch genau dazu muß der Assembler wissen, welches der eben definierten Segmente nun das Datensegment ist! Analog gilt für alle anderen Segmente: Falls ein Sprungziel berechnet werden muß, ist der Offset der Adresse einfach durch Abzählen der Bytes vom Beginn des Segments an zu berechnen. Auch hier muß aber der Segmentanteil der Adresse dadurch bestimmt werden, daß die Adresse des dazugehörigen Codesegments verwendet wird. Und dieses muß dem Assembler bekanntgemacht werden.

Zur Veranschaulichung folgen ein Assemblerquellcode und ein Listing, das der Assembler aus dem Quellcode macht. Zunächst sehen Sie das, was wir als Quellcode schreiben:

```
DATA SEGMENT PARA PUBLIC
Var1 dw ?
DATA ENDS

EXTRA SEGMENT PARA PUBLIC
Var2 dw ?
EXTRA ENDS

CODE SEGMENT WORD PUBLIC
Var3 dw ?

ASSUME CS:CODE

start:   mov   ax,04711h
         mov   Var1,ax
         mov   ax,00815h
         mov   Var2,ax
         mov   ax,01234h
         mov   Var3,ax
         ret
CODE ENDS

END start
```

Der Sinn dieses Programms dürfte klar sein! Wir möchten drei Variablen mit irgendwelchen Werten belegen. Beachten wir dabei zunächst nicht, daß die Art, wie wir dies tun, sehr umständlich ist, da man Konstanten auch direkt in Variablen eingeben kann, ohne den Umweg über das AX-Register gehen zu müssen. Ja, man würde besser solche Startwerte sogar schon als Vorgaben programmieren. Aber das Programm soll ja den Sinn von ASSUME verdeutlichen.

Wir haben drei Segmente definiert: EXTRA, DATA und CODE. In allen drei Segmenten sind Variablen definiert: *Var1* in DATA, *Var2* in EXTRA und *Var3* in CODE. Alle drei Variablen sind Wortvariablen und besitzen keinen Startwert, was durch das »?« signalisiert wird. Wir müssen auf jeden Fall ASSUME CS:CODE angeben, damit der Assembler weiß, in welchem Segment sich der ausführbare Code befinden soll. Falls wir nun den Quelltext assemblieren, so stellen wir den Assembler vor ein Problem: Er muß aus den Informationen, die ihm zur Verfügung stehen, vollständige Adressen berechnen, die der Prozessor dann dazu benutzen kann, die Variablen auch tatsächlich

adressieren zu können. Der Assembler weiß, daß *Var1* in DATA steht und *Var2* in Extra! Aber wie kann er dem Prozessor kundtun, wo dieser die Segmentadressen von DATA und EXTRA findet? Auf die Weise wie oben nicht!

Deshalb erzeugt er auch eine Fehlermeldung, falls Sie den Code oben assemblieren wollten. Denn – selbst *Sie* wissen zum jetzigen Zeitpunkt nicht, in welchem der Segmentregister die Segmentadressen von DATA und EXTRA stehen. Also müssen wir diese zunächst laden:

```
DATA SEGMENT PARA PUBLIC
var1 dw ?
DATA ENDS

EXTRA SEGMENT PARA PUBLIC
var2 dw ?
EXTRA ENDS

CODE SEGMENT WORD PUBLIC
var3 dw ?

ASSUME CS:CODE

start:    mov   ax,SEG DATA
          mov   ds,ax
          mov   ax,SEG EXTRA
          mov   ES,AX
          mov   ax,04711h
          mov   Var1,ax
          mov   ax,00815h
          mov   Var2,ax
          mov   ax,01234h
          mov   Var3,ax
          ret
CODE ENDS

END start
```

Durch die vier neu hinzugekommenen Zeilen laden wir in die Segmentregister DS und ES die Segmentadressen des Daten- und Extrasegments. Wir müssen hier tatsächlich den Umweg über AX gehen, da sich die Segmentregister nicht direkt mit Konstanten belegen lassen. SEG ist eine Assembleranweisung, die den Assembler veranlaßt, die Segmentadresse des betreffenden Segments an dieser Stelle zur Verfügung zu stellen.

Doch damit haben wir das Problem immer noch nicht gelöst! Denn wir könnten ja auch in DS die Adresse des Segments EXTRA eintragen und in ES die von DATA. Dann haben wir zwar immer noch die gleichen Adressen, aber eben in unterschiedlichen Segmentregistern. Also ergänzen wir die ASSUME-Anweisung entsprechend:

```
DATA SEGMENT PARA PUBLIC
Var1 dw ?
DATA ENDS

EXTRA SEGMENT PARA PUBLIC
Var2 dw ?
EXTRA ENDS

CODE SEGMENT WORD PUBLIC
Var3 dw ?

ASSUME CS:CODE, DS:DATA, ES:EXTRA

start:   mov   ax,SEG DATA
         mov   ds,ax
         mov   ax,SEG EXTRA
         mov   ES,AX
         mov   ax,04711h
         mov   Var1,ax
         mov   ax,00815h
         mov   Var2,ax
         mov   ax,01234h
         mov   Var3,ax
         ret
CODE ENDS

END start
```

Nun verfügt der Assembler über alle notwendigen Informationen, die der Prozessor braucht. Er erstellt aus dem Quellcode folgendes Assemblat, das man sich mit dem Debugger anschauen kann:

```
MOV      AX,0BE9
MOV      DS,AX
MOV      AX,0BF9
MOV      ES,AX
MOV      AX,4711
MOV      [0000],AX
MOV      AX,0815
MOV      ES:[0000],AX
```

```
MOV       AX,1234
MOV       CS:[0000],AX
RET
```

Wenn Sie dieses Disassemblat mit dem Quellcode oben vergleichen, so sollten Ihnen drei Dinge auffallen:

- Der Assembler hat bei diesem Assemblierlauf die Anweisung SEG DATA mit 0BE9 übersetzt, SEG EXTRA mit 0BF9, ganz wie gewünscht. Sie sehen also, daß Sie sich als Programmierer um solche Berechnungen nicht kümmern müssen!
- Der Assembler erzeugt vollständige Adressierungen. So übersetzt er die Anweisung *mov var2,ax* in MOV ES:[0000], AX. Genau hierzu diente die ASSUME-Anweisung. Weil der Assembler weiß, daß die Adresse des Extrasegments in ES steht (durch ASSUME ES:EXTRA), sagt er dem Prozessor, daß dieser an die Stelle ES:[0000] den Wert aus AX einzutragen hat (bitte erinnern Sie sich, daß eine gültige Adresse immer aus einem Segmentanteil und einem Offset besteht, die in Assemblerschreibweise durch Segment:Offset dargestellt werden). Ebenso geschieht es mit *Var3*! Weil (über ASSUME CS:CODE) der Assembler weiß, daß sich in CS die Adresse des Codesegments befindet, kann er den Befehl *mov var3, ax* in MOV CS:[0000], AX übersetzen, was den Prozessor dann veranlaßt, den entsprechenden Wert in der Variablen *Var3* im Codesegment zu speichern.
- Das eben Gesagte scheint für die dritte Variable nicht zu gelten: Hier übersetzt der Assembler den Befehl *mov var1, ax* in MOV [0000], AX, also ohne die Segmentadresse. Der Prozessor verwendet bei bestimmten Befehlen nämlich bestimmte Standardwerte. Bei den MOV-Befehlen geht er davon aus, daß die Daten in das jeweilige Segment gespeichert werden, dessen Segmentadresse in DS verzeichnet ist. Daher benötigt er dann, wenn auf eine Variable in einem Segment mit Segmentadresse in DS zugegriffen werden soll, diese Information nicht – sie ist überflüssig! Das heißt also, daß der Assembler tatsächlich nur dann, wenn auf Segmente zugegriffen werden soll, die nicht die zu dem Befehl gehörigen Standardeinstellungen verwenden, vollständige Adressenangaben erzeugt.

Versuchen Sie einmal herauszufinden, wie sich dieses Disassemblat ändert, falls in der ASSUME-Anweisung folgendes steht:

```
ASSUME CS:CODE, DS:EXTRA, ES:DATA
```

Fassen wir also noch einmal zusammen. Mit den Assembleranweisungen SEGMENT und ENDS können wir Segmente erzeugen, in die wir Daten und/oder ausführbaren Code packen. Diese Segmente müssen wir jedoch beim Assembler mittels ASSUME quasi »anmelden«, damit dieser

sie als genau das erkennt, was sie darstellen und eventuell notwendige Berechnungen von Adressen von Sprungmarken oder Daten auf das richtige Segment bezieht.

20.2 Das erste Programm

Nachdem wir nun die Verwaltungsaufgaben beschrieben haben, können wir uns an das eigentliche Programm, den ausführbaren Code machen, um ihn zu untersuchen.

```
Start:   mov   ax,DATA
         mov   ds,ax
         mov   dx,OFFSET Msg
         mov   ah,009h
         int   021h
         mov   ah,04Ch
         int   021h
```

Hier also ist der ausführbare Code. MOV und INT kennen wir schon aus Teil 1 als Assemblerbefehle. Auch AH, DS und DX erkennen wir als Bezeichnungen von Registern des 8086. Unbekannt ist eigentlich nur `Start`, gefolgt von einem Doppelpunkt. Aber auch dies ist nichts Geheimnisvolles: Genau so, wie man vor die Assemblerdirektive DB den Namen eines Labels setzen kann, kann man es vor Assemblerbefehlen. Um dieses Label dann auch als Label erkennbar zu machen, läßt man dem Namen einen Doppelpunkt folgen. Mit `Start:` haben wir also ein Label, das auf den ersten ausführbaren Befehl im Codesegment zeigt: *mov ax, data*.

Da haben wir auch wieder unser reserviertes Wort DATA. Als Operand in einem Assemblerbefehl wie MOV übergeben, gibt dieses Wort die Adresse des Segments DATA zurück. *mov ax, data* lädt also die Segmentadresse des Datensegments in das Register AX. Von hier aus wird es mit dem nächsten Befehl in das *Datensegmentregister* DS kopiert: *mov ds, ax*. Nach dem zweiten Befehl also haben wir das Datensegmentregister mit der richtigen Adresse geladen.

Hier also ist der Punkt, an dem das DS-Register auf den richtigen Wert gesetzt wird, nicht etwa durch die ASSUME-Anweisung. Doch in Verbindung mit der ASSUME-Anweisung »weiß« der Assembler jetzt, in welchem Segment die Daten sind, und der Prozessor weiß es durch den korrekt gesetzten Inhalt des DS-Registers.

Nun folgt ein weiterer Ladebefehl, *mov dx, OFFSET Msg*. OFFSET ist im Quelltext auch wieder groß geschrieben und nach unserer Vereinbarung heißt das, daß es sich um eine Assembleranweisung, also kei-

nen Befehl handelt. Assembleranweisungen dienen zur Steuerung des Assemblers bei seiner Arbeit. Was also bewirkt OFFSET? Genau das, was Sie nun vermuten: OFFSET übergibt dem Befehl den Offsetanteil einer Adresse, in diesem Fall des Labels *Msg*. Das heißt, daß nach dieser Befehlszeile in DX der Offsetanteil des Bytes steht, auf das das Label *Msg* im Datensegment zeigt (denn wir haben dem Assembler durch ASSUME DS:DATA mitgeteilt, daß er das Segment DATA als Datensegment benutzen soll). Die vollständige Adresse dieses Bytes, also sein Segment- und Offsetanteil, steht also nach dieser Zeile in der Registerkombination DS:DX.

Im nächsten Schritt wird einfach die Konstante 9 in das AH-Register geschrieben, wozu auch der MOV-Befehl verwendet wird. Bei dieser Zeile ist eigentlich nur ungewöhnlich, daß der Wert 9 als 009h geschrieben wird. Aber das hat seinen guten Grund!

ACHTUNG Wenn das erste Zeichen einer Zeichenfolge ein Buchstabe ist, so nimmt der Assembler an, daß es sich um eine Anweisung, einen Befehl oder ein Label handelt (weshalb alle diese Elemente auch mit einem Buchstaben anfangen müssen). Falls wir nun aber Hexadezimalzahlen eingeben wollen, so führt das in all den Fällen zu Schwierigkeiten, in denen das führende Hexadezimalzeichen mit einem Buchstaben beginnt, also z.B. bei den Zahlen 10 bis 15, die ja hexadezimal als $A bis $F geschrieben werden. Um also dem Assembler kundtzutun, daß das Zeichen A in diesem Fall eine Konstante ist, muß es als Ziffer bezeichnet werden, was wir ganz einfach durch eine führnde 0 erreichen.

TIP Gewöhnen Sie sich an, *jede* Hexadezimalzahl im Quelltext mit einer führenden 0 anzugeben, selbst wenn es z.B. bei 9 oder 4711 oder 815 nicht notwendig wäre! Sie ersparen sich dadurch unliebsame Fehlermeldungen beim Assemblieren, weil der Assembler nicht erkennen kann, daß die Hexadezimalzahl, die Sie als Konstante eingegeben haben und die mit einem Buchstaben beginnt, eben tatsächlich eine Konstante und kein Label ist.

TIP Die führende 0 im obigen Beispiel ist damit erklärt, nicht aber die zweite, sich anschließende. Warum nicht einfach 09h? Dies resultiert daraus, daß Bytes, die hier als Konstanten übergeben werden, maximal zwei Ziffern umfassen können, nämlich von 0 bis $FF.

Es erhöht nun die Lesbarkeit eines Quelltextes, wie mir scheint, ungemein, wenn Sie, unabhängig von der führenden 0, jedes Byte daher mit zwei Ziffern darstellen, jedes Wort mit vier usw. Sie können dann allein schon optisch sehen, was für eine Konstante Sie verwenden und daß die Angabe *mov ah,00815h* zu einer Fehlermeldung des Assemblers führen wird!

INT

Was nun folgt, ist ein INT-Befehl. Dieser Befehl unterbricht die Ausführung des Programms. Der Prozessor sichert den Inhalt des Flagregisters und verzweigt an eine Stelle im RAM, an der die sogenannte Behandlungsroutine des aktuellen Interrupts steht. Diese Zusammenhänge werden im Anhang erläutert. Wenn Sie nachvollziehen möchten, was bei einem Interrupt passiert, so lesen Sie nun auf Seite 905 weiter!

HINWEIS

Wenn Sie den ersten Teil des Buches aufmerksam durchgelesen haben und sich noch an den Abschnitt über die Speicherbefehle erinnern, werden Sie sich vielleicht gewundert haben, daß wir zum Laden von DS:DX den augenscheinlich umständlichen Weg über *mov ax,DATA; mov ds, ax; mov dx,OFFSET Msg* gegangen sind. Man könnte doch DX mit dem Befehl *lea dx, Msg* laden? Richtig! Mit LEA wird ja der Offsetanteil eines Labels geladen. Daher ist *lea dx, Msg* das gleiche wie *mov dx, OFFSET Msg*. Wie gleich diese beiden Befehle tatsächlich sind, erkennt man daran, daß der Assembler die Zeile *lea dx, Msg* in *mov dx, OFFSET Msg* übersetzt. Das bedeutet also, daß LEA eigentlich kein Prozessorbefehl ist, sondern eine Assembleranweisung, die dieser mit anderen Prozessorbefehlen codiert.

TIP

Da tatsächlich die Befehle *LEA reg, label* und *MOV reg, OFFSET label* identisch sind, können sie wahlweise verwendet werden. Ich empfehle Ihnen jedoch, vor allem in der Eingewöhnungszeit, die MOV-Version zu verwenden. Wie ich glaube, zeigt sie deutlicher als der LEA-Befehl, was eigentlich abläuft, und Sie üben damit auch den Umgang mit der gewöhnungsbedürftigen Speichersegmentierung.

20.3 Eine nicht ganz unwichtige Assembleranweisung

```
END Start
```

END

END als Assembleranweisung teilt dem Assembler mit, daß hier der Assemblerquelltext zu Ende ist. Jede Zeile, die nach einem END kommt, ignoriert der Assembler vollständig.

HINWEIS

Die Angabe von END ist zwingend erforderlich. Ohne ein END erzeugt der Assembler eine Fehlermeldung, die uns sagt, daß der Quelltext für den Assembler unerwartet zu Ende ist.

Das wäre eigentlich auch schon alles, wenn hinter END nicht noch ein Wort stünde, das uns bekannt vorkommt: *Start*. *Start* haben wir als Label im Quellcode kennengelernt, das auf den ersten ausführbaren Befehl im Codesegment zeigt. Bisher haben wir von diesem Label keine Verwendung gemacht. Im vorletzten Abschnitt haben wir gesagt,

daß CS und DS, also die Segmentregister des Prozessors, nicht durch die ASSUME-Anweisung geladen werden, und im letzten Abschnitt haben wir das DS-Register selbst korrekt belegt, nicht aber das CS-Register. Warum eigentlich nicht?

Ganz einfach: Um dem Prozessor eine Adresse mitzuteilen, an der der ausführbare Code beginnt, müßten wir neben CS, das die Adresse des Codesegments beinhaltet, auch IP mit dem Offset des ersten ausführbaren Befehls laden. Das aber können wir nicht, weil der Assembler jeden direkten Zugriff auf IP, etwa *mov ip, ax* verweigert. Und das ist auch gut so!

Könnten wir nämlich direkt auf IP zugreifen, so würde das bewirken, daß der Prozessor sofort mit der Abarbeitung der Befehle beginnt, die an dem eben eingetragenen Offset stehen. Das wollen wir aber gar nicht, denn dies hieße ja, daß der Assembler während seiner Tätigkeit ein noch nicht vollständig übersetztes Programm ausführen läßt.

Der Assembler soll lediglich den für uns Menschen lesbaren Quelltext in Opcodes übersetzen! Das Programm selbst wollen wir zu einem Zeitpunkt aufrufen, den wir bestimmen – und nicht aus dem Assembler heraus, sondern wohldefiniert aus DOS. Das aber heißt wiederum, daß die Registerkombination CS:IP von DOS geladen werden muß – und zwar zu einem Zeitpunkt, an dem wir das Programm aufrufen.

ACHTUNG Die Registerkombination CS:IP ist für uns tabu! Ganz abgesehen davon, daß wir, zumindest am Offsetanteil IP, nichts daran verändern können, würden wir uns schwere Probleme schaffen, wenn wir es könnten.

Dennoch muß DOS wissen, welchen Wert es in die Register einzutragen hat. Daher muß der Assembler dem Linker die Information übergeben, welche Werte DOS beim Programmaufruf in CS:IP einzutragen hat. Der Wert für CS ist klar: Durch *ASSUME CS:CODE* teilten wir ihm mit, daß das Segment CODE das Codesegment ist und somit seine Adresse der neue Inhalt von CS. *Start* aber ist nach unserem Willen der Einsprungspunkt innerhalb des Segments, weshalb dieses Label den Offsetanteil, den DOS in IP einzutragen hat, widerspiegelt. Fehlt also nur noch, daß wir dem Assembler genau dieses kundtun: durch die Angabe des Einsprungpunkts als »Operand« der Anweisung END.

Der Assembler kennt nun mit *ASSUME CS:CODE* und *END Start* den von uns gewünschten Anfang des Programms und übermittelt diese Information jedem, der sie braucht.

20.4 Nachtrag

An dieser Stelle sei das Ergebnis der Aufgabe von Seite 262 dargestellt:

```
MOV       AX,0BE9
MOV       ES,AX
MOV       AX,0BF9
MOV       DS,AX
MOV       AX,4711
MOV       ES:[0000],AX
MOV       AX,0815
MOV       [0000],AX
MOV       AX,1234
MOV       CS:[0000],AX
RET
```

Ich bin mir fast sicher, daß Sie sich zumindest in einem Punkt geirrt haben: Mit großer Wahrscheinlichkeit haben Sie nicht die beiden Zeilen berücksichtigt, mit der die Segmentregister geladen werden. So dürften Zeile 2 und 4 des Listings oben nicht mit dem übereinstimmen, was Sie auf dem Papier stehen haben.

ACHTUNG

Achten Sie immer darauf, daß in der ASSUME-Anweisung tatsächlich die Segmente den Segmentregistern zugeordnet werden, deren Adressen Sie dann im Codeteil auch wirklich in die korrespondierenden Register laden.

Falls ich nämlich mit meiner Vermutung recht habe, passiert sonst Fatales. Schauen wir uns dies einmal an:

```
EXTRA SEGMENT PARA PUBLIC
Var1 dw ?
Var2 dw ?
EXTRA ENDS

DATA SEGMENT PARA PUBLIC
Var3dw ?
DATA ENDS

CODE SEGMENT WORD PUBLIC
ASSUME CS:CODE, DS:EXTRA, ES:DATA

start:    mov    ax,SEG DATA
          mov    ds,ax
          mov    ax,SEG EXTRA
          mov    ES,AX
          mov    ax,04711h
```

```
          mov   Var1,ax
          mov   ax,00815h
          mov   Var2,ax
          mov   ax,01234h
          mov   Var3,ax
          ret
CODE ENDS

END start
```

Beachten Sie bitte, daß in der ASSUME-Anweisung andere Beziehungen zwischen den Segmenten und ihren Registern hergestellt werden, als durch das Laden der Adressen in den ersten vier Zeilen des Quelltextes faktisch bestehen! Der Assembler moniert diese Anweisung nicht! Wie sollte er auch, denn er kann ja nur die Informationen benutzen, die ihm zur Verfügung stehen. Ob diese falsch sind, kann er nicht erkennen! Er stellt daher folgendes Assemblat her:

```
MOV     AX,0DDF
MOV     DS,AX
MOV     AX,0DCF
MOV     ES,AX
MOV     AX,1234
MOV     [0000],AX
MOV     AX,4711
MOV     [0002],AX
MOV     AX,0815
MOV     ES:[0000],AX
RET
```

Interpretieren wir diesen Code, so zeigt sich, daß $1234 an die Speicherstelle DS:[0000] geschrieben wird (das Datensegment ist Standardsegment für den MOV-Befehl, deshalb schreibt der Assembler nicht DS: vor die Speicherstelle [0000]) und der Wert $4711 an die Stelle DS:[0002]. Doch was passiert hier? Das Datensegment besitzt nur eine einzige Variable, nämlich *Var3*. Diese steht an Adresse DS:[0000]. Mit ihr endet das Segment definitionsgemäß. Es wird also eine Speicherstelle verwendet, die gar nicht existiert. Anders gesagt: Was nach dem Datensegment folgt, wissen wir nicht. Folgt z.B. ausführbarer Code, so würde dieser von der so harmlos aussehenden Zeile *mov [0002], ax* überschrieben, was mit einiger Sicherheit zum Absturz des Rechners, zumindest aber zu Problemen führt. Aber auch dann, wenn sich »nur« Daten in einem anderen, folgenden Segment anschließen, wird das Programm zumindest seltsame Reaktionen zeigen. Denn durch eben diese harmlos erscheinende Zeile werden Daten verändert, die eigentlich nicht verändert werden sollten – zumin-

dest nicht unkontrolliert! Die falsche Zuordnung von Segmenten zu ihren Segmentregistern und deren falsche Adressierung wird Ihnen sehr viel Frust bescheren. Hieraus resultierendes Fehlverhalten der Routinen läßt sich nur äußerst schwer und mit massivem Einsatz von Debuggern und viel Erfahrung finden – nach Stunden oder Tagen!

TIP

Zu jeder ASSUME-Anweisung gehört eine Ladeanweisung und die Prüfung, ob die Ladesequenzen, mit denen die Segmentregister beladen werden, korrekt programmiert sind. Es lohnt sich mit Sicherheit, dies einmal mehr zu überprüfen, als Ihnen notwendig erscheint.

ASSUME-Anweisungen lassen sich durchaus auch in den Quelltext einstreuen. Wenn Sie z.B. zwei Datensegmente verwenden, können Sie umschalten, etwa wie folgt:

```
DATEN1 SEGMENT BYTE PUBLIC
         Var1 dw ?
         Var2 dd ?
         :
         :
         Varn db ?
DATEN1 ENDS

DATEN2 SEGMENT BYTE PUBLIC
         Weiter1 dd ?
         Weiter2 dw ?
         :
         :
         Weitern dd ?
DATEN2 ENDS

CODE SEGMENT BYTE PUBLIC
ASSUME CS:CODE, DS:DATEN1
Quatsch:mov ax,SEG DATEN1
         mov ds,ax
         mov Var3,09876h
         mov Var2,01234h
         :
         :
ASSUME DS:DATEN2
         mov ax,SEG DATEN2
         mov ds,ax
         mov Weiter23,0ABCDh
         mov Weiter1,0FEDCh
         :
         :
```

```
ASSUME DS:DATEN1
         mov ax,SEG DATEN1
         mov ds,ax
         mov Var1,0F1E2h
         :
         :
         ret
CODE ENDS
END Quatsch
```

Dieses Vorgehen lohnt sich jedoch nur, falls Sie mehrfachen Zugriff auf die verschiedenen Segmente benötigen. Grundlos das Segment zu wechseln ist schlechter Programmierstil und führt in der Regel zu Geschwindigkeitseinbußen. Hierfür gibt es bessere Methoden.

21 Hallo, Coprozessor

Versuchen wir als nächstes, ein etwas sinnvolleres Programm zu erstellen. Wir werden die Existenz eines Coprozessors feststellen. Nun werden Sie einwenden, daß Ihre Hochsprache über Variablen verfügt, die Sie abfragen können und mit denen Sie feststellen können, ob ein solcher Coprozessor auch verfügbar ist. Sie haben recht! Aber dennoch: in unserem zweiten Programm werden wir durch diese Existenzabfrage einiges über das Assemblerprogrammieren im allgemeinen und über die Coprozessorprogrammierung im besonderen erfahren.

21.1 Unser zweites Programm

Das Programm heißt CHECK_87.ASM und ist auf der CD-ROM zum Buch enthalten. Hier ist der Quellcode:

```
CODE SEGMENT BYTE PUBLIC

Temp DW 0FFFFh
Env  DW 7 DUP (?)

Msg1 DB 'Kein $'
Msg2 DB 'Coprozessor vorhanden!',13,10,'$'
ASSUME CS:CODE, DS:CODE

Start:   mov      ax,cs
         mov      ds,ax
         fnstenv  [Env]
         fninit
```

```
          fnstsw    [Temp]
          cmp       BYTE PTR [Temp],000h
          jne       Nee
          fnstcw    [Temp]
          cmp       BYTE PTR [Temp+1],003h
          jne       Nee
          fldenv    [Env]
          jmp       Ist_da
Nee:      mov       ah,009h
          mov       dx,OFFSET Msg1
          int       021h
Ist_da:   mov       ah,009h
          mov       dx,OFFSET Msg2
          int       021h
          mov       ah,04Ch
          int       021h

CODE ENDS

END Start
```

Sie werden im Quellcode oben all die Dinge wiederfinden, die auch in unserem ersten Programm schon angesprochen wurden:

- `CODE SEGMENT BYTE PUBLIC`
  Speichersegmentierung: wir haben hier nur ein Segment, nämlich das Codesegment. Wir nennen es, wie gehabt, `CODE`, legen es auf die nächste Bytestelle fest und erklären es als `PUBLIC`, also frei für öffentlichen Zugriff. Wenn Sie Unterschiede zu der SEGMENT-Anweisung unseres ersten Programms feststellen, ist schon ein erster didaktischer Zielpunkt erreicht! Hier fehlt der »Name« des Segments, dafür wurde ein *Kombinationsoperand* angegeben. Wesentlich sind diese Unterschiede hier jedoch nicht. Wenn Sie sich etwas sicherer fühlen, schauen Sie einmal in Teil 3 unter »Assembleranweisunen«, in dem die SEGMENT-Anweisung detailliert beschrieben ist.

- `ASSUME CS:CODE, DS:CODE`
  Hier erfolgt »Anmeldung« der verwendeten Segmente beim Assembler: »Nimm an, daß sich in CS die Segmentadresse des CODE-Segments befindet. Auch in DS ist diese Adresse gespeichert!« Beachten Sie bitte auch, daß wir es bei dieser Anmeldung nicht belassen. Die ersten beiden Zeilen des eigentlichen Programmcodes laden tatsächlich die Adresse des Codesegments in DS. Wir kommen gleich darauf zurück.

- `CODE ENDS`
  Hierdurch wird dem Assembler das Ende des Segments mitgeteilt.

- `END Start`
  Auch die Tatsache, daß wir insgesamt fertig sind, erfährt er. Darüber hinaus sagen wir ihm auch, an welcher Stelle im Programm der ausführbare Code beginnt, nämlich am Label *Start*. Dies ist hier besonders wichtig, da vor dem eigentlichen ausführbaren Code noch Platz für Variablen liegt, die nicht ausgeführt werden dürfen!

Soviel zu den Anweisungen, die wir dem Assembler geben müssen, damit er weiß, was er mit dem Quelltext anfangen soll. Zusammengefaßt und etwas lockerer formuliert haben wir dem Assembler bisher nur mitgeteilt: »Es gibt ein Segment CODE. Dieses Segment, es liegt an Byte-Grenzen, enthält einen Einsprungspunkt für den Prozessor, an dem der ausführbare Code beginnt. Dieser Punkt heißt *Start*. Falls Du auf Daten in Variablen zurückgreifen mußt, suche sie ebenfalls im Segment CODE! Wo die dann tatsächlich stehen, sage ich Dir noch. An Befehlen, die Dich zu interessieren haben, berücksichtige lediglich all das, was zwischen dem Anfang der Datei und dem Punkt steht, der mit *END* bezeichnet wurde. Danke.«

Kommen wir nun zu den eigentlichen Befehlen. Wie schon gesagt, beginnen wir damit, das Segmentregister für die Daten mit der korrekten Segmentadresse zu belegen. Dies tun wir über die Sequenz:

```
mov     ax,cs
mov     ds,ax
```

Über den Kopierbefehl MOV holen wir uns die Adresse, die im Codesegmentregister CS steht, in das Prozessorregister AX. Beachten Sie bitte, daß immer der Inhalt des zweiten Operanden (hier: CS) in den ersten Operanden (hier: AX) kopiert wird. Anschließend kopieren wir wiederum den Inhalt aus AX in das Datensegmentregister DS. Dies sieht sehr umständlich aus und ist es auch! Aber leider ist der Befehl *MOV DS,CS* nicht erlaubt, so daß wir tatsächlich diesen Umweg gehen müssen. Was nach diesen beiden Zeilen erreicht ist, ist die Einhaltung des Versprechens, dafür zu sorgen, daß der Assembler die Anweisung *ASSUME CS:CODE, DS:CODE* für bare Münze nehmen kann!

Doch – wie kommt die Adresse des Codesegments in CS? Bitte erinnern Sie sich: Das Betriebssystem sorgt dafür, daß beim Start des Programms CS mit der korrekten Adresse geladen wird! Wir brauchen uns nicht darum zu kümmern.

21.2 Das eigentliche Programm

Wie kann man die Existenz eines Coprozessors feststellen? Die meisten Coprozessorbefehle setzen die Existenz des Coprozessors voraus und führen zum Absturz des Rechners, wenn keiner vorhanden ist. Aber es gibt einen Befehl, der den Coprozessor initialisiert – und es gibt ihn in einer Version, die »ungefährlich« ist: FNINIT!

FINIT bzw. seine »entschärfte« Form FNINIT stellt den Coprozessor auf einen bestimmten Anfangszustand ein: das Statusword des Prozessors wird gelöscht, und über das Kontrollwort werden bestimmte Bedingungen eingestellt. Also rufen wir einfach FINIT (FNINIT) auf und prüfen, ob das Statuswort des Prozessors tatsächlich 0 ist und im Kontrollwort die korrekten Anfangsbedingungen stehen. Ist beides der Fall, so gibt es einen Coprozessor. Trifft auch nur eine der beiden Bedingungen nicht zu, liegt auch kein Coprozessor vor.

Zunächst: das »N« in FNINIT ist extrem wichtig. Denn FINIT assembliert der Assembler zu:

```
WAIT
FINIT
```

Das bedeutet, daß vor dem INIT-Befehl ein WAIT eingestreut wird. Dies heißt nun, daß der Prozessor so lange wartet, bis der Coprozessor ihm signalisiert, daß er bereit ist. Wenn wir aber keinen Coprozessor haben? Dann wartet der Prozessor, und wartet, und wartet, ... FNINIT dagegen verhindert, daß der Assembler das WAIT voranstellt.

Außerdem verändern wir die Inhalte des Status- und Kontrollregisters durch den FINIT-Befehl. Dies ist jedoch in diesem Fall nicht besonders schlimm, da das Programm beendet wird, nachdem die Existenz des Coprozessors festgestellt wurde, der Coprozessor also anschließend nicht mehr benötigt wird. Andere Programme, die danach aufgerufen werden, müssen sich selbst darum kümmern, daß der Coprozessor die korrekten Registerinhalte hat.

Dennoch ist dies ein Beispiel für ausgesprochen schlechten Programmierstil! Es liegt in der Verantwortung des Programmierers, alles nach Beendigung des Programms so zu hinterlassen, wie er es vorgefunden hat. Die Nichtbeachtung dieser Regel ist ein Hauptgrund, der zu Inkompatibilitäten und Schwierigkeiten im Zusammenleben der unterschiedlichen Programme, Treiber und TSRs miteinander führt – und den man leicht verhindern kann, wenn man gewisse Spielregeln einhält.

Außerdem sollten Sie eines bedenken: Assemblerprogrammierung soll die Programmerstellung mit einer Hochsprache unterstützen – sie nicht ad absurdum führen! Denn natürlich initialisieren z.B. Pascal,

aber auch C und andere moderne Hochsprachen beim Programmstart den Coprozessor in einer Weise, wie die Sprache es verlangt. Wenn wir dies verhindern, indem wir in eigenen Assemblermodulen einfach ohne nachzudenken unkontrolliert Registerinhalte verändern, können wir uns gewaltige Probleme schaffen!

Da wir Registerinhalte des Coprozessors verändern (müssen), wenn wir seine Existenz nachweisen wollen, müssen wir diese vorher sichern und anschließend wiederherstellen. Die konsequente Anwendung solcher Regeln auch in Programmen oder Programmteilen, in denen dies eigentlich nicht notwendig ist, ist zum einen guter Programmierstil und führt zum anderen dazu, daß eventuelle Fehlerquellen gar nicht erst entstehen.

Es gibt einen Coprozessorbefehl, der alle Register mit Ausnahme der Rechenregister sichert: FSTENV. Auch hier müssen wir, damit wir keine Probleme mit den vom Assembler eingestreuten WAITs bekommen, die »N«-Variante dieses Befehls verwenden: FNSTENV. FSTENV/ FNSTENV besitzt einen Operanden, und zwar die Adresse einer Variablen, die sieben Worte faßt. In diese Variable wird dann der Inhalt der Prozessorregister kopiert. Also müssen wir noch daran denken, eine solche Variable zu kreieren. Nichts leichter als das: wir definieren irgendwo im Codesegment, allerdings außerhalb des ausführbaren Codes, ein Label *Env* und weisen ihm einen Speicherbereich von sieben Worten Größe zu:

```
Env DW 7 DUP (?)
```

DW
DUP

Die Assembleranweisung DW (*Define Word*) sagt dem Assembler, daß das Label *Env* auf einen Speicherbereich zeigt, der ein Wort beinhaltet. Die folgende 7 veranlaßt ihn, in Verbindung mit der Anweisung DUP (*Duplicate*) dieses Wort noch sechsmal zu duplizieren, so daß insgesamt 7 Duplikate des Worts existieren. Eine Standardvorgabe für ihren Inhalt wird nicht erzeugt, was das »?« ausdrückt. Frei formuliert heißt diese Zeile also: »Lieber Assembler, bitte erzeuge einen Speicherbereich von 7 Worten (= 14 Bytes) ohne definierten, vorgegebenen Inhalt, und lasse das Label *Env* auf das Anfangswort dieses Speicherbereichs zeigen!«

Nun können wir den Befehl FNSTENV benutzen:

```
fnstenv [Env]
```

Hiermit wird der Coprozessor veranlaßt, den Inhalt seiner Register (ohne die Rechenregister) in die Variable *Env* zu schreiben. Der Assembler setzt hier dank der Anweisung *ASSUME DS:CODE* die Adresse, wie wir noch sehen werden, explizit auf [0002]. Der Wert [0002] rührt daher, daß die Variable *Env* am Offset 0002 des Segments beginnt: Davor befindet sich ja noch die Variable *Temp*, die ein Wort, also 2 Bytes groß ist. Diese steht zu Beginn des Segments und hat daher den Offset 0000. Das Segmentpräfix

CS fehlt, da ja CODE als Datensegment angemeldet worden ist, der Assembler also weiß, daß in DS die Segmentadresse des Segments steht, in dem die Variable definiert ist (CODE!).

Als nächstes initialisieren wir den Coprozessor:

```
FNINIT
```

Dies führt dazu, daß das Statuswort auf 0 und das Kontrollwort des Coprozessors auf 3 gesetzt werden. Dies müssen wir nun überprüfen. Das geht jedoch nicht so einfach, da es keinen Coprozessorbefehl gibt, mit dem man direkt auf das Status- oder Kontrollwortregister des Coprozessors zugreifen kann. Allerdings gibt es analog zu FSTENV einen Befehl, den Inhalt des Statusworts (und den des Kontrollworts) in eine Variable zu kopieren: FSTSW und FSTCW – *Store Status Word* und *Store Control Word*. Wie bei FSTENV haben beide Befehle als Operanden die Adresse einer Variablen. In diesem Fall muß es nur eine Wortvariable sein, da die Register selbst nur ein Wort breit sind.

Nun müssen wir nicht beide Register gleichzeitig betrachten. Auch interessiert uns ihr Inhalt nur so lange, bis wir festgestellt haben, ob die Standardwerte für einen Coprozessor vorliegen. Daher brauchen wir nur eine Variable für beide Register. Weil wir sie nur temporär benötigen, nennen wir sie *Temp*.

Der Rest erfolgt ganz analog zu FSTENV: Definition eines Labels und Zuweisung eines Speicherbereichs von einem Wort Größe an dieses Label:

```
Temp DW OFFFFh
```

Im Programm wird dann zunächst das Statuswort betrachtet:

```
FNSTSW [Temp]
```

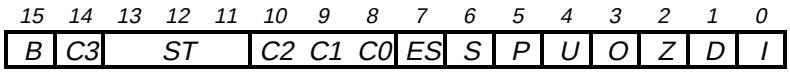
Auch hier muß die »N«-Variante des Befehls verwendet werden. Nun haben wir den Inhalt des Statusregisters des Coprozessors in der Variablen *Temp*, auf die auch der Prozessor Zugriff hat. Dieser verfügt über Befehle, mit denen man den Inhalt der Speicherstelle mit einer Konstanten vergleichen kann:

```
cmp BYTE PTR [Temp],000h
```

CMP ist ein Befehl, der den Inhalt des ersten Operanden mit dem des zweiten vergleicht. Dies erfolgt, indem der zweite Operand vom ersten arithmetisch abgezogen wird. Dann wird das Ergebnis betrachtet und die Flags anhand dieses Ergebnisses gesetzt. Das Ergebnis selbst wird dann verworfen; die Inhalte der beiden Operanden werden also nicht verändert.

Der erste Operand sieht etwas ungewöhnlich aus: `BYTE PTR [Temp]`! Warum nicht schlicht und einfach `[Temp]`? Schließlich wollen wir doch

den Inhalt von *Temp* mit einer Konstanten vergleichen! Dazu müssen wir uns noch einmal vergegenwärtigen, was im *Statuswort* des Coprozessors alles steht:



Wie wir aus Teil 1 dieses Buches wissen, sollte nach der Initialisierung des Coprozessors kein Ausnahmezustand herrschen. Das heißt, die Bits 5 bis 0 sollten gelöscht sein. Zusätzlich sollte Bit 7 ebenso durch seinen gelöschten Zustand signalisieren, daß kein Interrupt ausgelöst worden bzw. keine Exception aufgetreten ist[8]. Eigentlich sollten auch die Bits 15 bis 14 sowie 10 bis 8 gelöscht sein, da der Coprozessor nach der Initialisierung weder beschäftigt (*busy*) ist noch irgendeinen Condition Code anzeigt. Der Stack-Pointer sollte Register 0 als TOS angeben, also sollten auch die Bits 13 bis 11 gelöscht sein. Aber sicher sein darf man sich hier nicht – machmal kochen unterschiedliche Coprozessorhersteller unterschiedliche Süppchen, es könnte also doch irgendein Condition Code gesetzt sein.

PTR

Letzlich sind wir uns daher nur beim unteren Byte des Kontrollwortes sicher. Dies muß 0 sein, wenn FINIT durchgeführt wurde. Daher wird auch nur dieses Byte mit CMP überprüft. Wir müssen dem Assembler also mitteilen, daß wir nur das niederwertige Byte vergleichen wollen. Daher führen wir mit *BYTE PTR* etwas durch, das man in Hochsprachen als *Type-Casting* bezeichnen würde. Weil *Temp* als Wortvariable definiert ist, würde der Befehl *CMP [Temp], 000h* das Wort in *Temp* mit dem Wort 0 vergleichen (auch wenn wir nur zwei Ziffern angeben. Der Assembler erweitert sie automatisch auf vier zu 00000h – die obligatorische führende 0 bei Hexadezimalzahlen nicht mitgerechnet). Durch den Operator PTR (= *Pointer*) in *CMP BYTE PTR [Temp], 000h* machen wir dem Assembler klar, daß wir nur »das Byte, auf das das Label *Temp* zeigt« untersuchen wollen. PTR ist also eine Assembleranweisung, die bewirkt, daß die Adresse, für die das Label steht, als Adresse eines Datums interpretiert wird, das durch die Assembleranweisung unmittelbar davor genauer spezifiziert wird. Sie sehen, es ist also nichts Ungewöhnliches!

Wenn Sie einmal genauer nachdenken und sich das Diagramm von eben noch einmal anschauen, in dem die Position der Flags dargestellt ist, so wollen wir das niederwertige Byte des *Statusworts* prüfen.

---

[8] Da Interrupts nur beim 8087 interessant waren, signalisiert bei den neueren Prozessoren dieses Bit lediglich eine ODER-Verknüpfung der Bits 6 bis 0. Beachten Sie auch, daß Bit 6, *Stack Fault*, erst ab dem 80386 definiert ist.

Wenn wir aber ein Wort definieren und ein Label auf dieses Wort angeben, so wäre es doch logisch, daß dieses Label auf den Anfang des Wortes zeigt, quasi auf Bit 15 des Wortes. Das hieße aber, daß wir auf das dem Label *Temp* folgende Byte zeigen, also *CMP BYTE PTR [Temp+1], 000h* programmieren müßten.

ACHTUNG

Das wäre tatsächlich logisch! Aber wer behauptet, daß in der Computerei alles logisch zugehen soll? Niemand! Das dachte sich wahrscheinlich auch Intel und legte einfach fest, daß im Speicher alle niederwertigen Bytes von Daten vor den höherwertigen stehen, daß also in Prozessorregistern zwar $4711 geschrieben steht, im Speicher jedoch $1147! Das aber heißt nun nicht, daß im Speicher etwas Falsches steht! Nein – man muß Speicherinhalte »von hinten nach vorn« auslesen. Wenn man dies tut, so steht auch im Speicher $4711. Aber nur im Speicher! Im Prozessorregister ist alles »normal«.

So weit, so gut. Durch *CMP BYTE PTR [Temp], 000h* haben wir also die Flags des Prozessors anhand des Ergebnisses der »in Gedanken durchgeführten« Subtraktion der Kostanten 0 vom Inhalt des Bytes in Temp gesetzt. Welche Flags können nun also gesetzt sein – und was bedeutet dies?

- Falls in *Temp* (genauer im niederwertigen Byte von *Temp*) auch 0 stand, so ist das Ergebnis dieser Subtraktion ebenfalls 0, weshalb das Zero-Flag gesetzt ist.
- Stand in *Temp* jedoch ein von 0 verschiedener Wert, so führt die Subtraktion von 0 von diesem Wert wieder zum gleichen Wert. Das bedeutet, daß das Zero-Flag gelöscht ist, weil das Subtraktionsergebnis ja nicht 0 ist. Das Carry-Flag, das Auxiliary-Flag und das Sign-Flag dagegen sind in Abhängigkeit vom Wert in *Temp* gesetzt oder gelöscht. Welchen Zustand sie tatsächlich haben, braucht uns nicht zu interessieren. Denn wir wollen ja nur wissen, ob das Subtraktionsergebnis 0 ist oder nicht. Daher brauchen wir auch nicht zu erwägen, ob der Inhalt in *Temp* vorzeichenlos oder vorzeichenbehaftet zu interpretieren ist – Sie erinnern sich an Teil 1 des Buches!

Das heißt nun, daß wir lediglich das Zero-Flag untersuchen. Dies tun wir, indem wir unser Programm in Abhängigkeit vom Zustand dieses Flags verzweigen. Das Programm soll die Meldung »Kein Coprozessor vorhanden« ausgeben, wenn kein Coprozessor vorhanden ist, andernfalls »Coprozessor vorhanden«. Der Unterschied zwischen beiden Meldungen ist lediglich, daß im Falle der Nichtexistenz das Wörtchen »Kein« gefolgt von einem Leerzeichen zusätzlich ausgegeben werden muß.

Das bedeutet also, daß wir die Textausgabe in zwei Schritten vornehmen können.

```
Nee:    mov ah,009h
        mov   dx,OFFSET Msg1
        int   021h
```

Zuerst die Ausgabe der *message1*, »Kein «. Das erfolgt wie bei HI_WORLD über eine darauf spezialisierte DOS-Systemroutine. Dieser Programmteil erhält das Label *Nee*. Unmittelbar anschließend erfolgt die Ausgabe des restlichen Teils, also *message2*, »Coprozessor vorhanden«; erreichbar durch das Label *Ist_da*.

```
Ist_da: mov   ah,009h
        mov   dx,OFFSET Msg2
        int   021h
```

Der Rest ist einfach: Ist bei der Prüfung das Zero-Flag nicht gesetzt, so wurde nicht 0 in das Statuswort geschrieben, es kann also kein Coprozessor vorhanden sein. Folgerichtig ist der darauf folgende Befehl ein Sprung zu dem Label, das die Meldung »Kein « ausgibt. Die dafür zuständigen bedingten Sprungbefehle sind JNZ, also *Jump If No Zero Flag Set* oder JNE, *Jump If Not Equal*, was ja gleichbedeutend ist, denn das Zero-Flag wird ja nur dann nicht gesetzt, wenn das Ergebnis der Subtraktion nicht 0 ist. Das kann nur sein, wenn beide Operanden nicht gleich groß, also not equal, sind (de facto werden beide Befehle vom Assembler in den gleichen Opcode übersetzt.).

Falls aber 0 gefunden wurde, das Zero-Flag also gesetzt ist, so überprüfen wir sicherheithalber noch das Kontrollwort. Denn wir wissen ja, daß dieses einen bestimmten Wert erhält! Der Inhalt von *Temp* hat zu diesem Zeitpunkt seine Schuldigkeit getan, wir brauchen ihn nicht mehr. Daher können wir *Temp* als Operand für den nächsten Befehl, FNSTCW, verwenden. Es geht weiter mit:

- der Speicherung des Inhalts des Kontrollworts in die Variable *Temp* durch *FNSTCW [Temp]* (auch hier wird wieder die »N«-Variante eingesetzt),
- der Prüfung, ob (in diesem Fall) das höherwertige Byte von *Temp* den Wert $03 hat, was FINIT als Initialisierung einträgt, mit Hilfe des Vergleichs CMP und der Konstanten 003h (auch hier wieder die Notwendigkeit von *BYTE PTR*. Und weil es diesmal das höherwertige Byte ist, muß zur Adresse von *Temp* noch 1 Byte addiert werden, also *BYTE PTR [Temp+1]*.) und
- dem Sprung an die Stelle, an der »Kein « ausgegeben wird, also an das Label *Nee*, falls das Zero-Flag nicht gesetzt ist.
- Wurde kein bedingter Sprung durchgeführt, so ist ein Coprozessor vorhanden! In diesem Fall müssen wir die Prozessorregister wiederherstellen, die wir am Anfang des Programms gesichert

haben. Dies tun wir mit *FLDENV [Env]*, einem Befehl, der kein »N«-Analogon hat. Aus diesem Grunde dürfen wir ihn auch nur dann ausführen, wenn sichergestellt ist, daß ein Coprozessor vorhanden ist. Wurde dieser Befehl abgearbeitet, so können wir die Ausgaben von »Kein « getrost überspringen, was wir mit dem unbedingten Befehl *JMP Ist_da* realisieren.

Was nun noch kommt, ist Routine, nämlich der geordnete Rückzug aus dem Programm durch Aufruf der entsprechenden Betriebssystemroutine in Form des DOS-Interrupts, den wir ebenfalls aus HI_WORLD schon kennen.

Eine kleine Denkaufgabe zum Schluß! Warum wird der Assembler oben angewiesen, die Variable *Temp* mit dem Initialisierungswert $FFFF zu belegen? Die Lösung steht auf Seite 289.

22 Wie heißt Du, Coprozessor?

Wir haben nun ein Programm, das uns auf dem Bildschirm mitteilt, ob ein Coprozessor vorhanden ist. Aber vielleicht ist es von Interesse, welcher Coprozessor dies ist. So kennen z.B. der 80387 und sein Nachfolger den sehr leistungsstarken Befehl FSIN, alle Vorgänger dagegen nicht. Wenn man also die wirklich sinnvollen Verbesserungen dieser Coprozessoren ausnutzen will, auf der anderen Seite dagegen kompatibel mit den Vorgängern bleiben möchte (oder muß), so kommt man nicht um eine Prüfung herum, welcher Typ Coprozessor vorhanden ist.

Es ist nicht schwierig, den Coprozessortyp festzustellen. Im ersten Teil dieses Buches haben wir bei der Besprechung der Coprozessorbefehle festgestellt, daß es teilweise Unterschiede in der Behandlung von NaNs gibt (genauer gesagt: in der Art, wie Unendlichkeiten zu interpretieren sind!) oder daß bestimmte Befehle bei den unterschiedlichen Typen anders ablaufen. Diese Tatsache nutzen wir nun aus.

22.1 Die Strategie

Zunächst müssen wir feststellen, welche Coprozessoren es überhaupt gibt und welche Unterschiede zu den anderen sie aufweisen:

- Pentium. Dieser Prozessor hat keinen Coprozessor. Besser gesagt: Er hat einen, der jedoch als Floating-Point-Unit bezeichnet wird, auf dem Pentium-Chip integriert ist und anders arbeitet als die bekannten Coprozessoren. Die »Coprozessor«-Befehle sind jedoch die gleichen wie bei allen anderen. Dennoch ist eine Prüfung auf

einen Coprozessor bei Vorliegen eines Pentiums nicht erforderlich, da der neue Befehl CPUID ja entsprechende Informationen gibt. Daher wird in diesem Fall der Pentium nicht berücksichtigt.

- 80487. Dieser Coprozessor ist auch eigentlich nicht existent – er wurde auch auf dem Chip des 80486 integriert (wenn überhaupt). Also können wir seine Existenz dadurch feststellen, daß wir prüfen, ob ein Coprozessor vorhanden ist und, wenn ja, ob es ein 80486 ist.
- 80387. Ist dies nicht der Fall, so könnte, falls nachweislich ein Coprozessor existent ist, ein 80387 installiert sein. Ihn finden wir leicht: Da dieser Coprozessor als erster Unendlichkeiten grundsätzlich unterschiedlich behandelt, brauchen wir nur zu prüfen, ob das Umkehren des Vorzeichens einer Unendlichkeit unterschiedlich behandelt wird. Wenn ja, dann liegt ein 80387 vor.
- 80287. Hier wird es etwas kritischer. Es kann lediglich noch ein 8087 oder ein 80287 vorhanden sein! Diese unterscheiden sich aber nicht so offensichtlich voneinander. Daher können wir als Unterscheidungsmerkmal nur eine Eigenheit verwenden, die nicht sehr bekannt ist: Der Befehl FSTENV speichert neben allen Registern des Coprozessors auch die Adresse des Befehls, der auf FSTENV folgt. Aber nur beim 80287. Der 8087 speichert dagegen die Adresse des FSTENV-Befehls selbst.
- 8087. Konnte keiner der oben genannten Tests einen Coprozessor auffinden, so kann nur noch ein 8087 vorhanden sein, einen 80187 hat es nie gegeben!

Routinen

Das bedeutet aber, daß wir zunächst die Anwesenheit eines Coprozessors feststellen müssen. Das aber haben wir im letzten Abschnitt schon getan, so daß wir diesen Teil des Programms schnell realisiert haben. Dennoch wollen wir hier eine kleine Änderung vornehmen. Dieser Anwesenheitsnachweis war im Programm CHECK_87 die bloße Existenzberechtigung, alleiniger Sinn und Zweck. Jetzt spielt er eigentlich nur noch eine untergeordnete, wenn auch entscheidende Rolle! Also stellen wir ihn in ein Unterprogramm.

Sicher kennen Sie Unterprogramme aus den Hochsprachen. Dennoch gibt es zu Unterprogrammen beim Assembler eine Menge zu sagen. Der Aufruf von Unterprogrammen ist nämlich nicht trivial. Hochsprachencompiler nehmen Ihnen sehr viel Arbeit ab, wenn Sie dort ein Unterprogramm ausführen wollen. Sie rufen, egal in welcher Sprache, einfach das Unterprogramm durch Nennung seines Namens auf, übergeben ihm ggf. ein paar Parameter und erwarten, falls das Unterprogramm eine Funktion war, ein Ergebnis. Doch was passiert auf Assembler-Ebene?

Nehmen wir an, daß der Prozessor schon eine Reihe von Befehlen abgearbeitet hat. Nun stößt er auf einen CALL-Befehl, mit dem man Unterprogramme im Assembler aufruft. Betrachten wir zunächst den einfachsten Fall, nämlich den Aufruf einer Prozedur, also eines funktionswertlosen Unterprogramms, dem in diesem Fall auch keine Parameter übergeben werden!

Klar ist, daß der Prozessor nach dem Abarbeiten des Unterprogramms wieder an die Stelle zurückkehren soll, an der es aufgerufen wurde! Also muß sich der Prozessor irgendwie merken, an welcher Stelle er das Programm unterbrochen hat. Doch wohin mit der Rücksprungadresse? Prozessorregister kommen nicht in Frage, da diese eventuell im Unterprogramm für andere Zwecke benötigt werden. Ferner soll ja auch das Verschachteln von Unterprogrammen möglich sein, also das Aufrufen eines Unterprogramms aus einem Unterprogramm eines Unterprogramms in einem Unterprogramm usw. Mit den acht Prozessorregistern kommen wir da nicht weit!

22.2 Ein Stapel Worte

Stack

Also Speicherstellen. Aber wo? Uns fiele so etwas leicht, denn wir können für unsere Daten ja ein Datensegment definieren, in das wir alles mögliche, also auch Rücksprungadressen ablegen könnten. Aber der Prozessor? Der eben auch! Intel sagte sich: »Was dem Programmierer recht ist, ist dem Prozessor billig!« und spendierte ihm ein eigenes Datensegment mit allem Drum und Dran! Dieses Segment wurde Stack genannt – wir kommen noch darauf zurück, warum! Der Stack ist das Datensegment des Prozessors, das dieser für interne Zwecke verwenden kann. Die Adresse dieses Segments kennt der Prozessor zu jedem Zeitpunkt, da sie im Registerpaar SS:SP eingetragen ist.

Der Stack ist also der Ort, wo der Prozessor die Rücksprungadresse ablegt. Dies erfolgt durch den CALL-Befehl, der – falls es sich um ein Unterprogramm handelt, das im gleichen Segment liegt –, nur den Offsetanteil, andernfalls die vollständige Segment-Offset-Kombination dorthin kopiert, bevor die neue Adresse in CS:IP geladen wird. Der RET-Befehl, mit dem üblicherweise die Unterprogramme beendet werden, holt nun die eben abgelegte Adresse wieder vom Stack und schreibt sie entweder in das IP-Register, falls es ein *Near Call* war (das Unterprogramm liegt dann im gleichen Segment wie das aufrufende Programm) oder eben in CS:IP. Dies führt dann dazu, daß der Prozessor mit dem auf den CALL-Befehl folgenden Befehl fortfahren kann.

Wie Sie sehen, brauchen Sie sich um diese Dinge nicht zu kümmern! Der Prozessor verwaltet sein »Datensegment« selbständig, ohne Ihr Zutun. Sie müssen lediglich dafür sorgen, daß er überhaupt Speicher zugeteilt bekommt, den er als Stack nutzen kann, und daß dieser Speicher groß genug ist! Denn die Verantwortung für die Auslastung des Speichers überläßt er *Ihnen*! Nun ist es aber sehr einfach, dies zu erreichen. Sie brauchen nur ein Segment zu definieren, das den geschützten Namen Stack erhält:

```
STACK SEGMENT WORD

DW       00100h DUP (?)

STACK ENDS
```

Mit diesem Programmteil definieren wir ein Segment mit dem Namen STACK und richten es auf Wortgrenzen aus. Der Hintergrund hierzu ist, daß Stacks immer nur wortweise verwendet werden können. Das Ausrichten auf ungerade Speicherstellen, wie es beim *Alignment* mittels BYTE passieren könnte, würde den Zugriff auf den Stack verlangsamen. In diesem Segment werden durch die DW-Anweisung $100 also 256 Worte reserviert, die nicht vorbelegt werden!

HINWEIS Assemblerprogramme können z.T. sehr »stackintensiv« sein, das heißt einen großen Stack benötigen. Hier ist generell keine Regel anzugeben, was »stackintensiv« heißen kann! So hängt die benötigte Größe des Stacks von vielen Faktoren ab, beispielsweise von der Anzahl verschachtelter Unterprogramme (etwa bei Rekursionen). Oder, wie wir noch sehen werden, von der Art und Anzahl von Parametern oder lokalen Variablen! Oder von der Art der Programmierung von anderen Programmen (TSR), die in (hoffentlich) friedlicher Koexistenz mit Ihrem Assemblerprogramm zusammenarbeiten.

ACHTUNG SS:SP hat immer einen Inhalt, egal ob Sie einen Stack definiert haben oder nicht. Wenn *Sie* keinen eingerichtet haben, Ihr Programm aber einen Stack benötigt, so benutzt es den, den jemand anderes irgendwann einmal eingerichtet hat. Im Zweifelsfall ist dies das Betriebssystem! Dieses geht in der Regel recht knausrig mit der Stackgröße um. Das heißt, daß es nicht damit rechnet, daß andere den DOS-eigenen Stack mitbenutzen und somit wenig »Raum« übrig läßt. Dies kann dann zu Problemen führen. Denn für den Stack gilt wie für jedes andere Segment auch: Falls die Segmentgrenzen überschritten werden, werden Daten verändert, die nicht verändert werden sollten. Böse Konsequenzen sind dann die Folge.

Dies ist auch mit ein Grund, warum »unsauber« programmierte Programme häufig zu unschönen Resultaten führen. Denn ein TSProgramm, das ebenfalls keinen Stack definiert, aber selbst Speicher benötigt, benutzt Ihren Stack! Sie werden in der Regel auch nicht daran denken, daß andere dies nicht berücksichtigen könnten.

Übrigens müssen Sie außer der Definition des Stacks nichts weiter unternehmen! Der Assembler erkennt am reservierten Namen *STACK*, daß ein solcher Datenspeicher vorhanden ist und sorgt dafür, daß SS:SP zu Programmbeginn den korrekten Inhalt bekommt!

Doch zurück zu unserem Programm. 256 Bytes Stack sollten mehr als ausreichend sein, da unsere Unterprogramme keine Parameter übergeben bekommen und auch keine lokalen Variablen anlegen. Daher benötigt der Prozessor eigentlich maximal 2 Bytes davon für die Rücksprungadresse. Da die Unterprogramme alle im gleichen Segment angesiedelt sind, werden sie per *Near Call* aufgerufen, und es muß nur der Inhalt von IP gesichert werden!

22.3 Bürokratie!

Nun folgt die übliche Verwaltungsarbeit, die wir schon von den beiden vorangehenden Beispielen her kennen. Zusätzlich definieren wir einige Variablen, die wir etwas später erklären und benutzen werden. Beachten Sie bitte, daß hier neben den schon bekannten Deklarationen von Bytes und Worten für Variablen auch Doppelworte reserviert werden! Zuständig hierfür ist die Assembleranweisung DD (Define Dword).

DD

Mit DD reserviert man vier Bytes Platz, der für Variablen verwendet werden kann. Aber Achtung! Prozessor und Coprozessor belegen diesen Platz anders!

ACHTUNG

Wenn Sie DDs reservieren, die der Coprozessor nutzen soll, und diesen DDs einen Startwert zuweisen wollen, so achten Sie peinlich genau darauf, was für einen Zahlentyp die DD aufnehmen soll. Falls sie nämlich für Integer gedacht ist, so können Sie ganz normal eine Deklaration der Form

```
IntDD  DD 4711
```

vornehmen! Die DD enthält nun den Wert 4711 im Integerformat, das sowohl der Prozessor als auch der Coprozessor verarbeiten kann, wenn letzterer sie mit FILD ausliest! Soll dagegen eine Realzahl in DD gespeichert werden, so muß die Zahl auch als Realzahl mit Dezimalpunkt und mindestens einer Nachkommastelle angegeben werden, wie etwa:

```
RealDD  DD 4711.0
```

Der Hintergrund ist einfach: Wie soll der Assembler wissen, ob »4711« als Integer oder als Real gespeichert werden soll? Das muß er wissen, da Integer- und Realformat nicht miteinander kompatibel sind. Die Angabe eines Punktes mit einer Nachkommastelle aber legt den Zahlentyp eindeutig fest.

HINWEIS Die Deklaration einer Realzahl ohne Dezimalpunkt und Nachkommastelle führt bei einem Zugriff mit Realzahl-Befehlen des Coprozessors zu falschen Ergebnissen!

```
CODE SEGMENT WORD PUBLIC

Temp DW OFFFFh
Env  DW 7 DUP (?)
X_87 DB 0C1h
T_87 DB 0FFh

C_087  DD 8237.97
C_187  DD ?
C_287  DD 8237.97
C_387  DD 3540.25
C_487  DD 2598.96

Msg1       DB 'Kein Coprozessor vorhanden!',13,10,'$'
Msg2       DB 'Coprozessor-Typ: $'
Msg487     DB '80487',13,10,'$'
Msg387     DB '80387',13,10,'$'
Msg287     DB '80287',13,10,'$'
Msg087     DB '8087',13,10,'$'
SoSchnell  DB 'Coprozessorgeschwindigkeit: '
Hierhin    DW ?
           DB ' MHz',13,10,'$'

OffsetTable DW OFFSET Msg087
            DW ?
            DW OFFSET Msg287
            DW OFFSET Msg387
            DW OFFSET Msg487

ASSUME CS:CODE, DS:CODE, SS:STACK
```

Beachten Sie bitte, daß wir mittels der ASSUME-Anweisung dem Assembler mitteilen, daß im Register DS die Segmentadresse des Codesegments steht, daß also alle Befehle, die DS als Segmentanteil bei Adressen verwenden, automatisch auf Speicherstellen im Codesegment zugreifen. Und wie eventuelle Datensegmente auch, müssen wir das Stacksegment beim Assembler anmelden!

22.4 Unterprogramme

Wie in Hochsprachen auch, kann der Assembler nur Dinge verarbeiten, die er kennt. Wenn wir mit Unterprogrammen arbeiten wollen, so müssen wir diese daher erst programmieren, bevor wir sie im Hauptprogramm nutzen können:

```
Check_87 PROC NEAR
```

Nach unseren Regeln läßt sich diese Zeile wie folgt analysieren:

- Es gibt ein Label namens *Check_87* (weil wir dies in der bei uns üblichen Groß-/Kleinschreibung angeben).
- Es folgen zwei Assembleranweisungen: PROC und NEAR (die wir als Anweisungen erkennen, weil sie groß geschrieben werden).

Im Klartext heißt diese Zeile: »An dieser Stelle steht ein Label mit Namen *Check_87*. Dieses Label zeigt auf den Beginn einer Routine (*PROCedure*), die im gleichen Segment liegt wie das Hauptprogramm und daher über einen *NEAR Call* angesprungen werden kann.« Die Unterscheidung zwischen Prozeduren und Funktionen ist eine »Erfindung« der Hochsprachen. Der Assembler kennt diesen Unterschied nicht. Für ihn sind alle Programmteile, die mittels CALL aufgerufen werden, *procedures*!

ACHTUNG

Ob der Assembler *Near-* oder *Far Calls* erzeugt, entscheidet nicht er, wie es analog in Hochsprachen der Compiler tut, sondern *Sie*! In *dieser* Definition, durch die Assembleranweisung *NEAR* oder *FAR*. Dies kann zu Problemen führen!

Es ist alles unproblematisch, solange Sie nur ein Codesegment haben und innerhalb dieses Segments bleiben. Dann sind alle Sprünge durch *Near Jumps* und alle Unterprogrammaufrufe durch *Near Calls* realisierbar. Denn weil Sie ja im gleichen Segment bleiben, braucht der Segmentanteil der Adresse in CS nicht verändert zu werden.

Dies ändert sich drastisch, wenn mehrere Segmente vorliegen. Dies ist immer dann der Fall, wenn Sie z.B. Assemblermodule in Hochsprachen einbinden, oder wenn Sie Assemblerroutinen entwickeln, die in Units oder anderen Modulen einer Hochsprache implementiert, aber vom Hauptprogramm aus aufgerufen werden. In diesen Fällen befinden sich die betroffenen Codesequenzen mit Sicherheit in unterschiedlichen Segmenten! In solchen Fällen müssen *Far Calls* verwendet werden, weil Sie ja mit dem Sprung auch das Segment wechseln müssen.

TIP

Wenn Sie sich also unsicher sind, ob *Near-* oder *Far Calls* angebracht sind, oder wenn während der Entwicklung von Routinen der Rechner abstürzt, dann benutzen Sie im Zweifel immer *Far Calls* und eine Definition der

Routinen mittels PROC FAR! Dies schadet nie und verhindert Probleme. Einzige Konsequenz: Der Prozessor verschwendet immer dann zwei Bytes auf dem Stack, wenn Sie eine Routine aufrufen, die Sie als *far* deklariert haben, die sich aber im gleichen Segment befindet und daher *near* aufgerufen werden könnte. Aber wenn Ihr Stack ausreichend dimensioniert ist …

Was nun kommt, ist der Code der Routine:

```
push    ds
push    cs
pop     ds
cmp     [HX_87],0C1h
jne     Ende
xor     ah,ah
```

Es ist guter Programmierstil, als eine der ersten Maßnahmen in einer Routine den Inhalt des Datensegmentregisters zu sichern, wenn dieser in ihr verändert wird. Denn andere Programmteile verlassen sich darauf, daß immer derjenige Veränderungen wieder rückgängig macht, der sie verursacht! Das heißt hier, daß ein Programm, das ein Unterprogramm aufruft, nicht »wissen« kann, ob (und wenn ja wie) der Inhalt von DS durch die Routine verändert wurde. Eventuell benutzt das aufrufende Programm ein anderes Datensegment als das aufgerufene. In unserem Beispiel spielt das allerdings keine Rolle, da das gesamte Programm nur auf Daten im Codesegment zurückgreift.

Mit `push ds` sichern wir also den Inhalt von DS. Doch wohin? Auf den Stack! Spätestens jetzt müssen wir uns ihm etwas genauer widmen!

22.5 Ein weiterer Stapel Worte

Stack heißt »Stapel«. Mit dieser einfachen Übersetzung des englischen Fachbegriffs ist eigentlich schon alles gesagt! Oder vielleicht doch nicht. Überlegen wir uns nämlich, wie der Prozessor sein Datensegment, den Stack, überhaupt verwalten kann!

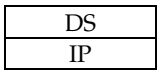
Der Prozessor kann sich nicht merken, daß in der Variablen *Ruecksprungadresse* die Rücksprungadresse verzeichnet ist, die nach Beendigung der Routine verwendet werden soll (andernfalls könnte er sich ja gleich die Adresse merken). Daher verzichtet er vollständig auf alle Namensgebungen jedweder Art und organisiert seinen Datenbereich als Stapel, den man sich wie einen Stapel Teller oder etwas ähnliches vorstellen kann. Wenn Sie sich schon intensiver mit Teil 1 des Buches auseinandergesetzt haben, so werden Sie sicherlich spätestens jetzt an den Stack des Coprozessors denken!

Aber Achtung: Der Stack des Coprozessors und der des Prozessors haben außer dem Namen nichts gemeinsam!

Das Bild des Stapels Teller ist gar nicht schlecht. Wenn Sie abwaschen, nehmen Sie jeweils einen Teller, spülen ihn und legen ihn nach dem Abtrocknen auf den Stapel, der schon besteht. Wenn noch kein Stapel existiert, so eröffnen Sie einen mit dem ersten Teller. Analog verfährt der Prozessor. Wann immer er ein Datum zu sichern hat, legt er es auf dem Stapel ab. Zuoberst auf dem Stapel ist daher immer das zuletzt gesicherte Datum.

Umgekehrt nehmen Sie immer den obersten Teller vom Stapel, wenn Sie einen benötigen. So auch der Prozessor: Er kann nur das Datum vom Stapel nehmen, das sich zuoberst befindet. Ein Stapel realisiert somit eine *LIFO*-Struktur. In ihr wird immer das Element, das zuletzt in die Struktur geschrieben wurde (Last In), zuerst wieder ausgelesen (First Out). Eine andere Struktur, die *Queue* oder Schlange, verwendet eine andere Strategie (*FIFO*; First In First Out).

Das aber bedeutet für unseren Programmausschnitt, daß sich nun auf dem Stack über der gesicherten Rücksprungadresse der Inhalt von DS befindet:



Wenn Sie dies nicht nachvollziehen können, so blättern Sie bitte ein paar Seiten zurück. Das Verständnis dieses Vorgangs ist elementar für die Assemblerprogrammierung! Daher sollten Sie diese Zusammenhänge verstanden haben. Hier noch einmal eine Hilfe: Der Inhalt des untersten Wortes (Stacks arbeiten, wie gesagt, immer wortweise!) ist die Rücksprungadresse, die hier lediglich aus dem Offset, also dem Inhalt von IP besteht, da die Routine *near* aufgerufen wird. Nach *push ds* wurde »auf« diese Adresse der Inhalt von DS kopiert, was zu diesem Bild führt.

Doch weiter mit der Routine. Die nächsten beiden Befehle dienen zum Kopieren des Codesegments in das DS-Register. Hierzu schieben wir kurzfristig den Inhalt von CS mittels *push cs* auf den Stack, um ihn mit *pop ds* sofort wieder vom Stack in das DS-Register zu laden. Diese Nutzung des Stacks ist sehr häufig und beliebt. Man hätte dies auch über

```
mov     ax,cs
mov     ds,ax
```

realisieren können, hätte dazu aber den Inhalt von AX verändert. Anschließend vergleichen wir den Inhalt der Variablen mit $C1: *cmp*

[X_87], 0C1h. *X_87* ist ein Byte, das uns Auskunft darüber gibt, ob die Routine schon einmal aufgerufen wurde. Denn wir brauchen ja nur einmal festzustellen, ob ein Coprozessor installiert ist. Falls diese Information häufiger benötigt würde, bräuchte lediglich dieses Byte getestet zu werden. Dies erspart manchmal das unnötige Aufrufen z.T. sehr langer Routinen.

Doch wie können wir feststellen, ob die Routine schon einmal durchlaufen wurde? Bei der Definition der Variablen weiter oben haben wir den Assembler angewiesen, der Variablen einen festen Startwert, nämlich $C1, beim Programmstart mitzugeben. Hochsprachen nennen solche Variablen mit Startwert *Konstanten*. Womit auch en passant der Grund erklärt ist, daß in Pascal z.B. Konstanten alles andere als konstant sind: Sie können ebenso wie Variablen während des Programmablaufs verändert werden. Der einzige Unterschied zu »echten« Variablen besteht also lediglich darin, daß Konstanten einen Startwert besitzen, Variablen nicht (was in Assembler durch ein »?« in der Variablendefinition angezeigt wird).

Wenn wir nun diesen Startwert nach dem Abarbeiten der Routine verändern, so können wir bei einem erneuten Aufruf der Routine feststellen, daß die Coprozessorexistenz schon geklärt ist und die Routine beendet werden kann. So führt konsequenterweise der folgende bedingte Sprungbefehl genau diese Aktion aus. Falls nämlich der Vergleich Identität erbracht hat, so ist das Zero-Flag gesetzt (Sie wissen ja: CMP ist eigentlich eine Subtraktion und ergibt daher »0«, wenn beide Operanden gleich sind), und die Routine ist noch nicht durchlaufen worden. JNE wird dann nicht verzweigen. Unterscheiden sich dagegen beide Werte, so wird JNE aufgrund des gelöschten Zero-Flags den Prozessor veranlassen, zum Label *Ende* zu springen (JNE ist ja der gleiche Befehl wie JNZ, *Jump If Not Zero Flag Set*). $C1 wurde willkürlich als Startwert genommen.

Der letzte Befehl in der Sequenz oben dient nur der Optik: Hier wird das AH-Register gelöscht. Warum, klären wir später!

TIP

Das AH-Register wird durch *xor ah, ah* gelöscht, da eine exklusive Oder-Verknüpfung ein Bit immer dann löscht, wenn die zu verknüpfenden Bits entweder beide 0 oder beide 1 sind. Dies ist bei einer XOR-Verknüpfung von zwei identischen Bytes immer der Fall. XOR mit identischen Operanden aufgerufen ist also das gleiche wie das konkrete Löschen eines Bytes mit MOV, nur viel schneller und mit weniger Bytes für den Opcode.

Was nun kommt, kennen Sie aus unserem letzten Programm:

```
fninit
fnstst  [Temp]
cmp     BYTE PTR [Temp],000h
```

```
        jne     No_87
        fnstcw  [Temp]
        cmp     BYTE PTR [Temp+1],003h
        jne     No_87
        mov     al,001h
        jmp     Store
No_87:  xor     al,al
Store:  mov     [X_87],al
```

Hierzu folgende Anmerkungen:

- Die Labels heißen ein wenig anders: variatio delectat! Für alle, die in der Schule Französisch anstelle von Latein hatten: vive la difference!
- Sie vermissen FSTENV/FLDENV? Ich habe diese Befehlskombination hier tatsächlich nicht eingebaut, da es mir unsinnig erschien, in einem Programm, das ja sowieso gleich beendet wird, diesen ungeheuren (und hier wirklich überflüssigen) Aufwand zu treiben. Bauen Sie die Befehle zur Übung nachträglich ein. Überlegen Sie genau, wo und warum dort. Versuchen Sie hierbei, nicht auf das Listing des Programms CHECK_87 zu schauen. Wenn Sie glauben, die Lösung zu haben, so überprüfen Sie sie anhand dieses Listings.
- Nochmals die schon auf Seite 279 gestellte Frage: Warum wird *Temp* mit $FFFF vorbelegt? Ganz einfach! Wir wollen ja testen, ob der Befehl FINIT einen eventuell vorhandenen Coprozessor initialisiert. Kriterium für dessen Existenz ist, daß das Statuswort vollständig gelöscht wird, also 0 ist. Falls ein Coprozessor anwesend ist, erfolgt dies auch. Dies bedeutet, daß das anschließende *fnstsw [Temp]* diese 0 in *Temp* schreibt. Wenn nun aber kein Coprozessor sein karges Leben fristet, so macht weder FNINIT noch FNSTSW etwas, der Inhalt von *Temp* wird nicht verändert. Wenn nun jedoch zufällig dort eine 0 steht, weil *Temp* bei der Deklaration keinen Startwert erhalten hat, so muß die Routine zu dem fatalen Ergebnis kommen: »Coprozessor vorhanden!« (Was hier nicht weiter schlimm wäre: Wir prüfen ja im positiven Falle noch das Kontrollwort ab! Spätestens diese Prüfung würde den Fehler aufdecken, denn auch FNSTCW würde nichts tun und die in *Temp* stehende 0 nicht anrühren. Aber solche achtlosen »Unterlassungen« haben schon Generationen von Assembler-Programmierern schlaflose Nächte bereitet). Unzufrieden mit dem Startwert $FFFF? Dann nehmen Sie doch einen anderen! Jeden beliebigen – nur nicht »0«!
- Auf das Ergebnis der Tests wird anders reagiert. Während im letzten Programm aufgrund der durch die CMP-Befehle gesetzten Flags (de fatco des Zero-Flags) unterschiedliche Labels ange-

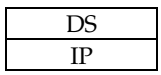
sprungen wurden, die unterschiedliche Meldungen auf dem Bildschirm generieren, so wird hier an den entsprechenden Stellen lediglich eine Variable mit unterschiedlichen Werten belegt. Falls nämlich ein Coprozessor entdeckt wurde, so wird AL mit 1 belegt, andernfalls mit 0 (Sie erinnern sich an *xor ah, ah*?). Dieser Inhalt wird nun mit dem MOV-Befehl in die Variable *X_87* kopiert. Das hat Konsequenzen! Denn einerseits kann nun an jeder Stelle des gesamten Programms festgestellt werden, daß ein (oder daß kein) Coprozessor vorhanden ist! Andererseits weiß nun die Routine *CHECK_87* selbst, daß sie nicht mehr aufgerufen werden muß! Der Startwert $C1 wurde ja nun in beiden Fällen überschrieben, und der vorher erläuterte Mechanismus verhindert einen neuen Coprozessor-Existenz-Test!

Wir haben erreicht, was wir wollen: Wir wissen nun, ob ein Coprozessor vorhanden ist. Also können wir die Routine beenden:

```
Ende:     pop     ds
          ret
Check_87 ENDP
```

Das Label `Ende` wird vom bedingten Sprungbefehl am Beginn der Routine angesprungen, wenn der Test schon erfolgt ist. Da wir aber die Routine mit dem Sichern des Inhalts des Datensegments begonnen haben, müssen wir diesen Inhalt mit `pop ds` nun wieder restaurieren.

Der Grund dafür ist, daß die Routine beendet werden soll, was bedeutet, daß ins aufrufende Programm zurückgesprungen werden soll. Zuständig hierfür ist der Befehl RET, der im Listing auch unmittelbar hinter `pop ds` folgt. RET aber macht nichts anderes, als die Rücksprungadresse vom Stack zu lesen und in IP oder CS:IP zu schreiben. Da die Routine als *near* deklariert wurde, ist in diesem Fall lediglich IP betroffen. Schauen wir uns also nochmals an, was auf dem Stack steht:



An oberster Stelle steht der Inhalt von DS, denn mit `push ds` haben wir ihn ja als erstes in der Routine dorthin plaziert. Wenn wir nun also DS *nicht* durch `pop ds` restaurierten, so geschähen fatale Dinge:

- RET holt einen Wert vom Stack und lädt ihn in IP. Da dies aber nicht der Offsetanteil der Rücksprungadresse ist, sondern der Segmentanteil des Datensegments, springt Der Prozessor an eine Stelle »zurück«, deren Offsetanteil den gleichen Wert hat wie der Segmentanteil des Datensegments. Und was steht dort? Ich weiß es nicht – und Sie? Wo

ist das überhaupt? Konsequenz: Der Prozessor führt die Befehle aus, die dort stehen. Ob das Befehle sind oder Daten, ob sie programmiert wurden oder nur zufällig bestimmte Inhalte haben, ob also das sinnvoll ist, was da steht, oder nicht, interessiert ihn überhaupt nicht. Er tut es einfach. Der Absturz ist vorprogrammiert!

- Damit nicht genug! Es könnte ja sein, daß dort zufällig sinnvoller Code steht, der dann ausgeführt wird. Dann muß es nicht unbedingt zum Absturz kommen. Zwar dürfte es sehr wahrscheinlich sein, daß das, was dann an Programm abläuft, nicht sehr sinnvoll ist, aber immerhin wäre der Zustand des Programms stabil. Doch Vorsicht: Falls mitten in eine Routine »zurück«gesprungen wurde, steht irgendwann einmal ein Rücksprung an, ohne daß irgend jemand eine Rücksprungadresse angegeben hätte. Nun können zwei Fälle eintreten: Einerseits kann die Routine mit einem *near ret* beendet werden. Dann holt sich der Prozessor den Rücksprungoffset vom Stack und lädt ihn in IP. Das wäre das beste, was Ihnen passieren könnte! Denn dort steht ja, nachdem der etwas verunglückte Rücksprung von eben den »DS-Wert« vom Stack genommen hat, tatsächlich eine korrekte Rücksprungadresse – wenn niemand in der Zwischenzeit den Stack verändert hat. Es handelt sich dabei um die Rücksprungadresse, die eigentlich hätte verwendet werden sollen. So endete die Odyssee nach mehr oder weniger kurzer Irrfahrt mit mehr oder weniger angerichtetem Schaden in Form unkontrolliert veränderter Variablen etc. genau dort, wo sie begann – ob wir aufatmen können, weiß niemand.

  Andererseits kann die Routine, in die wir unkontrolliert »zurück«gesprungen sind, auch mit einem *far ret* abgeschlossen sein. Dann holt dieser den Inhalt von CS:IP vom Stack. Den Prozessor interessiert überhaupt nicht, daß dort nur ein Offset steht! Gnadenlos kopiert er zwei weitere undefinierte Bytes in CS! Die Gefahr ist noch nicht gebannt.

- Treten all diese Fälle nicht ein, gibt es immer noch ein Problem: Es liegt noch ein Wert auf dem Stack, der erstens dort nicht hingehört, weil die aufrufende Routine den Stack anders übergeben hat, und den zweitens niemand wieder kontrolliert dort abholt.

Was auch immer geschehen sein mag – wir können nicht mehr behaupten, das Programm unter Kontrolle zu haben! Aber gerade als Assemblerprogrammierer müssen Sie diese Kontrolle in größerem Umfang haben als in Hochsprachen, die gewisse Fehler abfangen können! Als Quintessenz können wir also folgende Regeln festhalten:

ACHTUNG

Zu jedem PUSH gehört ein POP! Diese Regel ist allgemein gültig, auch wenn das POP in einem anderen Befehl oder Mechanismus versteckt ist (auf die wir auch noch zu sprechen kommen)! Ein Abweichen von dieser Regel hat garantiert fatale Folgen! Achten Sie immer

darauf, den Stack tatsächlich so zu hinterlassen, wie Sie ihn zu Beginn der Routine übergeben bekommen! Achten Sie auf die Reihenfolge, in der Werte auf den Stack gelegt werden! Entfernen Sie sie in genau der umgekehrten Reihenfolge wieder vom Stack, in der sie dort abgelegt wurden. Prüfen Sie, falls ein Programm in einer Routine abstürzt, zunächst, ob der RET-Befehl tatsächlich die korrekte Rücksprungadresse findet und benutzt.

TIP Zwingen Sie sich gerade am Anfang dazu, im Kopf nachzuvollziehen, was Ihr Quellcode an welcher Stelle tut. Benutzen Sie, wenn es sein muß, Papier und Bleistift, um sich den Stack aufzumalen und genau zu wissen, welcher Wert wo und wie steht! Ihre Programme werden es Ihnen durch ein Minimum an Fehlern danken!

Auch bei Routinen müssen wir noch ein wenig Bürokratie betreiben. Wir müssen nämlich dem Assembler mitteilen, daß die Routine, die wir mit *PROC NEAR* begonnen haben, beendet ist. Dies erfolgt durch die Anweisung **ENDP**rocedure, wobei der Name der Routine vorangestellt wird. »Reicht nicht das RET?« Nein! RET ist ein Prozessorbefehl, der überall stehen kann! Durch RET wird lediglich ein Rücksprung veranlaßt. Ob der RET-Befehl tatsächlich das Ende der Routine ist, ist nicht klar:

```
Routine1 PROC NEAR
         cmp   al,001h
         je    Ende1
         cmp   al,002h
         je    Ende2
         mov   ah,003h
         ret
Ende1:   mov   ah,002h
         ret
Ende2:   mov   ah,001h
         ret
; hier müßte das ENDP statement stehen!

Routine2 PROC NEAR
         :
         :
```

Es ist offensichtlich, daß das erste RET die Routine noch nicht beendet. Dennoch ist die Verwendung an dieser Stelle nicht nur legitim, sondern auch korrekt! Tatsächlich beendet wird in diesem Fall die Routine erst hinter dem RET von *Ende2*. Aber da auch wirklich, da *Routine2* beginnt! Somit muß in der Zeile mit dem Kommentar (eingeleitet durch das reservierte Zeichen »;«) die Anweisung stehen, die Definition der Routine abzuschließen.

22.6 Coprozessorunterscheidung

Wir sind bisher aber unserem Problem nur ein kleines Stück nähergekommen! Bisher wissen wir nur, ob ein Coprozessor vorhanden ist. Das aber konnte das Programm aus dem letzten Abschnitt auch feststellen. Machen wir also mit dem Schwierigsten weiter, der Feststellung, ob ein 80487 vorhanden ist. Nach dem, was wir uns eingangs überlegt hatten, ist diese Prüfung identisch mit einer Prüfung auf die Anwesenheit eines 8048**6**! Dies ist nicht trivial! Denn dazu müssen wir auf Eigenheiten des 80486 zurückgreifen, die sich in Registern abspielen, die zwar ein 80386 (und somit aufgrund der Abwärtskompatibilität auch ein 80486) hat, nicht aber die Prozessoren davor.

Also zunächst die Prüfung auf 80386, bevor wir die 32-Bit-Register benutzen können. Doch auch diese Prüfung ist nicht ganz trivial! Denn so wesentlich unterscheiden sich 80386 und 80286 im Real-Mode nicht voneinander, wenn man die Register nicht betrachtet, die wir beim 80286 nicht vorliegen haben. Nutzbare Unterschiede gibt es nur in der Behandlung einiger Flags. Für diesen Test müssen wir aber Prozessortypen vor dem 80286 ausschließen. Es ergibt sich also folgende Problemstellung:

- Prüfung, ob ein Prozessor vor 80286 vorliegt. Wenn ja, kann kein 80486 und somit kein 80487 existieren. Andernfalls
- Prüfung, ob ein 80386 eingebaut ist. Wenn nein, kann ebenfalls kein 80487 vorhanden ist.
- Falls doch, so existieren ab diesem Prozessor auch die 32-Bit-Register, und der nächste Test kann diese nutzen. Dieser bringt dann die Entscheidung, ob ein 80486 vorliegt.

Auch diese Prüfung führen wir in einer Routine durch. Wir könnten dies zwar auch im Hauptprogramm erledigen, es soll jedoch die Benutzung von Routinen betrachtet werden. Auch der 486-Check kann in einer Routine ausgeführt werden, die im gleichen Segment wie das Hauptprogramm liegt, weshalb wir sie *near* deklarieren:

```
Check_486 PROC NEAR
          push  ds
          push  cs
          pop   ds
          xor   ax,ax
          push  ax
          popf
```

Nach dieser Vorbereitung des DS-Registers analog zu der letzten Routine beginnt der erste Teil, Prüfung auf < 80286, damit, daß wir das Flagregister mit dem Inhalt 0 beladen, also alle Flags simultan löschen. Realisiert werden kann dies nur über den Umweg Stack, da der

MOV-Befehl ein Beschreiben des Flagregisters nicht erlaubt. Deshalb wird zunächst der Inhalt von AX gelöscht, dann das Register auf den Stack gepusht und in das Flagregister zurückgepoppt.

Doch wozu das Ganze? Das Flagregister aller Prozessoren bis 80286 ist 16 Bits breit, von denen aber nur sehr wenig benutzt werden. Die 8086/8088/80186/80188-Prozessoren haben die Eigenheit, daß einige Bits immer gesetzt sind. Das bedeutet also, daß sie, unmittelbar nachdem man sie (durch das Ablegen einer 0 im Flagregister) löscht, vom Prozessor wieder gesetzt werden. Das tun die Rechner ab dem 80286 nicht mehr. Also brauchen wir nur den Inhalt des Flagregisters wieder auszulesen, nachdem die 0 eingeschrieben wurde, und zu prüfen, ob die Bits gesetzt sind:

```
pushf
pop       ax
and       ax,0F000h
cmp       ax,0F000h
je        Nein
```

Also: Flaginhalt auf den Stack und zurück in AX, denn dort wirken die Vergleichsbefehle!

Es interessieren uns nur die Bits 15 bis 12 des Wortes, da nur diese Bits das beschriebene Verhalten zeigen. Daher blenden wir alle anderen aus (= setzen sie auf 0) und vergleichen das Resultat mit $F000. In dieser Konstanten sind ja ebenfalls die Bits 15 bis 12 gesetzt. Sind die Werte gleich, so setzt der CMP-Befehl das Zero-Flag und der bedingte Befehl JE kann zum Label `Nein` verzweigen.

Das Ausblenden der Bits ist einfach: Wir benutzen dazu den Befehl AND, der alle Bits zweier Operanden UND-verknüpft. Somit sind im Resultat nur die Bits gesetzt, die in beiden Operanden auch gesetzt sind. Und weil wir in der übergebenen Konstante die Bits 11 bis 0 gelöscht haben, sind sie es auch im Ergebnis.

Warum der »Umweg« über AND und CMP? Genügt nicht *test ax, 0F000h*? Antwort: nein! Die Begründung hierzu finden sie am Ende dieses Abschnitts. Versuchen Sie zunächst selbst, den Grund zu finden.

```
mov       ax,07000h
push      ax
popf
pushf
pop       ax
and       ax,07000h
jz Nein
```

Auch für die Prüfung auf das Vorliegen eines 80286 verwenden wir das Flagregister. Doch dieses Mal schreiben wir einen Wert hinein, der die Bits 12 bis 14 setzt. Wenn wir das Flagregister dann auslesen, so bleiben alle Bits gesetzt, wenn ein Prozessor ab dem 80386 vorliegt. Denn nur der 80286 löscht diese Flags explizit.

Also wieder die Sequenz »Konstante in AX – AX auf den Stack – Wert vom Stack ins Flagregister – von dort auf den Stack, zurück in AX und Test auf gesetzte Bits! Getestet wird beim AND-Befehl, dieser blendet nicht nur die uninteressanten Bits aus, er setzt auch die Flags anhand des Ergebnisses! Das aber heißt, daß das Zero-Flag gesetzt ist, wenn alle Bits, auch die nicht »maskierten«, gelöscht sind. Ist auch nur ein Bit gesetzt, ist das Zero-Flag gelöscht. Dies signalisiert uns und dem bedingten Sprungbefehl, daß ein 80386er vorliegt und dementsprechend ein Sprung erfolgen soll.

Nun aber können wir sicher sein, daß wir die Befehle einsetzen können, die beim 80286 und allen Prozessoren davor zu Problemen geführt hätten, weil es sie dort nicht gibt. **.386**

```
.386
```

Ohne eine Anweisung erzeugt jeder Assembler aus Kompatibilitätsgründen 8086-Code. Also müssen wir ihm mitteilen, daß er im folgenden die 80386-Erweiterungen verwenden soll. Dies tun wir über die Anweisung .386! Ab hier wird 80386-kompatibler Code erzeugt, der auf 80286ern und Vorgängern nicht mehr läuft.

Bei den Assembleranweisungen .386, .387 und .486 ist es nicht egal, wo sie im Quelltext stehen! So bewirken diese Anweisungen vor der Definition des Codesegments, daß das *Flat-Model* des 80386 verwendet wird. In diesem Fall erfolgen alle Adressierungen mit 32 Bit und sind nicht mehr mit den meisten Programmen kompatibel. Diese Anweisungen dürfen (und müssen dann auch!) nur dann am Anfang des Quelltextes stehen, wenn alle Programmteile, also auch Hochsprachenteile, in die die Assemblermodule eingebunden werden, ebenfalls das *Flat-Model* benutzen. Dies ist auch der Hintergrund dafür, daß der Linker eine Fehlermeldung erzeugt, wenn er (z.B. via *Assemble/Link/Link* in tpASM) den Quelltext linken soll, ihm aber (via *Options/Linker*) nicht die Erlaubnis zum Erzeugen von 32-Bit-Adressen gegeben wurde. **ACHTUNG**

Verwendet man dagegen die genannten Anweisungen nach der Definition des Codesegments, so wird zwar weiterhin die 16-Bit-Adressierung benutzt, die Erweiterungen der 80386er sind aber verfügbar.

Achten Sie grundsätzlich darauf, daß Sie, wenn möglich, Anweisungen, die nur auf bestimmte Programmteile wirken sollen, erst unmit- **TIP**

telbar davor geben und so bald wie möglich wieder rückgängig machen. Sie ersparen sich dadurch sehr viele Komplikationen.

Alignment

Beim Test auf den 80486 werden wir wiederum das Flagregister verwenden. Ab dem 80486 gibt es nämlich ein Flag, das das sogenannte *Alignment* steuert. Der theoretische Hintergrund ist, daß in bestimmten Programmierumgebungen die verwendeten Adressen *doppelwort*-weise ausgerichtet sein sollen (müssen). Dies aber heißt, daß die betreffende Stelle an Adressen stehen muß, die ohne Restbildung durch 4 teilbar sind. Falls dies in einem konkreten Fall nicht der Fall sein sollte, soll, so die Idee der Prozessorentwickler, der 80486 dies durch eine Ausnahmesituation und deren Behandlung korrigieren.

Diesen Mechanismus soll man aber auch ausschalten können. Ob dieser Mechanismus aktiv ist oder nicht, entscheidet demnach das oben genannte *Alignment-Check-Flag*. Beim 80486 ist dies veränderbar, während der 80386 diese Möglichkeiten nicht kennt und das Bit somit zurücksetzt, falls jemand daran herumspielen sollte. Dies führt wieder zu unserem nun schon wohlbekannten Schema »Konstante in AX – von dort auf den Stack – von dort ins Flagregister – von dort zurück auf den Stack und schließlich zur Auswertung in AX«. Einziger Unterschied: statt AX wird EAX verwendet und statt Flag EFlag!

Doch wir müssen noch etwas anderes berücksichtigen. Wenn wir bisher so ohne weiteres beliebige Werte in das Flagregister schreiben konnten, ohne uns Gedanken zu machen, können wir dies jetzt nicht mehr. Einige Flags im EFlagregister der 80386er und folgender Prozessoren haben nämlich eine bestimmte, genau definierte und situationsbedingte Bedeutung. Daher sollten wir tunlichst vermeiden, zu sehr an ihnen herumzuspielen. Was wir also brauchen, ist lediglich eine Konstante, bei der wir das *Alignment*-Check-Flag verändern können. Diese Konstante holen wir uns einfach aus dem EFlagregister. In diesem Wert sind ganz offensichtlich die Flags so gesetzt, wie sie stehen müssen, und das Zurückschreiben des nur am Alignment-Check-Flag manipulierten Wertes verursacht keine Probleme.

Aber wir müssen noch eine andere Vorsichtsmaßnahme ergreifen: Falls wir das Alignment-Check-Flag setzen, so wird der Prüfmechanismus ja eingeschaltet, und jeder Zugriff auf den Speicher, also auch das Pushen und Poppen des Stacks, wird geprüft. Um nun nicht in die Situation zu kommen, eine Exception auszulösen, müssen wir vor den Speicherzugriffen diesen noch ausrichten.

```
mov     ebx,esp
and     esp,0FFFCh
```

Dies ist schon der zuletzt angesprochene Schutz! Wir sichern uns den aktuellen Inhalt von ESP in EBX, da dieser Zeiger der Offset auf den Stack

ist, den wir nun ausrichten müssen. Dieses Ausrichten erfolgt einfach durch Löschen der Bits 0 und 1 in Form einer UND-Verknüpfung. Der resultierende Wert muß also eine durch 4 teilbare Adresse sein, die nun im ESP-Register steht. Der Stack ist damit auf Doppelworte ausgerichtet! Übrigens: auf den Stack sichern können wir den Inhalt von ESP vor dem Alignment nicht! Aber der Grund dafür dürfte wohl klar sein, oder?

```
pushfd
pop       eax
mov       ecx,eax
xor       eax,000040000h
```

An dieser Stelle holen wir (über den ausgerichteten Stack) den Inhalt des EFlagregisters in EAX und kopieren ihn (für später) in ECX. Dann kippen wir Bit 10 um! Hier müssen wir wieder etwas erklären. Wir wissen nicht, ob irgend jemand (meistens das Betriebssystem) das Alignment-Check-Flag gesetzt oder gelöscht hat. So könnte es, wenn ein Programm unter Windows läuft, sehr wohl gesetzt sein, während dies unter DOS unwahrscheinlich ist. Uns interessiert eigentlich auch gar nicht weiter, ob es gesetzt oder gelöscht ist. Was uns interessiert, ist, ob der Prozessor einen »falschen« Zustand korrigiert (80386) oder ihn als akzeptabel wertet (80486). Also drehen wir einfach das Bit um. War es gesetzt, so ist es nun gelöscht, war es gelöscht, so ist es nun gesetzt. Korrigiert ein 80386 hier etwas, so müssen sich die Werte in diesem Bit nach der Sequenz unterscheiden!

Das Kippen oder *Toggeln* der Flags, wie der Fachausdruck heißt, erfolgt durch eine exklusive ODER-Verknüpfung mit einer Maske, in der außer dem umzukippenden Bit alle gelöscht sind. Anschließend erhalten wir einen Zustand, in dem nur das betroffene Bit aus der Maske umgedreht ist.

```
push      eax
popfd
pushfd
pop       eax
```

Dies ist wieder die altbekannte Sequenz.

```
push      ecx
popfd
```

Da wir ja nun in EAX den neuen (korrigierten?) Wert haben, können wir getrost den alten (unmodifizierten) Zustand wiederherstellen. Hierzu speichern wir einfach die Kopie, die wir vor dem Bitkippen in ECX gesichert hatten, in EFlag. Fehlt nur noch die Feststellung, ob unsere Veränderung akzeptiert wurde.

```
xor       eax,ecx
and       eax,000040000h
mov       esp,ebx
jz        Nein
```

Auch hier hilft uns wieder XOR. Wenn ein 80486 installiert ist, so mußte er unser Kippen des Alignment-Check-Flags akzeptieren. Dann aber ist der in EAX stehende, neu geholte Wert von der Sicherungskopie in ECX verschieden! Denn diese Kopie ist ja der Wert vor dem Kippen! Der XOR-Befehl nun wird in diesem Fall alle Bits löschen, deren Zustand in EAX und ECX übereinstimmt. Und dies sind alle, außer dem Alignment-Check-Flag.

Hat dagegen ein 80386 unsere Eingabe nicht akzeptiert und korrigiert, so stimmt der neue Wert in EAX nicht mit dem überein, den wir konstruiert haben. Dann muß er aber identisch mit der Sicherungskopie in ECX sein! XOR löscht dann alle Bits! Der Rest ist einfach: eine UND-Verknüpfung mit einem Wert, der ein gesetztes Bit 10, also das Alignment-Check-Bit hat und sonst nur gelöschte Bits, setzt das Zero-Flag dann, wenn alle Bits im Testwert gelöscht sind, also bei Vorliegen eines 80386.

.8086

```
          .8086
          mov   ax,00004h
          jmp   Basta
Nein:     xor   ax,ax
Basta:    pop   ds
          ret
Check_486 ENDP
```

Um nun wieder 8086-kompatiblen Code zu erzeugen, der von jedem Prozessor verarbeitet werden kann, weisen wir mit `.8086` den Assembler an, nur noch 8086-Befehle zuzulassen.

22.7 Function or not function – that is a decision!

Wie kann man jedoch das Ergebnis dem Programm mitteilen? *Check_87* hat einfach eine Variable belegt, die von überall aus zugänglich ist. Die Routine *Check_486* legt einfach einen Wert »4« in AX ab, wenn ein 80486 geantwortet hat, ansonsten 0! Dann wird die Routine einfach beendet, und ihre Definition ebenfalls abgeschlossen.

Dennoch sollten wir dies nicht so hinnehmen! Es ist doch tatsächlich schon etwas sehr merkwürdig, daß die als letztes ausgeführten Befehle vor dem Rücksprung noch Werte in das AX-Register schreiben! Wozu und – wer hat etwas davon?

RET kopiert nur die Rücksprungadresse vom Stack in (CS:)IP. Veränderungen am Flagzustand oder an Inhalten der Allzweckregister erfolgen nicht! Daher steht der in AX befindliche Wert nach dem Rücksprung in das aufrufende Programm immer noch dort – und kann verwendet werden! Somit ist über die Inhalte von Prozessorregistern ein Informationsaustausch zwischen aufgerufenem und rufendem Programmteil möglich. Routinen, die aber dem rufenden Teil Ergebnisse übermitteln, heißen in Hochsprachen üblicherweise Funktionen! *Check_486* ist also eine Funktion.

HINWEIS

Der Assembler selbst macht keine Unterschiede zwischen Prozeduren (also Routinen, die kein Ergebnis zurückgeben) und Funktionen. Ob eine Routine eine Funktion ist oder nicht, entscheidet sich dadurch, wie mit den Prozessorinhalten verfahren wird, nachdem die Routine abgearbeitet wurde – also im rufenden Programmteil! Wird nämlich ein Registerinhalt irgendwie weiterverarbeitet (also entweder gespeichert oder für Entscheidungen verwendet oder modifiziert), so spricht man definitionsgemäß von Funktionen. Unterbleibt dies, liegen Prozeduren vor.

Ein Beispiel:

```
Routine1 PROC NEAR
         :
         :
         mov al,012h
         ret
Routine1 ENDP

Routine2 PROC NEAR
         call  Routine1
         ret
Routine2 ENDP

Start:   call Routine2
         cmp   al 012h
         :
         :
```

In diesem Listing definieren wir zwei Routinen. Es ist offensichtlich, daß *Routine1* eine Funktion ist (genauer: die Art, wie *Routine1* programmiert wurde, läßt darauf schließen, daß sie eine Funktion sein soll!). Sie übergibt dem rufenden Programm den Wert $12 in AL. *Routine2* dagegen scheint eine Prozedur zu sein. Denn sie übergibt dem rufenden Programmteil nichts.

Tatsächlich ist auch *Routine2* eine Funktion, da sie das Funktionsergebnis von *Routine1* an das rufende Programm weitergibt! Dieses

wird im Programm auch tatsächlich weiterverwendet! In Pascal würde man ein solches Konstrukt durch folgendes Listing erhalten:

```
function Routine1:byte;
begin
  Routine1 := $12;
end;

function Routine2:byte;
begin
  Routine2 := Routine1;
end;

begin
  EinByte := Routine2;
  :
```

Also ist *Check_87* von vorhin nur deshalb eine Prozedur, weil es keine Information direkt an das Programm übergibt! Das Ergebnis ihrer Aktivität wird in eine global verfügbare Variable eingetragen. Obwohl die gleiche Information auch noch in AL steht und dort bleibt (*pop ds* verändert AL nicht), macht, wie wir noch sehen werden, das rufende Programm keine Verwendung davon! Informationen aber, die nicht verwendet werden, sind keine Informationen.

Es ist jedoch einfach, aus *Check_87* eine Funktion zu machen! Wenn nämlich das rufende Programm die Information in AL speichern oder weiterverarbeiten würde, so hätten wir eine Funktion, obwohl sich an *Check_87* überhaupt nichts verändert hat!

Andererseits ist *Check_486* nur deshalb eine Funktion, weil der in AL stehende Wert vom rufenden Programm weiterverarbeitet wird.

Es ist bei der Datenübergabe vollkommen unerheblich, ob die Information, die dabei weiterverwendet wird, sinnvoll ist oder nicht:

```
Routine PROC NEAR
         mov   al,ch
         ret
Routine ENDP

Start:   call  Routine
         cmp   al,012h
         :
```

Die Routine kopiert den Inhalt von CH nach AL und wird dann beendet. In CH steht, so wie es oben gezeigt wird, etwas Undefiniertes, da weder in der Routine noch im Hauptprogramm jemals etwas mit CH

passiert. Dennoch ist die Routine eine Funktion, weil der Inhalt von AL im Hauptprogramm weiterverwendet wird und die Routine eben dieses Register verändert.

Nach diesem Exkurs über Prozeduren und Funktionen weiter mit unserem Projekt. Wir können nun feststellen, ob ein Coprozessor überhaupt vorhanden und ob der Prozessor ein 80486 ist.

Machen wir uns daher an die Prüfung auf andere Coprozessoren! Auch diese wollen wir in eine Routine einbinden.

```
Get_87 PROC NEAR
         push  ds
         push  cs
         pop   ds
         push  bx
         mov   al,[X_87]
         dec   al
         jnz   NichtOk
```

Wir nennen diese Routine *Get_87*. Auch sie befindet sich im gleichen Codesegment wie die restlichen Programmteile, so daß sie *near* definiert wird! Es ist guter Programmierstil, Register, die man in Routinen verändert, vorher zu sichern, um sie später zu restaurieren. Das geschieht mit DS und BX, wobei auch DS auf das Codesegment festgelegt wird. Dann erfolgt die erste echte Nutzung unseres Existenzbytes: Der Wert wird in AL kopiert und dekrementiert. Nun gibt es drei Möglichkeiten:

- In *X_87* steht noch der Startwert $C1. Die Dekrementierung liefert dann $C0 in AL, ein Wert, der nicht 0 ist und bei dem somit das Zero-Flag nicht gesetzt ist!
- *X_87* enthält 0. Dann wissen wir entsprechend der Definition von *Check_87*, daß diese Routine keinen Coprozessor festgestellt hat! Dekrementierung von AL liefert in diesem Fall $FF, was bedeutet, daß auch hier das Zero-Flag nicht gesetzt ist!
- In *X_87* steht »1«. Dann hat *Check_87* einen Coprozessor gefunden. Dekrementierung um 1 liefert endlich 0 und ein gesetztes Zero-Flag.

Wir können also den bedingten Sprungbefehl JNZ (bzw. JNE) verwenden, um in den ersten beiden Fällen zu einem Label zu springen, das den Programmteil spezifiziert, der bei Nichtexistenz eines Coprozessors abgearbeitet werden soll. Ansonsten fahren wir fort:

```
call     Check_486
or       ax,ax
jnz      Ok
```

Die Routine ruft jetzt selbst eine Routine auf, und zwar die Funktion *Check_486*. Diese liefert ja 0 zurück, wenn kein 80486 gefunden wurde, ansonsten den Code 4. Also kann *Get_87* dann seine Tätigkeit einstellen, wenn 4 gefunden wird. Der Test hierauf sieht mysteriös aus, ist aber sehr effizient und kurz! Die ODER-Verknüpfung des AX-Registers mit sich selbst ändert an den Bits nichts! Aber wir wissen, daß die logischen Verknüpfungen nach ihrer Ausführung die Flags anhand des Ergebnisses setzen. Wenn nun 0 in AX stand, so ist das Ergebnis der ODER-Verknüpfung auch 0 und damit das Zero-Flag gesetzt. Andernfalls eben nicht! Es wird hier also de facto nicht geprüft, ob das Funktionsergebnis 4 ist! Vielmehr wird auf 0 getestet – einen von zwei möglichen Funktionswerten. Somit kommen wir zur gleichen Erkenntnis: Ein gelöschtes Zero-Flag ist gleichbedeutend mit »Coprozessor gefunden«. Konsequenz: Sprung zum Label *Ok* – wir sind fertig!

Die Prüfung auf einen 80387 ist dank der grundsätzlichen Unterscheidung von positiven und negativen Unendlichkeiten einfach. Dazu erzeugen wir eine Unendlichkeit durch Division einer Zahl durch 0, negieren eine Kopie davon und vergleichen die beiden Zahlen miteinander. Unterscheiden sie sich, ist es ein 80387:

```
mov     ax,00003h
fld1
fldz
fdivp   st(1),st
fld     st(0)
fchs
fcompp
fstsw   [Temp]
fwait
test    [Temp],00100h
jnz     OK
```

Da der 80486 den Code 4 in AX übergibt, verwenden wir für den 80387 den Code 3. Diesen legen wir sofort ins AX-Register, weil dieses für die 80387-Prüfung nicht benötigt wird. Wir können dann im positiven Fall gleich die Prüfung beenden. Dann verwenden wir FLD1, um die Konstante *1.0* in den TOS des Coprozessorstapels zu legen.

Als nächstes laden wir die Konstante *0.0* mit FLDZ. Dieser Befehl »schiebt« vorher den derzeitigen Inhalt des TOS in Register 1. Danach steht also in ST(1) 1.0, im TOS 0.0. Nun tun wir das, was beim Prozessor zu einem Interrupt führte: Division des ST(1) (also 1.0) durch TOS (also 0.0). Das Ergebnis ist eine spezielle NaN, die für +∞ steht. Da wir den Stack gleichzeitig poppen, steht dieser Wert dann im TOS. Nun erzeugen wir eine Kopie dieser NaN, indem wir einfach den Inhalt des TOS erneut laden. Jetzt haben wir im TOS und in ST(1) die gleiche (positive) NaN. Die

NaN im TOS negieren wir anschließend. Der Vergleich von TOS und ST(1) setzt nun den Condition Code im Statuswort des Coprozessors. Das kennen Sie ja! Dieses Statuswort können wir nicht direkt prüfen: Wir müssen es über den Umweg einer Variablen auswerten. Also wird es in einer Variablen namens *Temp* gespeichert.

Da dieser Speichervorgang eine Weile Zeit kosten kann, müssen wir an dieser Stelle den Synchronisationsbefehl FWAIT einfügen, damit der Prozessor auf *Temp* erst dann zugreifen kann, wenn der Coprozessor mit der Speicherung fertig ist. Falls dann im Wert in *Temp* Bit 8 gesetzt ist, wird zwischen positiver und negativer Unendlichkeit unterschieden. Ob dieses Bit gesetzt ist, stellt TEST mit einer Maske fest. TEST macht ja eine Pseudo-UND-Verknüpfung, nach der zwar die Flags wie nach AND gesetzt sind, die Operanden aber nicht verändert werden. Ist das Bit gesetzt, so können wir an das Ende-Label springen: der Coprozessor-Code steht ja schon in AX.

HINWEIS

Übrigens ist Bit 8 des *Statusworts* identisch mit C0 des Condition Codes! C0 ist nach Vergleichen dann gesetzt, wenn entweder die Operanden nicht vergleichbar sind (dann ist auch C3 gesetzt) oder wenn Operand 1 < Operand 2 ist (C3 ist dann 0). Den ersten Fall können wir hier ausschließen, weil zwar mit »merkwürdigen« Zahlen in Form spezieller NaNs gerechnet wird (Unendlichkeiten), dies allerdings ganz legitim und nach den Regeln der (Programmier-)Kunst erfolgt. Also brauchen wir (hier!) tatsächlich nur Bit 8 zu testen. Besser und eindeutiger jedoch wäre eine Verzweigung über *mov ax, [Temp]; sahf; jb Ok*. Doch ich wollte hier einmal demonstrieren, daß dies nicht der einzig gangbare Weg ist.

Bleibt noch die Unterscheidung zwischen 80287 und 8087! Gehen wir zunächst davon aus, daß ein 80287 vorhanden ist, und legen wir daher in AL den Code 2 ab. Da vom vorherigen Test noch der Code 3 dort steht, brauchen wir den Inhalt von AL nur um 1 zu verringern:

```
        dec   al
        fld1
Hic:    fistp cs:[Temp]
        fstenv [Env]
```

Wie wir weiter oben schon festgestellt hatten, unterscheiden sich ein 80287 und ein 8087 in sehr wenigen Punkten! Einer davon ist, welche Adresse als *Instruction-Pointer* durch den Befehl FSTENV gespeichert wird. Im Referenzteil des Buches können Sie nachschlagen (bei FLDENV), daß durch diesen Befehl der Inhalt des *Instruction-Pointer-Register* als Byte 6 und 7 zusammen mit den Inhalten der Coprozessorregister gespeichert wird. Der 80287 speichert hier als Adresse die Adresse des Befehls vor dem FSTENV-Befehl, während der 8087 die Adresse von FSTENV selbst sichert!

Also müssen wir den Inhalt von Byte 6 und 7 der Variablen, in die wir während des Tests das Environment des Coprozessors speichern, nur mit der Adresse des Befehls vor FSTENV vergleichen. Doch wie kommen wir an letztere heran? Nichts leichter als das! Denn wie schon mehrfach erwähnt, sind Labels nichts anderes als Zeiger auf den Befehl, der hinter dem Label steht. So lassen wir den Befehl vor FSTENV einem Label, *hic*, folgen.

Wir müssen daher nur berücksichtigen, daß FSTENV nicht der erste Coprozessorbefehl sein darf, da beim 80287 ja die Adresse des Befehls vor FSTENV gespeichert wird! Also programmieren wir etwas Belangloses: Wir laden 1.0 auf den TOS und speichern diesen Wert in einer Variablen, die wir eigentlich gar nicht brauchen. *Temp* leistet hier hervorragende Dienste, da wir es im Moment nicht zur Speicherung wesentlicher Argumente benötigen!

Doch Achtung! *Temp* ist eine Wortvariable, weshalb der Speicherbefehl auch nur auf Wortvariablen zugreifen darf. Genau das tut FIST. Da wir den im TOS stehenden Wert aber nicht mehr brauchen und den Stack (hier wirklich den Coprozessorstack!) so hinterlassen wollen, wie wir ihn vorgefunden haben, benutzen wir die Variante, die den TOS auch gleich wieder poppt: FISTP.

Nun können wir FSTENV ausführen.

```
mov     bx,cs
shl     bx,1
shl     bx,1
shl     bx,1
shl     bx,1
add     bx,OFFSET Hic
```

Anschließend berechnen wir die Adresse von *hic*. Wir wissen, daß die vollständige Adresse eines Bytes im Speicher aus einem Segmentanteil und einem Offset besteht. Wir wissen aus Teil 1 des Buches auch, daß sich die physikalische Adresse aus Segment · 16 + Offset berechnet. Den Segmentanteil kennen wir. Es ist ja das Codesegment, dessen Adresse in CS steht. Den Offset erhalten wir durch die Assembleranweisung OFFSET.

Mit dem MOV-Befehl holen wir uns eine Kopie des CS-Inhalts ins BX-Register und multiplizieren diesen Wert mit 16. Dies erfolgt hier über vier Schiebebefehle, da eine Verschiebung einer binär codierten Zahl um 1 Bit nach links gleichbedeutend mit einer Multiplikation mit 2 ist. Somit sind vier aufeinanderfolgende Verschiebungen um 1 Bit nach links identisch mit einer Multiplikation von 2·2·2·2=16! Dies geht erheblich schneller als über MUL!

Merken Sie sich diesen Trick: Wenn Sie mit Potenzen von 2 multiplizieren oder durch sie dividieren, benutzen Sie am besten die Bitschiebebefehle SHL und SHR! **TIP**

Addieren wir zum Ergebnis noch den OFFSET von *hic* – und schon haben wir die Adresse des Befehls vor FSTENV, also von FISTP ...

```
inc     bx
```

... denkt man sich! Irrtum! Woran Sie immer denken müssen, wenn Sie solch trickreiche Programme erstellen, ist, daß der Assembler eventuell eigene Ansichten über die Bytefolge des Assemblats hat!

Denken Sie daher daran, daß der Assembler zwecks Synchronisation des Coprozessors mit dem Prozessor FWAITs vor jedem Coprozessorbefehl einstreut! Das bedeutet, daß Sie den Quelltext jeweils um ein automatisch eingefügtes FWAIT ergänzen müssen! **ACHTUNG**

Hic deutet somit nicht auf den FISTP-Befehl selbst, sondern auf das durch den Assembler eingesetzte FWAIT, das zum FISTP-Befehl gehört. FSTENV aber richtet sich nur nach den eigentlichen Coprozessorbefehlen, weshalb die Adresse von FISPT selbst dort steht (FWAIT ist eigentlich gar kein Coprozessorbefehl, sondern der Prozessorbefehl WAIT. Da aber FSTENV nur die Adresse von Coprozessorbefehlen sichert, ist WAIT außen vor)!

Das ist jedoch nur der Fall, wenn der Assembler nicht durch eine der Assembleranweisungen *.287*, *.387* usw. andere Anweisungen erhält! Da nämlich ab dem 80286/80287 die Synchronisation anders abläuft, braucht der Assembler keine FWAITs einzustreuen, wenn sichergestellt ist, daß nur 80287-Befehle zur Geltung kommen. Dies erfolgt über die genannten Anweisungen! **ACHTUNG**

Falls Sie trickreich programmieren, sollten Sie deshalb grundsätzlich genau das Assemblat studieren, das der Assembler erzeugt. Achten Sie darauf, in welchem Modus (mit welchen Anweisungen) der Assembler seine Aufgaben erledigt, und denken Sie daran, daß alles ein wenig anders sein kann als erwartet, wenn Sie vom Pfad der 8086/8087 – Kompatibilität abweichen! **HINWEIS**

Doch zurück zum Listing. Mit *inc bx* wird also, da das FWAIT (bzw. WAIT!) ein Ein-Byte-Befehl ist, die Adresse um 1 erhöht. Sie zeigt nun, weil wir eben keine *.287*-Anweisung angegeben haben, korrekt auf FISPT. Aber Achtung: BX ist ein 16-Bit-Register, und die Multiplikation eines 16-Bit-Segments mit 16 überschreitet sicherlich die Kapazität dieses Registers.

Es entsteht also ein Überlauf, und zwar vielleicht schon beim ersten SHL! Eigentlich müßten wir, wenn wir korrekt wären, ein Registerpaar benutzen und eine vollständige 20-Bit-Adresse konstruieren.

Dies würde aber den Code erheblich verlängern und verkomplizieren. Wir vertrauen daher einfach darauf, daß 16 Bits (und zwar die niederwertigen) ausreichen, um auf unterschiedliche Adressen zu prüfen. Es ist unwahrscheinlich, daß sich ausgerechnet in den verworfenen vier Bits ein Unterschied zeigt, der uns auf diese Weise entgeht.

Aber sauber programmiert ist das nicht! Man sollte solche Unterlassungen auch wirklich nur dann anwenden, wenn man damit keinerlei Gefahr läuft. Dies ist hier gegeben: Sollte der Test wegen unserer Unterlassung »falsch negativ« verlaufen, so führt das nur dazu, daß ein 8087 angenommen wird, wo ein 80287 seinen Dienst tut. Damit liegen wir aber auf der sicheren Seite: Denn ein 80287 ist abwärtskompatibel. Anders herum sähe es schon schlechter aus, und in diesem Fall sollte man dann den höheren Programmieraufwand betreiben!

```
fwait
cmp       bx,[Env+6]
je        Ok
```

Der Rest ist trivial! Nun folgen die Synchronisation von Prozessor und Coprozessor, da ersterer die Adreßberechnung eventuell schneller durchführen konnte als letzterer, das Sichern des Environments in *Env* und der Vergleich der in BX berechneten Adresse mit dem Wort, das an Byte 6 in *Env* beginnt. Sind beide gleich, so liegt ein 80287 vor, und wir können zum Ende springen, weil der Code 2 schon in AL liegt.

```
          xor   al,al
OK:     mov     [T_87],al
NichtOk:pop     bx
          pop   ds
          ret
Get_87 ENDP
```

Andernfalls löschen wir AL, um mit Code 0 zu signalisieren, daß ein 8087 vorhanden ist. Der Code, egal wo er erzeugt wurde, wird nun am Label *Ok*, an dem alles wieder zusammenläuft, gesichert, und BX und DS werden wieder mit der Sicherungskopie auf dem Stack restauriert.

ACHTUNG Restaurieren Sie immer in umgekehrter Reihenfolge, wie Sie auf dem Stack sichern! Der Stack ist, wie schon gesagt, eine *LIFO*-Struktur, aus der Sie immer das zuletzt abgelegte Datum zuerst holen!

Funktion oder Prozedur – das ist hier die Frage! Da der Code des Coprozessors in der Routine in eine global verfügbare Variable gespeichert wurde, kann man fast davon ausgehen, daß *Check_87* eine Prozedur ist. Wir werden im Hauptprogramm sehen, ob dies stimmt!

22.8 Kombination der Routinen zum Programm

Nun können wir die einzelnen Routinen zu einem Gesamtprogramm zusammenbauen. Im Prinzip erfolgt dies genau so wie in Hochsprachen auch:

```
Start:   push  cs
         pop   ds
```

Wir beginnen wie üblich mit dem Label *Start*. Zuerst wird ASSUME berücksichtigt.

```
call     Check_87
or       al,al
jz       NixDa
```

Es folgt die Prüfung, ob überhaupt ein Coprozessor vorhanden ist. Dies erledigt die Prozedur *Check_87*, die wir mit dem CALL-Befehl aufrufen.

Wenn Sie aber den obigen Code betrachten und sich *Check_87* ins Gedächtnis zurückrufen, so sollte Ihnen etwas auffallen! Denn wir haben *Check_87* als Prozedur implementiert, die das Testergebnis in die Variable *X_87* schreibt. Und was machen wir nach dem CALL? Statt zu prüfen, ob die Konstante *X_87* die Existenz eines Coprozessors anzeigt, testen wir AL! Das bedeutet ja nach unserer Definition, daß *Check_87* eine Funktion ist.

Sie sehen: ohne weitere Vereinbarung, die in Hochsprachen tatsächlich erfolgt, ist die Grenze zwischen Funktion und Prozedur mehr als fließend! Denn da in AL noch der gleiche Wert wie in *X_87* steht, können wir uns ein erneutes Laden mittels *mov al, [X_87]* sparen.

Der sich anschließende OR-Befehl prüft nun, ob der übergebene Wert 0 ist oder nicht. Ist er das, so ist nach *or al, al* das Zero-Flag gesetzt, andernfalls nicht. Wenn es gesetzt ist, heißt das, daß kein Coprozessor vorhanden ist, weshalb an ein Label verzweigt wird, das genau diese Meldung ausgibt und dann das Programm beendet.

Kann jedoch ein Coprozessor festgestellt werden, so wird eine Meldung ausgegeben, die dies bestätigt und gleichzeitig die sprachliche Voraussetzung für die dann folgende Typausgabe ist:

```
mov      dx,OFFSET Msg2
mov      ah,009h
int      021h
```

Es folgt der Aufruf der Funktion, die den Typ feststellt und genau dazu geschaffen wurde:

```
call     Get_87
```

Wie Sie sich sicherlich erinnern werden, gibt die Funktion einen Code zwischen 0 und 4 zurück, der für den Coprozessortyp steht. Funktion und Hauptprogramm haben sich darauf geeinigt, AX zu diesem Zwecke zu benutzen, weshalb das Hauptprogramm auch genau hier das Funktionsergebnis erwartet.

Doch wie kann nun der Code verwendet werden, um unterschiedlichen Text auszugeben? Hier stellen wir ein Verfahren vor, das in der Assemblerprogrammierung sehr effektiv, kurz und beliebt ist: Offset-Tabellen!

Stellen Sie sich vor, Sie wollen drei Texte ausgeben können:

1. Dies ist Text 1.
2. Und dies Text 2.
3. Schließlich Text 3!

In Assembler würde der Text wie üblich mit Hilfe von DBs definiert:

```
DB 'Dies ist Text 1.'
DB 'Und dies Text 2.'
DB 'Schließlich Text3!'
```

Wir wissen mittlerweile auch, daß wir jedem Text ein Label zuordnen können, das auf den Anfang des Textes zeigt:

```
Text1     DB 'Dies ist Text 1.'
Text2     DB 'Und dies Text 2.'
Text3     DB 'Schließlich Text3!'
```

Hiermit ersparen wir uns das mühselige Abzählen von Bytes! Was unsere Textausgaberoutine braucht, ist der Offset, an dem der jeweilige Text beginnt, also praktisch den Offset des Labels. Nun soll der Text aber anhand einer Codezahl ausgegeben werden, also muß der Routine als Parameter

- der Offset von Text 1 übergeben werden, wenn der Code 1 ist,
- der Offset von Text 2 bei Code 2 und
- der Offset von Text 3 bei Code 3.

Fassen wir doch einmal die Offsets in einer Tabelle zusammen:

```
OffsetTable     DW OFFSET Text1
                DW OFFSET Text2
                DW OFFSET Text3
```

Nun steht also an der Stelle, auf die das Label *OffsetTable* zeigt, ein Wort mit dem Offset von *Text1*, zwei Bytes später der Offset von *Text 2* und wiederum zwei Bytes später der von *Text3*.

Wenn wir zum Offset der Tabelle nun code-mal 2 addieren, so steht an dieser Stelle der gewünschte Offset des Textes. Anders gesagt: wenn wir den von *Get_87* bestimmten Code mit 2 multiplizieren, weil die Offsets

immer nur zwei Bytes lang sind, dann müssen wir diesen Tabellenindex nur zum Beginn der Tabelle addieren, den an dieser Stelle stehenden Wert auslesen und unserer Textausgaberoutine übergeben.

TIP

Machen Sie ausgiebig von Offset-Tabellen Gebrauch, wenn Sie auf unterschiedliche Adressen in Abhängigkeit von berechneten Werten zugreifen müssen. Diese Tabellen sind pflegeleicht, sehr flexibel, leicht lesbar und äußerst effizient!

Der Grund, weshalb diese Methode so flexibel und gleichzeitig pflegeleicht ist, liegt darin, daß man nachträglich noch Änderungen am Text vornehmen kann, ohne sich weiter um den Mechanismus zu kümmern! Wollte man z.B. den Text »80487« beim Vorliegen eines 80487 in »Pseudo-80487, da es nur einen 80486 gibt!« ändern, so bräuchte man tatsächlich nur diesen Text zu ändern. Da diese Änderung im Quelltext erfolgen muß und dieser somit neu assembliert wird, berechnet der Assembler auch neue Adressen und trägt sie in die Offset-Tabelle ein.

Konkret sieht das so aus:

```
mov     bx,ax
shl     bx,1
mov     dx,[OffsetTable+bx]
mov     ah,009h
int     021h
```

Den in AX übergebenen Code kopieren wir in BX, da nur dieses Register verwendet werden kann, um als Indexregister für indirekte Adressierungen zu dienen. Es ginge auch mit anderen Registern, dann wäre die Berechnung der korrekten Adresse aber aufwendiger und sehr viel weniger elegant!

Der SHL-Befehl multipliziert nun diesen Code mit 2 und erzeugt somit den Tabellenindex. Dieser in BX stehende Index wird nun zum Offset von *OffsetTable* (der Tabelle mit den Startadressen der Texte) addiert und als indirekter Spreicherzugriff via MOV verwendet. Das bedeutet, daß der Wert, der an *OffsetTable* + *BX* steht, in DX geladen wird.

Dort erwartet ihn die DOS-Routine, die wir schon kennen. Man muß also nur noch den DOS-Code 9 für die Textausgabe in AH und den DOS-Interrupt aufrufen.

Um unser Programm zu vervollständigen, wäre es eigentlich sehr schön, wenn wir auch die Arbeitsgeschwindigkeit des Coprozessors feststellen könnten. Auch dies werden wir realisieren. Aber, um im Fluß zu bleiben: Nehmen wir zunächst an, wir hätten eine Funktion, die das kann, schon realisiert und rufen sie daher einfach auf. Die Funktion selbst besprechen wird dann im Anschluß an das Hauptprogramm.

```
call    Get_Speed
mov     bl,10
div     bl
or      ax,03030h
mov     [Hierhin],ax
mov     dx,OFFSET SoSchnell
jmp     Schluss
```

Auch diese Funktion gibt ihren Funktionswert in AX zurück, jedoch als Hexadezimalzahl, die erst noch in Zeichen umgewandelt werden muß, die darstellbar sind (vgl. hierzu auch den Anhang). Also müssen wir die-Hexadezimalzahl in zwei ASCII-Zeichen überführen. Wir hoffen nun, daß die Auslieferung des 100-MHz-80486 oder -Pentiums noch ein klein wenig dauert. Denn in der hier realisierten Form können maximal zwei ASCII-Zeichen erzeugt werden, also Taktfrequenzen bis maximal 99 MHz dargestellt werden. Aber vielleicht ist das ja für Sie eine Herausforderung ... ?

Um aus der Hexadezimalzahl XY zwei ASCII-Ziffern zu machen, müssen wir die Ziffern erst einmal trennen. Dies erfolgt hier durch den DIV-Befehl. Dieser erzeugt ja in AL das Ergebnis einer Division und in AH ihren Rest. Wird XY in AL also durch 10 dividiert, dann steht in AL X und in AH Y! Nun ist die Sache einfach: Da die ASCII-Ziffern alle ab dem Zeichen $30 für »0« beginnen, brauchen wir zu AH und AL nur $30 addieren. Dies ist diesmal besonders einfach, da eine Addition von jeweils $30 durch Setzen der Bits 4 und 5 sowie 12 und 13 des Wertes in AX erfolgen kann: über *or ax, 03030h*.

Nun haben wir zwar zwei ASCII-Zeichen, aber wir wollen sie ja in Text eingebettet ausgeben. Dazu haben wir den Text, in den die erzeugten Zeichen eingebaut werden sollen, so präpariert, daß wir einfach an die gewünschte Stelle gelangen können. Der auszugebende Text besteht aus drei Teilen: dem Teil vor der berechneten Taktrate, der Taktrate und dem Teil danach.

Teil 1 beginnt am Label *SoSchnell*. Dieses Label benötigen wir für die DOS-Textausgaberoutine. Die Stelle, an der die ASCII-Zeichen stehen sollen, nennen wir *Hierhin*! Dort sind zwei Bytes Platz reserviert, in die sie passen. Daran schließt sich dann der Rest an. Also brauchen wir nur noch den Inhalt von AX an die Stelle *Hierhin* zu schreiben, was wir oben mit dem MOV-Befehl auch tun!

Doch auch hier wollen wir noch einmal nachdenken! Der DIV-Befehl hat uns in AL X und in AH Y geschrieben. In AX steht also das ASCII-Zeichen von Y »über« dem von X. Führt das nicht dazu, daß z.B. bei einer Taktrate von 20 MHz im Text »Geschwindigkeit: 02 MHz erscheint? Nein!

Sie dürfen bei solchen Manipulationen niemals vergessen:

ACHTUNG

Laut Intel-Konvention wird ein Wert beim Schreiben in den Speicher immer in der Reihenfolge niederwertiger Anteil gefolgt von höherwertigem Anteil geschrieben!

Zwar steht in AX »Y« vor »X«, *Hierhin* erhält diese Bytes aber in umgekehrter Reihenfolge, womit alles wieder seine Ordnung hat. Übrigens ist dies auch der Grund, warum wir nicht den Befehl AAM zur Trennung der Ziffern verwendet haben. Dieser Befehl tut das gleiche wie DIV und wäre, da er schneller ist, vorzuziehen. Er schreibt jedoch die »höhere« Ziffer in AH und die »niedrigere« in AL. Der sich anschließende MOV-Befehl würde nun tatsächlich die Reihenfolge verkehren! Probieren Sie es aus.

Nur der Vollständigkeit halber schließt sich nun das weiterhin unkommentierte Ende unseres Programms an. Sie finden das Programm unter dem Namen GET_87 auf der beiliegenden CD-ROM.

```
NixDa:    mov    dx,OFFSET Msg1
Schluss:mov      ah,009h
          int    021h
          mov    ah,04Ch
          Int    021h

CODE ENDS

END Start
```

22.9 Die Geschwindigkeit des Coprozessors

Doch nun zu *Get_Speed*! Könnten wir nicht beim Ermitteln des Coprozessortyps gleichzeitig auch prüfen, mit welcher Geschwindigkeit dieser arbeitet? Wir kommen hier zu einem etwas komplexen Thema.

Der Hintergrund ist einfach. Wenn wir die Geschwindigkeit messen wollen, mit der der Coprozessor arbeitet, so müssen wir ihn einen definierten Befehl ausführen lassen und die Zeit messen, die währenddessen vergeht. Die offensichtlichste Möglichkeit ist, den DOS-Systemtakt hierfür zu nutzen, indem wir die Anzahl an DOS-Ticks messen, die während der Ausführung des Befehls vergehen.

Doch der DOS-Systemtakt ist mit einer Frequenz von 18,2 Hz um Dimensionen langsamer als der am längsten dauernde Coprozessorbefehl! Stellen Sie sich z.B. den Befehl FYL2X vor. Dieser Befehl braucht im Durchschnitt beim 8087/80287 1000 Takte Ausführungszeit, beim 80387 noch 330 und beim 80487 immerhin noch 265 Takte, wie Sie Teil 3 entnehmen können. Selbst bei einem alten 8086 mit Coprozessor und der sagenhaften

Taktrate von 4,66 MHz heißt das, daß der Befehl innerhalb von 215 µsec abgearbeitet wird. Der DOS-Ticker aber kommt alle 54,9 *m*sec! Das bedeutet, daß in einen DOS-Ticker 256 FYL2X-Befehle »passen« (Werte für einen 66-MHz-487er: 4,02 µsec Ausführungszeit entsprechend 13.700 FYL2X-Ausführungen!). Wir brauchen also eine andere Stoppuhr. Die gibt es: den Timerbaustein, der in jedem Computer vorhanden ist und für solche Dinge wie Tonerzeugung, Refresh der Speicherbausteine und andere Dinge zuständig ist. Ohne zu tief in die Timer- und Hardware-Programmierung einsteigen zu wollen (es gibt sehr gute Bücher zu dem Thema), soll hier nur kurz demonstriert werden, daß dies mit Assembler nicht schwierig ist. Wir werden in der folgenden Routine den Timerbaustein direkt über seine Ports ansprechen, ihn programmieren und auch Daten von ihm holen.

```
Get_Speed PROC NEAR
          push  ds
          push  cs
          pop   ds
          push  bx
          xor   ax,ax
          cmp   [X_87],0FFh
          jne   Da
          jmp   Aus
```

Den Grund für das einführende *push bx* kennen wir schon: Wir werden BX benötigen und »retten« daher dessen Inhalt. Dann löschen wir den Inhalt von AX, das wir zur Funktionswertübergabe verwenden, und geben somit ein »Default«- oder Standardergebnis vor. Dann prüfen wir, ob die Existenz eines Coprozessors schon bekannt ist. Falls also in *X_87* der beim Programmstart vorgegebene Wert $FF steht, so wurde noch kein Coprozessor festgestellt und die Routine wird beendet.

Hier zeigt sich auch schon ein Haken, der vor allem dem Anfänger sehr viele Kopfschmerzen macht: Die bedingten Befehle haben nur eine »Reichweite« von 127 Bytes, das heißt, daß das Sprungziel maximal 127 Bytes weit vom augenblicklichen Standort entfernt sein darf. Das ist hier der Fall. Somit meckert der Assembler, falls Sie oben anstelle des JNE-Befehls den eigentlich richtigen JE-Befehl einsetzen. Denn an das Ende der Routine soll ja gesprungen werden, wenn der Inhalt von *X_87* gleich $FF ist.

Deshalb verwenden wir hier den gegenteiligen Befehl, der unmittelbar hinter einen unbedingten Sprungbefehl zielt. Letzterer kann nämlich so weit springen, wie Sie wollen – nötigenfalls sogar intersegmentiell! Also: wenn der Inhalt von *X_87* nicht gleich $FF ist, springen wir hinter einen JMP-Befehl, der für den anderen Fall an das Routinenende verzweigt.

```
Da:     cmp   [T_87],0FFh
        jne   Los
        call  Get_87
```

Dort wird dann geprüft, ob der Typ schon feststeht, und, wenn nicht, durch Aufruf von *Get_87* genau dies nachgeholt. Da wir damit rechnen müssen, daß wir sehr schnelle Coprozessoren vorliegen haben, müssen wir mehrere Befehle hintereinander schalten, da selbst für unseren »schnellen« Timerbaustein, den wir für die Zeitmessung verwenden, 4 µsec bei 66-MHz-487ern sehr kurz sind. Somit werden wir diesen Befehl – es ist der mit der längsten Ausführungszeit – siebenmal hintereinander ausführen. Daher brauchen wir acht Startwerte, weil durch FYL2X ein Register gepoppt wird!

Was wir in die acht Register schreiben, ist eigentlich egal, solange der Wertebereich für FYL2X nicht überschritten wird. Hierfür kann die Konstante π verwendet werden. Also belegen wir alle acht Coprozessorregister mit dieser Konstanten vor:

```
Los:    fldpi
        fldpi
        fldpi
        fldpi
        fldpi
        fldpi
        fldpi
        fldpi
```

Jeder FLDPI-Befehl pusht die Coprozessorregister, bevor die Konstante in den TOS geschrieben wird. Somit sind nun alle acht Register belegt.

Nun müssen wir den Timerbaustein programmieren. Hierbei darf uns niemand stören, denn wir wollen die Zeit messen, die die FYL2X-Befehle benötigen und nicht die durch eventuelle Interrupts verfälschte Zeit. Also sperren wir die Interrupts mit CLI. Anschließend wählen wir den Timer 0 und programmieren ihn für den Modus 2 mit dem Startwert 0.

```
        cli
        mov   al,034h
        out   043h,al
        jmp   L1
L1:     jmp   L2
L2:     xor   al,al
        out   040h,al
        jmp   L3
L3:     jmp   L4
L4:     out   040h,al
```

Die OUT-Befehle dienen dazu, Werte an den Timerbaustein auszugeben. Die Konstante $43 im ersten OUT-Befehl aktiviert den Steuerport, $40 im zweiten den Datenport des Timers. Die etwas seltsam anzusehenden JMP-Befehle auf die jeweils nächste Zeile sind »Minizeitschleifen«. Sie existieren, weil der Prozessor schneller Daten auf die Ports ausgeben kann, als es die Elektronik der angesprochenen Bausteine verkraftet. Durch diese vergleichsweise langsamen Jumps wird dem Timer die Möglichkeit gegeben, auf die Befehle zu reagieren.

Nun kommt die eigentliche Zeitmessung. Doch auch hier gibt es noch ein Problem: Der Assembler streut vor jeden FYL2X-Befehl ein FWAIT, das hier nicht nur unnötig ist, sondern auch die Zeitmessung verfälscht. Daher muß es eliminiert werden. Die natürliche Lösung wäre, mittels .287 den Assembler dazu zu veranlassen, die FWAITs zu unterdrücken. Dann aber haben wir keine 8087-Kompatibilität mehr.

Also verwenden wir wieder unseren DB-Trick. Aus Teil 3 können wir den Opcode entnehmen, den FYL2X hat: $D9, $F1. Also schreiben wir anstelle der sieben FYL2X-Befehle siebenmal DB 0D9h, 0F1h:

```
db 0D9h,0F1h
db 0D9h,0F1h
db 0D9h,0F1h
db 0D9h,0F1h
db 0D9h,0F1h
db 0D9h,0F1h
db 0D9h,0F1h
fwait
```

Das letzte FWAIT allerdings muß sein, da wir ja warten müssen, bis der Coprozessor fertig ist. Was nun kommt, ist einfach. Wir lesen nun den Timerbaustein aus und rechnen die Anzahl der Ticks, die er gemessen hat, in eine Rechengeschwindigkeit um:

```
         xor   al,al
         out   043h,al
         jmp   L5
L5:      jmp   L6
L6:      in    al,040h
         xchg  ah,al
         jmp   L7
L7:      jmp   L8
L8:      in    al,040h
         xchg  ah,al
```

Dazu weisen wir mit `out 043h`, al und 0 in AL den Timer über einen Befehl an den Steuerport an, uns byteweise den Zählerstand zu übermitteln. Dazu sind zwei Lesebefehle (`in al, 040h`) des Datenports notwendig, die das Byte in AL ablegen, das somit temporär in AH zwischengespeichert wird.

Nach dieser Sequenz liegt nun der Zählerstand des Timers nach dem letzten FYL2X-Befehl vor. Doch der Zähler hat dekrementiert, also zurückgezählt. Daher ziehen wir diesen Zählerstand von 0 ab, um die Anzahl von Systemticks zu erhalten. Erreicht wird dies mit NEG:

```
neg     ax
sti
mov     [Temp],ax
```

Nachdem wir die Zeitmessung durchgeführt haben, können wir uns wieder durch Interrupts stören lassen – ja, wir müssen es sogar, falls wir nicht wollen, daß sich das System stillschweigend verabschiedet. Denn der DOS-Ticker beispielsweise muß ja weiterlaufen. Also: STI! Anschließend speichern wir den Zählwert in der Variable *Temp*, die offensichtlich eine Mehrzweckvariable für uns ist.

An dieser Stelle haben wir nur einen Zählerstand, der angibt, wie viele Systemticks während der Ausführung der sieben FYL2X-Befehle vergangen sind. Eine Geschwindigkeit ist das noch nicht! Zur Ermittlung der Geschwindigkeit gibt es zwei Möglichkeiten:

- Wir müssen wissen, welche Frequenz der Systemtick hat. Dann ist alles einfach: Anzahl Ticks (gemessen) · Systemtickfrequenz (bekannt) ÷ 7 (Anzahl gemessener Befehle) ÷ Anzahl Takte pro FYL2X = Geschwindigkeit.

  Diese Methode hat jedoch mehrere Nachteile: Erstens kennen wir die Systemtickfrequenz, wenn überhaupt, nur sehr ungenau, zu ungenau für solche Berechnungen: 1,19... MHz! Zweitens müssen wir berücksichtigen, daß auch die Port-Befehle bei der Zeitmessung zumindest eine kleine Verfälschung bewirkt haben. Denn so schnell kann man zwischen einzelnen Komponenten nicht »umschalten«. Drittens verfälscht auch der absolut notwendige FWAIT-Befehl am Ende der Messung die theoretische Taktrate: Bei 8087ern ist der Wert abhängig von der uns unbekannten Rate, mit der der 8086/8088 die *busy*-Leitung abfragt (siehe Teil 3).

  Außerdem ist die Taktrate für die Befehle nicht exakt bestimmbar. Sie variiert unter anderem aufgrund unterschiedlicher Werte, die zu verarbeiten sind. So ist ein FYL2X mit 1 und 2 als Operanden sicherlich schneller ausgeführt als mit π und dem Ergebnis irgendeiner vorher abgelaufenen Operation, in diesem Fall FYL2X. Wir müßten also für jede FYL2X-Zeile die exakte Taktrate bestimmen.

- Wir messen einfach die Ticks auf einigen Computern, bei denen die Arbeitsgeschwindigkeit des Coprozessors bekannt ist. Dann brauchen wir nur die Anzahl der gemessenen Ticks mit dieser Geschwindigkeit zu multiplizieren, um eine Konstante zu erhalten, durch die wir dann umgekehrt den durch die eben durchgeführte Messung erhaltenen Wert für die aktuellen Ticks dividieren. Das Resultat ist die Arbeitsgeschwindigkeit des betrachteten Coprozessors!

In Abhängigkeit vom Code des Coprozessors holen wir die Konstante und dividieren sie durch den eben gemessenen Wert für die Ticker – und schon haben wir eine Geschwindigkeit. Wir lassen diese Division vom Coprozessor durchführen, da dies ja seine eigentliche Aufgabe ist.

Wie wir an die Konstante herankommen, dürfte nach der Meldungsausgabe klar sein: Wir schreiben alle Konstanten hintereinander in eine Tabelle, die wir hier *C_87* nennen, weil dies das erste Label ist, das auf die für den 8087 zuständige Konstante zeigt. Achtung! Es gibt keinen 80187! Aber wir haben einen solchen Code implizit vorgesehen, obwohl wir ihn nicht nutzen. Aus diesem Grunde müssen wir zwischen *C_8087* und *C_80287* eine sogenannte *Dummy*-Variable legen, die keinen Wert zu erhalten braucht, weil sie niemals verwendet wird, die also nur zwei Bytes Platz verschwendet!

Der Tabellenindex läßt sich ebenso leicht berechnen wie bei der Meldungsausgabe. Doch hier müssen wir den Code des Coprozessors mit 4 multiplizieren, da die Größe der Variablen nicht 2 wie bei den Offsets ist, sondern 4, um ein Doppelwort für die Realzahlen aufzunehmen.

```
xor     bh,bh
mov     bl,[T_87]
shl     bx,1
shl     bx,1
```

BH müssen wir mit 0 belegen, weil der Code nur ein Byte ist, für den Tabellenindex aber immer das gesamte BX-Register verwendet wird. Somit führen wir quasi ein *Type-Casting* eines Bytes in ein Wort durch!

Bevor wir nun mit der Division beginnen, müssen wir uns überlegen, was noch in welchen Coprozessorregistern steht, wenn wir sauber programmieren wollen! FYL2X poppt nach der Berechnung einen Wert vom Stack. Nach sieben FYL2X-Befehlen mit acht Konstanten bleibt also das Ergebnis des letzten FYL2X-Befehls im TOS übrig. Dieses brauchen wir jedoch nicht, so daß wir den Stack zunächst wieder aufräumen. Dies erfolgt, indem wir den Inhalt des TOS leeren und den Stack-Pointer inkrementieren.

```
ffree   st
fincstp
fld     [C_087+bx]
fidiv   [Temp]
fistp   [Temp]
fwait
```

Dann laden wir die Konstante. Wir verwenden hier wieder den Offset auf die Tabelle, also das Label *C_087* und addieren einfach den Tabellenindex in BX dazu. Diese Summe können wir als Operand für den FLD-Befehl angeben, da der Assembler durch die Deklaration von *C_087* als DD weiß, daß der dort verzeichnete Wert ein DWord ist.

Anschließend führen wir eine Integerdivision mit dem gemessenen Wert durch und speichern das Resultat ebenfalls als Integer in *Temp* zurück! Dies erfolgt durch FISTP, das nicht nur die Rundung der Realzahl auf die nächste Integerzahl durchführt, sondern den Stack auch poppt, so daß dieser nun wieder so zurückgelassen wird, wie wir ihn vorgefunden haben. Da wir eine Speicherstelle beschreiben, auf die im nächsten Schritt auch der Prozessor zugreift, müssen wir mit FWAIT dafür sorgen, daß der eine auf den anderen wartet!

```
        mov   ax,[Temp]
Aus:    pop   bx
        pop   ds
        ret
Get_Speed ENDP
```

Nun holen wir das Rechenergebnis aus *Temp* in das AX-Register, weil *Get_Speed* eine Funktion ist, die definitionsgemäß ihr Ergebnis in AX übergibt! Das Poppen des Stacks in BX ist nur die notwendige Restauration dieses Registers und erfolgt gemäß der oben angeführten Regel: »Auf jedes Pushen folgt ein Poppen!«

Soweit das Programm GET_87. Klären wir noch den Unterschied zwischen AND – CMP und TEST, den wir auf Seite 294 angesprochen haben.

Da TEST ja eigentlich nur ein AND-Befehl ist, bei dem das Ergebnis der UND-Verknüpfung nach dem Setzen der Flags verworfen wird, läuft das Problem auf die Frage hinaus, was der Unterschied zwischen AND – CMP und AND ist. Oder anders formuliert: Was macht CMP *nach* einem AND, was AND nicht schon selbst könnte?

Die Frage: »Flags setzen oder nicht« ist es nicht, da beide Befehle das Ergebnis in Form veränderter Flags auswerten. Es handelt sich also nur um das Wie! Überlegen wir, was wir mit der Prüfung eigentlich erreichen wollten! Wir wollten eigentlich nur feststellen, ob die Bits 15

bis 12 des Flagregisters gesetzt sind oder nicht. Nun haben aber die restlichen Bits 11 bis 0 auch einen bestimmten Zustand, den wir nicht kennen, der uns aber auch nicht interessiert. Also überführen wir sie (für den Test, nicht etwa tatsächlich im Flagregister) in einen definierten Zustand, z.B. in den gelöschten. Erreicht wird dies durch eine UND-Verknüpfung mit einer Maske, bei der die interessierenden Bits 15 bis 12 gesetzt sind, die uninteressanten Bits 11 bis 0 nicht. Das Resultat dieser Verknüpfung ist nun ein Wert, bei dem in Abhängigkeit vom ursprünglichen Testwert die Bits 15 bis 12 – und nur diese – gesetzt sind.

Würden wir nun lediglich die Flags, die AND gesetzt hat, berücksichtigen, so könnten wir mit dem Zero-Flag nur feststellen, ob alle interessierenden Bits gelöscht sind. Dann nämlich ist das Zero-Flag gesetzt. Ist es dagegen gelöscht, so wissen wir nicht, ob alle Bits 15 bis 12 gesetzt sind oder nur einige – oder sogar nur eines! Wir erhalten also nur Informationen über »mindestens ein Bit oder keines«. Bedingung für die Entscheidung »< 80286 oder nicht« ist aber, daß *alle* Bits 15 bis 12 gesetzt sein müssen, also »alle (betrachteten!) Bits«! Also muß das Resultat mit dem Wert $F000 verglichen werden! Denn nur durch diesen Vergleich mittels CMP werden alle (betrachteten) Bits berücksichtigt!

TIP

Denken Sie daher immer daran: Logische Befehle wie AND, OR, XOR und TEST arbeiten immer bitweise und lassen daher eine gleichzeitige Prüfung auf mehrere Bits nicht zu! Soll der Zustand mehrerer Bits simultan festgestellt werden, so muß dies mit arithmetischen Befehlen wie ADD, SUB und CMP erfolgen! Diese Befehle verändern die Flags anhand der Resultate mehrerer Bits in Kombination. Allerdings ist der Einsatz solcher arithmetischer Befehle für logische Probleme nur dann sinnvoll, statthaft und korrekt, wenn die (arithmetisch) zu prüfenden Werte vorher ggf. (logisch) »aufbereitet« wurden.

Hätten wir nämlich im obigen Fall den AND-Befehl nicht vorgeschaltet, so bekämen wir mit Sicherheit niemals eine korrekte Entscheidung! Denn falls nur eines der Flags 11 bis 0 (also Zero-, Sign-, Carry-Flag etc., aber auch Flags, die sich unserer Verantwortung und Manipulierbarkeit entziehen!) gesetzt wäre, so könnte niemals das Zero-Flag gesetzt werden!

23 Makros

Wenden wir uns einer weiteren Möglichkeit zu, die einem beim Programmieren in Assembler wertvolle Hilfe bringen kann: Makros! (Übersetzt etwa: »Riesen« – der Name rührt daher, daß im Unterschied zum *Microcode*, also zum fest verdrahteten Code auf dem Chip,

aus dem Befehle bestehen, mehrere Befehle zu einem »Superbefehl«, einem *Macrocode*, zusammengefaßt werden können.) Makros sind Deklarationen, die eine bestimmte Bytefolge mit einem Namen versehen. Dieser Name kann nun in Programmen wie ein Assemblerbefehl verwendet werden. Ein Beispiel:

```
PackBcd MACRO P
         IFNB <P>
            mov  ax,P
         ENDIF
         DB  0D5h
         DB  010h
ENDM
```

Die Makrodefinition ähnelt der einer Routinendeklaration. Hinter dem Namen des Makros erfolgt die Deklaration durch das Schlüsselwort MACRO, dem noch Pseudoparameter übergeben werden können, die im Makro Verwendung finden sollen.

In unserem Beispiel wird also ein Makro namens *PackBcd* deklariert, dem ein Parameter *P* mitgegeben wird. Schweifen wir kurz zu etwas ab, das mit dem eigentlichen Makro nichts zu tun hat: zur bedingten Assemblierung. Genau wie in Hochsprachen auch, kann mit Assemblern die Assemblierung an bestimmte Bedingungen geknüpft werden. Assembliert wird nur, wenn diese Bedingungen erfüllt sind.

Oben folgt der Anweisung `IFNB <P>` ein Block, der mit ENDIF abgeschlossen wird. IFNB (*If Not Blank*) weist den Assembler an, den eingeschlossenen Block nur dann zu assemblieren, wenn der Parameter *P* einen Inhalt hat (also nicht *blank* ist!), das Makro also mit Parameter verwendet wird. In diesem Fall soll der Assembler den Befehl `mov ax`, P berücksichtigen, andernfalls nicht.

Nach dem bedingt assemblierten Block folgen zwei Datenbytes: `$D5` und `$10`. Nach unserem derzeitigen Kenntnisstand sollen Makros doch Strukturen sein, die der Assembler in den Programmcode an gewissen Stellen einstreut! Der Anfang mit dem eventuellen `mov ax, P` scheint dies ja auch zu bestätigen! Und richtig, diese beiden Bytes stellen auch den Opcode eines Befehls dar!

Sie brauchen lediglich im Anhang nachzuschauen, für welchen Befehl das Byte $D5 steht. Wenn Sie dies tun, so finden Sie den Befehl AAD. Doch der vollständige Opcode für AAD heißt *D5 – 0A* und nicht, wie hier *D5 – 10*! Doch wenn Sie die Referenz zu AAD im Referenzteil aufmerksam lesen, werden Sie feststellen, daß das Byte $0A im Befehl AAD nur der Multiplikator ist, den AAD verwendet. Den können wir über die Definition mittels DBs ändern.

Wenn wir also den Assembler den Befehl AAD übersetzen lassen, so erzeugt er immer die Sequenz *D5 – 0A*! Ein *AAD (16)* aber erkennt der Assembler nicht als legales Mnemonic an. Daher erfolgt der Umweg über *DB 0D5h, 010h*. Was aber tut nun das Makro? Falls dem Makro ein Parameter übergeben wird, so wird zunächst der Inhalt dieses Parameters in AX kopiert. Anschließend wird AAD mit dem Multiplikator 16 aufgerufen. AAD macht nichts anderes als:

AL := AH · 16 + AL

AAD »packt« also lediglich zwei in AH und AL übergebene (BCD-) Ziffern. Bevor wir das Makro nun einmal anwenden, definieren wir noch drei weitere:

```
UnpackBcd MACRO P
          IFNB <P>
             mov  al,P
          ENDIF
          DB  0D4h
          DB  010h
ENDM

Bcd2Bin MACRO P
          IFNB <P>
             mov  al,P
          ENDIF
          DB  0D4h
          DB  010h
          DB  0D5h
          DB  00Ah
ENDM

Bin2Bcd MACRO P
          IFNB <P>
             mov al,P
          ENDIF
          DB  0D4h
          DB  00Ah
          DB  0D5h
          DB  010h
ENDM
```

UnpackBcd verwendet den Befehl AAM mit dem Divisor 16, führt also die Operation

AH := AL DIV 16; AL := AL MOD 16

aus und ist praktisch die Umkehrung von *PackBcd*. Die beiden anderen Makros kombinieren AAD und AAM mit verschiedenen Operanden. So besteht *Bcd2Bin* aus der Folge *AAM (16) – AAD (10)*, während *Bin2BCD* umgekehrt *AAM (10) – AAD (16)* ausführt. *Bcd2Bin* steht also für

AH := AL DIV 16; AL := AL MOD 16; AL := AH · 10 + AL

womit erreicht wird, daß die in AL stehende Binärzahl in eine BCD verwandelt wird. Umgekehrt erzeugt *Bin2Bcd* über

AH := AL DIV 10; AL := AL MOD 10; AL := AH·16 + AL

aus einer BCD eine Binärzahl. Alle angesprochenen Makros liegen im Quelltext auf der CD-ROM in der Datei MACRO.INC vor.

Doch nun wollen wir die Makros einsetzen. Wir stellen uns die Aufgabe, aus einem String, der das ASCII-Zeichen '9' enthält und die Zahl 99 repräsentiert, einen String mit der »Zahl« '66' zu erzeugen.

```
Code SEGMENT BYTE PUBLIC


NintyNine DB '99'
SixtySix  DB 2 DUP (?)


ASSUME CS:Code, DS:Code


TestProc PROC NEAR
         push  ds
         push  cs
         pop   ds
```

Ich glaube, mit dem Präludium müssen wir uns nicht mehr aufhalten! Daher gleich weiter:

```
mov      ax,WORD PTR NintyNine
and      ax,00F0Fh
```

Mit dem MOV-Befehl holen wir uns den String '99' in AX. Wir müssen hierbei über WORD PTR ein *Type-Casting* durchführen, da AX ein Wortregister ist, das Label *NintyNine* aber auf ein Byte zeigt. Der Assembler erzeugt sonst wieder eine Fehlermeldung. Anschließend extrahieren wir aus dem Wert $3939, der nun in AX steht, die Ziffern 9. Dies erfolgt, indem wir das jeweils »obere Nibble« der Bytes in AH und AL löschen, wozu wir den AND-Befehl verwenden. Nun haben wir in AX den Wert $0909 stehen. Das sieht schon sehr nach einer ungepackten BCD aus. Da wir mit BCDs aber schlecht rechnen können, transformieren wir sie in eine

Binärzahl. Dazu jedoch müssen wir die ungepackte BCD zunächst »packen«, also die beiden Ziffern zusammen in AL holen.

```
PackBcd
Bcd2Bin
```

Hier kommt nun schon unser erstes Makro zum Einsatz. Wie bereits erwähnt, brauchen wir nun an dieser Stelle nur das Makro *PackBcd* aufzurufen, das für das Packen zuständig ist. Anschließend rufen wir mit *Bcd2Bin* das Makro auf, das die Konvertierung der Zahl vornimmt.

Zwei Dinge fallen auf:

- Makro-Aufrufe erfolgen ganz analog zu Aufrufen von Assemblerbefehlen, einfach durch Angabe des Makronamens.
- Beide Makros werden hier ohne Parameter verwendet, so daß der Assembler den einen Fall der bedingten Assemblierung berücksichtigen muß.

Was wir nun haben, ist die Binärzahl 99 in AL, ...

```
sub       al,33
mov       dl,al
xor       ax,ax
```

... von der wir nun ganz »normal« 33 abziehen, um die gewünschten 66 zu erhalten! Das folgende Kopieren des Ergebnisses in DL und das Löschen von AX dient nur dazu, um zu demonstrieren, was passiert, wenn man die Makros mit einem Parameter aufruft! Das tun wir jetzt:

```
Bin2Bcd dl
mov       cl,al
xor       ax,ax
UnpackBcd cl
```

Dem Makro *Bin2Bcd*, das umgekehrt zu *Bcd2Bin* eine Binärzahl in eine BCD-Zahl umwandelt, wird als Parameter das Register übergeben, in dem die BCD steht, hier also DL. Das Ergebnis des Makros steht wieder in AL, so daß wir es wiederum zu Demonstrationszwecken in CL kopieren und AX löschen, bevor wir *UnpackBcd* mit dem Parameter CL aufrufen.

```
          or    ax,03030h
          mov   WORD PTR SixtySix,ax
          pop   ds
          ret
TestProc ENDP

Code ENDS

END
```

Schließlich »addieren« wir zu der so erzeugten ungepackten BCD $3030, um aus den BCD-Ziffern ASCII-Zeichen zu machen, die wir dann am Label *SixtySix* ablegen, wobei wiederum ein *Type-Casting* erforderlich wird.

Sie müssen zugeben, daß Makros großartige Hilfsmittel sind, um in Assembler zu programmieren. Sie sind einfach deklarierbar, können bequem angewendet werden und tragen sehr zur Lesbarkeit des Quelltextes bei. Oder finden Sie den folgenden Ausschnitt informativer?

```
:
DB  0D5h,010h
DB  0D4h,010h
DB  0D5h,00Ah
sub     al,33
mov     dl,al
xor     ax,ax
mov     al,dl
DB  0D4h, 00Ah, 0D5h, 010h
mov     cl,al
xor     ax,ax
mov     al,cl
DB  0D4h
DB  010h
:
```

ACHTUNG

Makros sind keine Unterprogramme! Dieser Fehler wird von Assemblerneulingen häufig gemacht. Während Routinen – also Prozeduren und Funktionen – nur ein einziges Mal im Assemblat erscheinen und Aufrufe dieser Routinen immer eine Programmverzweigung an die entsprechende *eine* Stelle (mit abschließendem Rücksprung) zur Folge haben, gibt es so viele Kopien eines Makros, wie es Aufrufe dieses Makros gibt!

Etwas nüchtern überlegt, ist das auch ganz logisch! Makros sind ja eigentlich nichts anderes als Pseudonyme für eine bestimmte Folge von Befehlen. Sie dienen dazu, einer Bytefolge eine für Menschen besser interpretierbare Bedeutung zu geben. Das bedeutet aber, daß der Assembler immer dann, wenn er auf den Namen eines Makros im Quelltext stößt, diesen durch die in der Makrodefinition angegebenen Assemblerbefehle ersetzt! Praktisch macht also der Assembler beim Assemblieren des Quelltextes nichts anderes als eine Textverarbeitung, bei der man ein bestimmtes Wort im gesamten Text suchen und durch ein anderes Wort oder eine Wortfolge ersetzen lassen kann.

TIP

Verwenden Sie daher Makros nur dann, wenn Sie eine bestimmte Befehlssequenz häufiger benötigen, aus verschiedenen Gründen jedoch nicht als Routine realisieren wollen. Die Definition von Makros in diesem Fall erhöht sicherlich die Lesbarkeit des Textes und dient somit der einfacheren Pflege des Programms. Sie hat aber – verglichen mit der direkten Programmierung – keinen Vorteil hinsichtlich Codelänge oder Ausführungsgeschwindigkeit.

Schauen wir uns einmal das vom Assembler aus dem obigen Quellcode erzeugte Assemblat an. Im folgenden Listing finden Sie hinter den Offsets im Codesegment (den Adressen) die Opcodes und anschließend die disassemblierten Mnemonics[9].

```
cs : 0000 3939
     0002 0000
     0004 1E            PUSH    DS
     0005 0E            PUSH    CS
     0006 1F            POP     DS
     0007 A10000        MOV     AX,[0000]
     000A 250F0F        AND     AX,0F0F
     000D D510          AAD     10
     000F D410          AAM     10
     0011 D50A          AAD
     0013 2C21          SUB     AL,21
     0015 8AD0          MOV     DL,AL
     0017 33C0          XOR     AX,AX
     0019 8AC2          MOV     AL,DL
     001B D40A          AAM
     001D D510          AAD     10
     001F 8AC8          MOV     CL,AL
     0021 33C0          XOR     AX,AX
     0023 8AC1          MOV     AL,CL
     0025 D410          AAM     10
     0027 0D3030        OR      AX,3030
```

[9] Wenn Sie diese genauer betrachten, so sehen Sie, daß einige Debugger, von denen man das gar nicht erwartet, in gewissen Dingen fortschrittlicher sind als die meisten anderen! Während nämlich z.B. der Turbo Debugger TD nichts außer AAD und AAM kennt und gar nicht auf die Idee kommt, daß man von Hand ja etwas an dem Multiplikator/Divisor im Befehl »drehen« könnte, scheint dies für den ganz normalen Debugger DEBUG, der mit DOS ausgeliefert wird, selbstverständlich zu sein – zumindest für den mit DOS 5.0 ausgelieferten! Denn der disassembliert z.B. $D5, $10 sehr korrekt zu AAM 10, während TD stur DB D5, DB 10 meldet, mit den sich daraus ergebenden Fehlern in der Disassemblierung!

```
002A A30200         MOV     [0002],AX
002D 1F             POP     DS
002E C3             RET
```

Wie man sieht, hat der Assembler an den (fett gedruckten) Stellen unseres Quellcodes, an denen wir die Makroaufrufe verwendet haben, die im Makro definierte Codesequenz eingetragen. Aufrufe von Unterprogrammen mit CALL fehlen in diesem Text naturgemäß genauso wie Unterprogramme selbst. Auch die im Quelltext vorangestellte Definition der Makros findet sich nirgends. Denn schließlich wird diese nicht mehr gebraucht, nachdem der Assembler alle Makronamen im Text durch den dazugehörigen Code substituiert hat.

Wie Sie sehen können, hält sich der Assembler sehr genau an die Angaben, die ihm durch die *Bedingte Assemblierung* gegeben wurden. Nur an den Stellen, an denen dem Makro im Quelltext ein Parameter übergeben wurde, fügte er die für den entsprechenden Fall vorgesehenen Befehle ein.

24 Namen für Werte: EQU

Wenn wir mit Labels und Makros nun schon zwei Arten kennengelernt haben, mit denen man sich das Leben erleichtern kann, wollen wir noch eine letzte Art der Benennung vorstellen. Fassen wir dazu zunächst noch einmal zusammen, was wir im Assembler bisher alles schon benennen können:

- *Labels* dienen dazu, bestimmte Bytes im Quelltext zu markieren. Labels unterscheiden nicht zwischen Programmcode oder Daten. Für den Assembler ist bei Labels lediglich wichtig, an welcher Stelle sie stehen, denn der Assembler ersetzt beim Assemblieren des Quelltextes den Namen des Labels durch die Adresse, auf die es zeigt. Somit sind Labels nichts anderes als Zeiger auf ein bestimmtes Datum – einerlei, ob Code oder echtes Datum.
- *Makros* sind »Platzhalter«. Sie stehen für eine bestimmte Befehlssequenz. Hier wird nicht der Name durch eine Adresse substituiert, sondern durch eine Reihe von Bytes.

Fehlt eigentlich nur noch, daß man bestimmte Werte benennen kann. Dies ist mit dem Assembler auch möglich. So kann jedem beliebigen numerischen Wert ein Name zugeordnet werden:

```
Null                EQU 0
Eins                EQU 1
```

```
EchtKoellnischWasser EQU 4711
Text                 EQU 'Dies ist ein Text'
```

EQU Die Zuordnung eines Wertes zu einem Namen erfolgt durch die Assembleranweisung EQU. EQU steht für *equal* und kann auch durch das Zeichen »=« ersetzt werden. *Null EQU 0* heißt also nichts anderes, als daß der **Name** *Null* mit dem **Wert** *0* gleichgesetzt wird.

Auf diese Weise kann der Assembler, wann immer er im Quelltext auf den Namen *Null* stößt, diesen durch den Wert 0 ersetzen. Aber Achtung: ob der Wert zu einem Byte, Wort oder Doppelwort gehört, hängt von der Situation ab. Die Befehle

```
mov     ax,Null
cmp     bl,Null
```

sind beide erlaubt. Sie werden vom Assembler in

```
mov     ax,00000h
cmp     bl,000h
```

abgeändert. Der Assembler führt hierbei eine Plausibilitätsprüfung durch. Die folgende Zeile führt zu einer Fehlermeldung

```
mov     cl,EchtKoellnischWasser
```

da *EchtKoellnischWasser* den Wert 4711 hat und mit Sicherheit nicht in einem Byte unterzubringen ist.

ACHTUNG Wie Makronamen und Labels sind auch EQU-Zuweisungen nur *Pseudonyme*, wie es z.B. auch *untypisierte Konstanten* in Hochsprachen sind. Mit der EQU-Anweisung haben Sie noch kein Datum erzeugt. Sie haben lediglich einem bestimmten Wert einen Namen gegeben.

Somit ist die Nützlichkeit dieser Anweisung zunächst wenig offensichtlich. Denn schließlich ist es ja egal, welche der beiden folgenden Befehle man im Quelltext benutzt, da sie äquivalent sind:

```
mov     ax,00000h
mov     ax,Null
```

Dennoch werden Sie spätestens dann froh sein, daß es diese Möglichkeit gibt, wenn wir uns auf Seite 333 näher mit lokalen Variablen auf dem Stack beschäftigen.

25 Assembler und Hochsprachen

Auf Seite 281 haben wir uns schon einmal mit dem Stack beschäftigt. Dort nannten wir ihn nicht ganz zu unrecht ein »Datensegment des Prozessors«, da dieser den Speicher, der über die Registerkombination SS:SP angesprochen wird, für verschiedene Zwecke benutzt. So legt jeder CALL-Befehl die Rücksprungadresse auf den Stack, die von RET wieder entfernt wird. Ein INT-Befehl legt nicht nur eine Rücksprungadresse dort ab, sondern auch den aktuellen Inhalt des Flagregisters. Auch hier sorgt IRET wieder für das korrekte Entfernen der temporär abgelegten Daten.

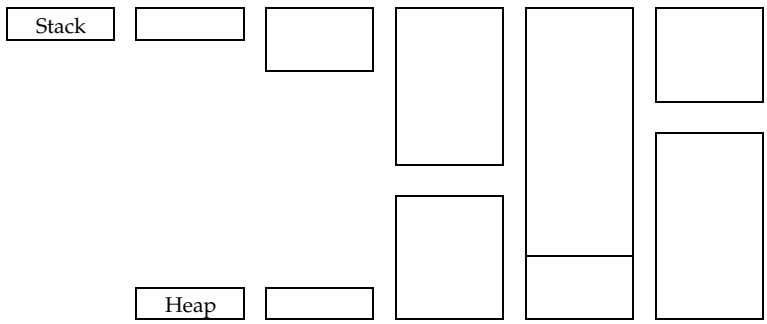
Wir haben auch gelernt, daß wir den Stack für unsere Zwecke benutzen können! So dienen die Befehle PUSH und POP dazu, Daten aus Registern oder Speicherstellen dort abzulegen oder sie von dort wiederzuholen. Einen Stack zu definieren dürfte inzwischen kein Problem mehr sein! Weiter vorn haben wir den Stack als Stapel bezeichnet und das Bild eines Tellerstapels verwendet. Hier liegt nun das Problem! Denn der Stack ist ein Stapel, der »von oben nach unten« wächst, also genau andersherum, als wir es gewohnt sind.

Der Grund, warum der Stack von oben nach unten wächst, hat wieder einmal traditionelle, aber auch recht gut durchdachte Gründe! Ais der 8086 erschien, gab es nur die Möglichkeit, Programme mit maximal 64 kByte zu erstellen. In diesen 64 kByte nun mußte alles verstaut werden, was ein Programm zum Laufen benötigte: Codesegment, Datensegment und Stacksegment. Da nun die Maximalgröße konstant und vorgegeben war, dachten sich die Entwickler, daß man dieses Supersegment möglichst optimal und so nutzt, daß man bei höchster Flexibilität ein Optimum an Sicherheit hat.

Nun ist der Stackbedarf sehr unterschiedlich und schlecht vorhersehbar! Werden Programmteile verwendet, die den Stack stark belasten, wie man sagt, so wird viel Platz dafür beansprucht. Dies kann aber nur von kurzer Dauer sein. Andererseits kann es sein, daß im Verlauf von Programmen der Bedarf an Speicherplatz für Variablen steigt. Dies ist zwar weniger bei Compilern der Fall, da hier durch das Übersetzen der maximal benötigte Datenbedarf vor dem Ausführen des Programms errechnet werden kann. Aber es gab und gibt ja noch Interpreter! Das Problem läßt sich ganz analog auch für Compiler heutzutage aufzeigen, die ja den sogenannten Heap, also den freien Speicher zwischen allen definierten Segmenten und dem Stack, nutzen können und es auch häufig genug sehr ausgeprägt tun.

So haben wir zwei »unsichere« Größen: die variable Größe des Stacks und die des Datenbereichs (also Datensegment bzw. Heap). Was tut

man, wenn man eine solche Situation vorfindet, alles aber in einen festen Kasten pressen muß? Man polarisiert und läßt den einen variablen Bereich am einen, den anderen am anderen Ende beginnen. Bei zunehmendem Platzbedarf wachsen nun beide Bereiche aufeinander zu, bis sie sich irgendwo berühren. Diese Grenze jedoch muß nicht festgelegt sein! Sie kann je nach Stackbelastung und Datenbedarf extrem weit von der Mitte verschoben sein! Schauen wir uns dies einmal an:



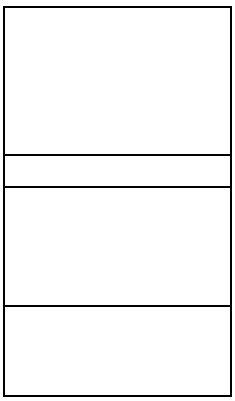
In der Regel hat praktisch jedes Programm einen Stack. Dadurch, daß man nun Stack und Heap polarisiert an den Grenzen des nutzbaren Bereichs ansiedelt, ist man sehr flexibel in der Ausnutzung geworden. Es darf nur eines nicht passieren: Stack und Heap dürfen nicht mehr Platz beanspruchen, als vorhanden ist. Dann nämlich gibt es einen sogenannten Stack-Heap-Konflikt, der sich darin äußert, daß der (nicht ausschließlich unserer Kontrolle unterstehende) Stack Teile des Heaps überschreibt!

Das Schaubild erinnert sehr an Tropfsteinhöhlen: unten die Stalagmiten (Heap) und oben die Stalagtiten (Stack). Dort wie hier wachsen beide in entgegengesetzte Richtungen! Dieses Bild erklärt eigentlich auch ganz zwanglos, warum nun der Stack »nach unten« wächst: Es ist erheblich einfacher, neu hinzukommende Daten am aktuellen Stackende anzusiedeln als zunächst alle Daten um einen gewissen Betrag nach unten zu verschieben, nur um »oben« auf den Stack schreiben zu können.

TIP Wenn Sie daher ab jetzt etwas vom Stack und wachsenden Stackadressen zu hören bekommen, denken Sie an die Stalagtiten. Je höher der Stapel ist, desto niedriger ist die Adresse, an der die Spitze liegt.

25.1 Sind Sie im Bilde? Aber fallen Sie bloß nicht aus dem Rahmen!

Der Stack nimmt nicht nur anders zu als andere Datenbereiche – er ist auch strukturiert! Und zwar in wirklich sehr sinniger Weise. So besteht der Stack eigentlich aus mehreren Blöcken unterschiedlicher Größe, die aufeinander folgen. Wie das dann optisch aussehen könnte, sehen Sie hier:

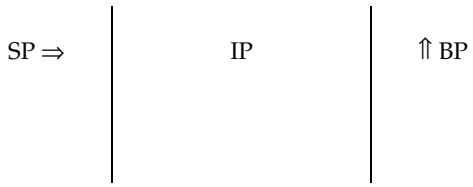


Wir wissen ja nun, daß der »neueste« Bereich hier unten am Ende liegt und weitere Daten dort angefügt werden. Doch wer hat die Strukturen eingeführt – und wie und warum?

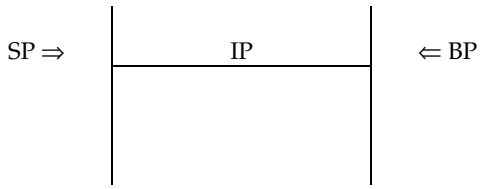
Die Antwort ist einfach: Sie! Wenn Sie nämlich zum Beispiel eine Routine aufrufen, so legt der Prozessor ja die Rücksprungadresse auf den Stack. So, – und nun sind wir »in einer anderen Welt«, dem Unterprogramm. Dieses hat zunächst einmal nichts mit dem Hauptprogramm zu tun. Daten, die hier anfallen, interessiert nur das derzeit ablaufende Unterprogramm – und sonst niemanden. Wenn nämlich von hier aus z.B. wiederum eine Routine aufgerufen wird, so braucht das Hauptprogramm nicht zu interessieren, wo die Rücksprungadresse zu liegen kommt. Denn um dorthin zurückzukehren, müssen wir zunächst hierher zurückkommen!

Somit erklären wir einfach einen kleinen Bereich des Stacks zu unserem Privatbesitz, in dem wir Daten verwalten, die niemand anderen etwas angehen. Wie wir das tun, ist sehr trickreich! Wie schon gesagt, zeigt die Registerkombination SS:SP immer auf das Ende des Stapels. Es gibt aber noch ein anderes Register, das beim Themenkreis Stack eine Rolle spielt: BP. Das *Base-Pointer-Register* enthält immer die Adresse des Anfangs des aktuellen Bereichs auf dem Stack. Geladen wird dieses Register von uns. Sehen wir uns dies einmal an. Stellen Sie sich vor, ein CALL-Befehl hätte

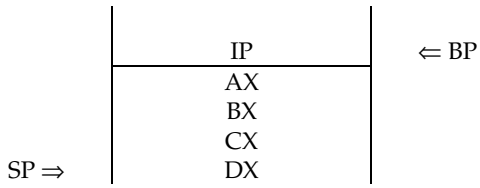
eine Routine aufgerufen. Dann läge jetzt, NEAR-Adressen vorausgesetzt, der Offset der Rücksprungadresse auf dem Stack und BP würde auf eine uns unbekannte Adresse zeigen:



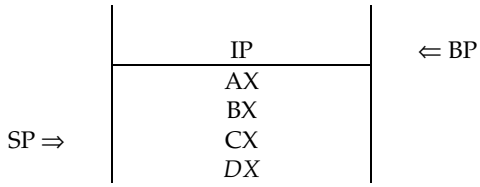
Wenn wir nun den Inhalt von SP in BP kopieren, so definieren wir gleichsam eine obere Grenze des aktuellen Bereichs. Alles, was davor steht, hat nichts mit unserem Unterprogramm zu tun!



Nun können wir auf dem Stack ablegen, was wir wollen. So könnten wir z.B. die vier Rechenregister dort ablegen, wozu wir vier PUSH-Befehle verwenden:



Poppen wir dann den Stack, so reduziert sich die Größe des Bereichs:

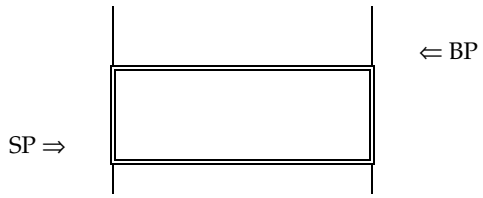


Der Inhalt von SP-2 (Achtung: Stalagtiten!), also die Spitze des Stapels, beinhaltet zwar noch den Wert von DX. Aber gemäß Konvention sind alle

Daten unterhalb von SP undefiniert! Wie man aus diesen Bildern erkennen kann, werden alle Daten zwischen den Adressen, die in SP und BP stehen, von der aktuellen Routine angelegt und verwaltet.

HINWEIS

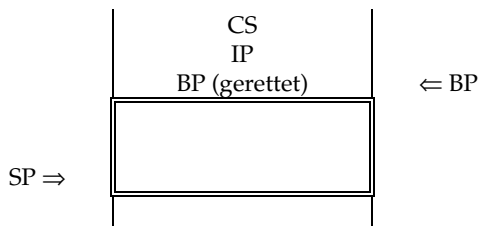
SP und BP zusammen definieren also den lokalen Stackbereich der aktuellen Routine. Da dieser Bereich ein Ausschnitt aus dem Gesamtstack ist, kann man SP und BP als Zeiger interpretieren, die einen Rahmen um den derzeit gültigen Stackbereich legen. Dieser Bereich heißt daher auch Stackrahmen!



Kommen wir nun dazu, wie man den Stackrahmen in Programmen einfügen kann. Wir haben schon gehört, daß durch Kopieren von SP in BP die obere Grenze festgelegt wird, durch die Modifikation von SP dann die untere. Bevor wir jedoch den Inhalt von BP einfach überschreiben – es könnte ja die obere Grenze des Rahmens der aufrufenden Routine sein – werden wir diesen Inhalt retten, und zwar auf den Stack! Zusammengefaßt richten wir daher einen Stackrahmen ein mit:

```
push    bp
mov     bp,sp
sub     sp,xx
```

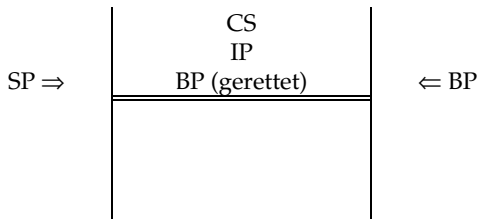
Uns stehen nun xx Bytes privaten Stacks zur Verfügung! Im Schaubild sieht das dann so aus:



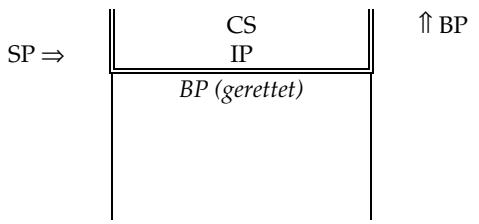
In diesem Fall wurde angenommen, daß die Sequenz von eben innerhalb einer Routine ausgeführt wurde, die mittels eines FAR CALL-Befehls angesprungen wurde. Daher liegt eine vollständige Rücksprungadresse auf dem Stack in dem Bereich, der dem rufenden Programm gehört!

ACHTUNG Werden innerhalb der Routine PUSH-/POP-Befehle ausgeführt, so wird der Inhalt von SP verändert. Diese Befehle verändern somit auch den Stackrahmen!

Was müssen wir nun tun, wenn wir den Stackrahmen entfernen wollen, z.B. vor Beendigung der Routine, wenn wir den lokalen Bereich nicht mehr benötigen? Wir brauchen im Prinzip nur zwei Register auf den ursprünglichen Inhalt zurückzusetzen: BP und SP, da nur sie den Rahmen definieren! Der alte Inhalt von SP aber steht in BP. Daher brauchen wir nur diesen Wert zu kopieren: *mov sp, bp*.



Hierdurch haben wir den neuen Rahmen entfernt, den eventuell vorhandenen, alten der aufrufenden Routine aber noch nicht wiederhergestellt! Dies erfolgt durch die Restauration des BP-Registers mit dem geretteten Inhalt durch *pop bp*:



Die vollständige Sequenz der Auflösung eines Stackrahmens lautet somit:

```
mov     sp,bp
pop     bp
```

Wie kommen wir jedoch an unsere lokalen Daten heran? Dazu gibt es prinzipiell zwei Möglichkeiten: über SP oder über BP! Denken wir ein wenig nach. SP scheint auf den ersten Blick bestechend, da man nur eine Konstante zum Inhalt von SP addieren muß, um an jede beliebige Position in Stackrahmen zu kommen. Aber nur auf den ersten Blick! Denn der Inhalt von SP wird durch PUSH/POP verändert. Man müßte sich also jeweils merken, wie viele PUSHs/POPs seit Einrich-

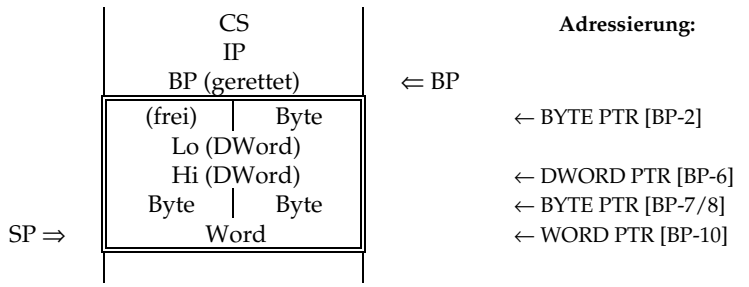
tung des Rahmens erfolgten und neben der Konstantenaddition auch anhand dieser Zahl die Korrektur vornehmen. Das ist wenig effizient.

Wenn wir dagegen BP verwenden, so können wir auch eine Konstante benutzen. Diese muß lediglich von BP abgezogen werden, was keine Probleme bereiten sollte. Im Gegensatz zu SP wird BP jedoch nicht verändert, außer man löst den Stackrahmen auf. Dann aber sind auch alle lokalen Daten undefiniert. Somit enfällt die Notwendigkeit zum Datenzugriff.

Dies haben auch die Prozessorkonstrukteure erkannt und daher eine besonders einfache Art der Adressierung implementiert. Man kann bei der *indirekten Adressierung* über BP zusätzlich eine Konstante angeben, die subtrahiert wird. Wenn also BP auf den oberen Teil des Rahmens zeigt, so zeigt BP-2 auf das Wort direkt darunter. Somit ist dessen Inhalt über *mov ax, [BP-2]* auslesbar, während man mit *mov [BP-2], cx* Daten dort ablegen kann.

ACHTUNG

Zwei Dinge müssen berücksichtigt werden: Das zu BP gehörige Segment ist SS. Die MOV-Befehle müßten also eigentlich *mov ax, SS:[BP-2]* und *mov SS:[BP-2], cx* lauten. Dies ist aber nicht nötig, da indirekte Adressierungen mit BP immer SS verwenden, ein Segmentpräfix kann hier also unberücksichtigt bleiben. Zweitens muß beachtet werden, daß der Stack wortorientiert ist! Bytes lassen sich zwar auf ihm mit *mov [BP-2], al* ablegen und holen. Sie besetzen aber immer ein Wort, wenn nicht unmittelbar zwei Bytes hintereinander definiert werden. Ein typischer Stackrahmen mit lokalen Variablen und Adressierung könnte also folgendermaßen aussehen:



TIP

Erleichtern Sie sich die Arbeit! Anstatt mit BYTE PTR [BP-xx] hantieren zu müssen, nutzen Sie die Möglichkeit, Namen zu vergeben. Wenn Sie nämlich z.B. folgende Anweisungen verwenden:

```
Status   EQU BYTE PTR [BP-2]
Adresse  EQU DWORD PTR [BP-6]
```

```
AnOffset EQU WORD PTR [BP-6]
ASegment EQU WORD PTR [BP-4]
```

ist der Zugriff auf lokale Daten kein Problem mehr:

```
        push  bp
        mov   sp,bp
        sub   sp,6
        Status   EQU BYTE PTR [BP-2]
        Adresse  EQU DWORD PTR [BP-6]
        AnOffset EQU WORD PTR [BP-6]
        ASegment EQU WORD PTR [BP-4]
        mov   al,[Status]
        cmp   al,1
        jz    L1
        lds   si,[Adresse]
        jmp   L2
L1:     mov   si,[AnOffset]
        mov   ds,[ASegment]
L2:
```

25.2 Parameter für Routinen

Die eigentliche Funktionalität von Routinen nutzt man nur aus, wenn man ihnen Parameter übergeben kann.

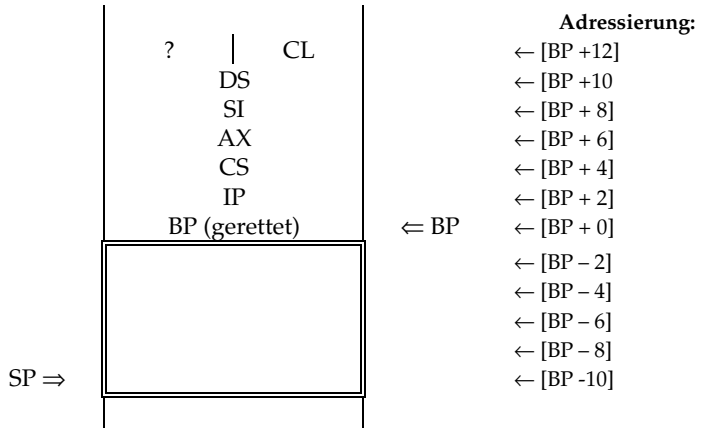
Da, wenn wir an Hochsprachen denken, Parameter ganz spezifisch für Routinen gedacht sind, müssen sie nicht global definiert sein. Wohlgemerkt – nicht, daß sie es nicht sein dürfen! Aber sie müssen es nicht sein. Oder gibt es einen zwingenden Grund, bei der Turbo-Pascal-Prozedur *GotoXY* die Koordinaten als Variablen *by reference* übergeben zu müssen? Also brauchen die Parameter für Routinen auch nicht global definiert zu werden. Wann immer aber etwas nicht global ist, ist es lokal, und bei dem Begriff *lokale Daten* denken wir gleich an den Stack!

Zwar könnten wir die Parameter auch über die Register des Prozessors übergeben. Tatsächlich gibt es auch mit den Interrupts Abarten von Routinen, die genau das tun. Ebenfalls bieten einige Hochsprachen, wie z.B. *C++*, die Möglichkeit, im Rahmen der Codeoptimierung diese Methode zu nutzen.

Allerdings haben wir, wenn wir berücksichtigen, daß einige Register mit speziellen Aufgaben dafür nicht mehr in Frage kommen, keine große Auswahl mehr: AX, BX, CX, DX; eingeschränkt vielleicht noch DI und SI sowie ES. SS, SP und BP aber sind für den Stack reserviert

und DS für unsere globalen Daten. CS entfällt ebenso wie IP. Also könnten 4 bis 6, höchstens 7 Worte an Parametern auf diese Weise übergeben werden.

Parameter werden daher tatsächlich von der aufrufenden Routine auf den Stack gelegt, bevor diese den CALL-Befehl ausführt. Wie also wird der Stack wohl aussehen, wenn Parameter mit im Spiel sind?



Über der Adresse und dem gepushten Inhalt von BP findet die Routine die übergebenen Parameter. Ganz logischerweise können sie analog der Adressierung lokaler Variablen auch über BP adressiert werden, wobei nun jedoch der Offset addiert werden muß. Auch für Parameter gilt wie bei lokalen Daten, daß der Stack wortorientiert ist! Auch hier können Sie zur Vereinfachung Namen vergeben.

HINWEIS

Parameter und lokale Variablen werden über das *Base-Pointer-Register* adressiert. Parameter können mit Hilfe der Addition eines Offsets, lokale Daten nach Subtraktion eines Offsets vom Inhalt von BP angesprochen werden.

ACHTUNG

Während die Lage der lokalen Variablen zum BP-Inhalt immer konstant ist – sie beginnen immer bei *[bp-2]* –, ist der notwendige Offset für Parameter davon abhängig, ob die aufgerufene Routine *near* oder *far* angesprungen wurde. Ist die Routine *near*, so sichert der aufrufende CALL-Befehl nur den Offset der Rücksprungadresse auf den Stack. Die Parameter der Routine beginnen damit bei *[bp+4]*. Andernfalls legt CALL die vollständige Adresse, also CS:IP, auf den Stack, und zwar CS »über« IP. In diesem Fall beginnen die Parameter an der Position *[bp+6]*.

Im obigen Fall wurde ganz offensichtlich zunächst das CL-Register auf dem Stack abgelegt, weil es an der physikalisch »obersten« bzw., vom Stack aus betrachtet, »untersten« Adresse steht, also am »ältesten« ist

(denken Sie bitte noch an die Stalagtiten!). Dies erfolgte mit PUSH. Da aber der PUSH-Befehl nur Worte pushen kann, wird CX insgesamt verwendet. Das obere Byte des Wertes auf dem Stack, an dem der Inhalt von CL steht, ist damit undefiniert! Anschließend wurde der Inhalt von SI, dann von DS und schließlich von AX auf den Stack gebracht, bevor die Routine gerufen wurde. Die Sequenz lautet somit:

```
push    cx
push    si
push    ds
push    ax
call    ...
```

26 Assembler und Pascal

Nachdem wir nun genügend Grundlagen kennen, wie man in Assembler programmiert, soll die Zusammenarbeit von Assemblermodulen mit Hochsprachen geklärt werden. Beginnen wir dazu mit Pascal, da Pascal ja ursprünglich eine Lehrsprache war, die Studenten das Programmieren näherbringen sollte. Somit kann man erwarten, daß die Assemblereinbindung hier auch besonders einfach ist.

Das ist auch so! Als wir in einem vorangegangenen Kapitel auf Prozeduren und Funktionen zu sprechen gekommen sind, haben wir festgestellt, daß es doch notwendig ist, bestimmte Regeln zu schaffen und einzuhalten. So kann ein Funktionsergebnis nur dann von einer Routine an den rufenden Programmteil übergeben werden, wenn eine entsprechende Schnittstelle geschaffen wurde.

Daher klären wir zuerst, in welcher Weise Pascal Parameter an Pascal-Routinen übergibt, um festzustellen, welche Regeln der Parameterübergabe bestehen. Genauso untersuchen wir dann, auf welche Weise Funktionsergebnisse zurückgegeben werden. Schließlich betrachten wir die notwendigen Verwaltungsaufgaben!

26.1 Parameter in Pascal

Kompilieren wir einfach einmal ein kleines Pascal-Programm, das einer Funktion einen Parameter übergibt, also z.B.:

```
var j : Byte;

function Test(i:Byte):byte;
begin
```

```
  Test := i;
end;

begin
  j := Test(2);
end.
```

Der Compiler macht hieraus:

```
0C58:0000 55            PUSH    BP
     0001 89E5          MOV     BP,SP
     0003 83EC02        SUB     SP,+02
     0006 8A4604        MOV     AL,[BP+04]
     0009 8846FF        MOV     [BP-01],AL
     000C 8A46FF        MOV     AL,[BP-01]
     000F 89EC          MOV     SP,BP
     0011 5D            POP     BP
     0012 C20200        RET     0002
     0015 9A00005B0C    CALL    0C5B:0000
     001A 55            PUSH    BP
     001B 89E5          MOV     BP,SP
     001D B002          MOV     AL,02
     001F 50            PUSH    AX
     0020 E8DDFF        CALL    0000
     0023 A25000        MOV     [0050],AL
     0026 5D            POP     BP
     0027 31C0          XOR     AX,AX
     0029 9A16015B0C    CALL    0C5B:0116
```

An $0C58:0000 beginnt offensichtlich die Funktion *Test*. Hier wird zunächst ein Stackrahmen eingerichtet und eine lokale Variable erzeugt. Der Parameter wird an der Stelle [BP+4] gefunden, da die Routine *near* ist, somit an [BP+2] nur der Offset der Rücksprungadresse steht und an [BP+0] der alte BP-Inhalt. Mehr soll uns zunächst an der Funktion nicht interessieren.

Das Hauptprogramm, das an CS:0015 beginnt, richtet nach dem Aufruf einer Routine, die uns nicht interessieren soll, zunächst ebenfalls einen Stackrahmen ein und schiebt dann die Konstante 2 auf den Stack. Anschließend wird die Funktion aufgerufen.

Doch was passiert, wenn mehrere Parameter übergeben werden? Ein Beispiel:

```
var j : Byte;

function Test(i,k:Byte):byte;
begin
```

```
  Test := i;
end;

begin
  j := Test(1,2);
end.
```

Im Vergleich zu eben hat sich lediglich geändert, daß die Funktion *Test* nun zwei Parameter erwartet und diese auch übergeben bekommt. Das Compilat sieht dann so aus:

```
CS : 0000 55            PUSH    BP
     0001 89E5          MOV     BP,SP
     0003 83EC02        SUB     SP,+02
     0006 8A4606        MOV     AL,[BP+06]
     0009 8846FF        MOV     [BP-01],AL
     000C 8A46FF        MOV     AL,[BP-01]
     000F 89EC          MOV     SP,BP
     0011 5D            POP     BP
     0012 C20400        RET     0004
     0015 9A0000D30D    CALL    0DD3:0000
     001A 55            PUSH    BP
     001B 89E5          MOV     BP,SP
     001D B001          MOV     AL,01
     001F 50            PUSH    AX
     0020 B002          MOV     AL,02
     0022 50            PUSH    AX
     0023 E8DAFF        CALL    0000
     0026 A25000        MOV     [0050],AL
     0029 5D            POP     BP
     002A 31C0          XOR     AX,AX
     002C 9A1601D30D    CALL    0DD3:0116
```

Verglichen mit dem Compilat von oben, stellen wir fest, daß nun in der Funktion an der Adresse CS:0006 die Variable *i* an der Stackposition [BP+6] gefunden wird und nicht an [BP+4]. Das bedeutet aber, daß der Parameter *k* an [BP+4] stehen muß, da sich ja weiter nichts geändert hat. Dies wird dadurch bestätigt, daß das Hauptprogramm an der Adresse CS:001D zuerst den Wert 1 (also *i*) auf den Stack schiebt, bevor dann 2 repräsentativ für die Variable *k* abgelegt wird. Offensichtlich arbeitet sich also der Compiler von links nach rechts durch die Parameter.

HINWEIS Pascal arbeitet die Parameter, die Routinen übergeben werden, von links nach rechts ab. Das heißt, daß der erste Parameter der Parameterliste einer Routine an der physikalisch obersten Position auf dem Stack liegt, während der letzte an der physikalisch untersten Stelle liegt. Um auf die Betrachtung aus Sicht des Stacks zurückzukommen: Der Parameter, der am

weitesten links steht, ist der »älteste«, weshalb er unten liegt, während der Parameter, der am weitesten rechts steht, als »jüngster« oben auf dem Stack liegt.

Wenn Sie nun für alle möglichen Parametertypen und -arten solche Miniprogramme entwickeln und das Compilat analysieren, so kommen Sie zu folgenden Vereinbarungen unter Pascal:

- Parameter vom Typ *boolean*, *byte*, *char* und *shortint*, die also nur ein Byte belegen, werden als Wort auf dem wortorientierten Stack abgelegt, wobei das »obere« Byte undefiniert ist.
- Parameter vom Typ *integer* und *word* werden ebenfalls über den Stack übergeben. Sie belegen ebenfalls ein Wort.
- Auch Parameter größerer Länge, also *longints* (4 Bytes), werden auf dem Stack abgelegt. Das jeweils »obere« Wort wird hierbei gemäß Intel-Konvention zuerst abgelegt, so daß es sich an physikalisch höherer Stackadresse befindet.
- Parameter vom Typ *real*, also Fließkommazahlen, die nicht coprozessorkompatibel sind, werden auch über den Stack übergeben. Solche Daten belegen 6 Bytes, so daß das oberste Wort wiederum an oberster physikalischer Stelle des Stacks liegt.
- Ebenfalls über den Stack übergeben werden alle Coprozessordaten, also Parameter vom Typ *single*, *double*, *extended* und *comp*. Achtung: bei Turbo Pascal 4.0 (und nur dort) werden diese Daten über den Coprozessorstack übergeben!
- Zeiger jeglicher Art, also Parameter vom Typ *pointer* oder jedem anderen typisierten Zeigertypen, werden wiederum über den Stack *mit 2 Worten* übergeben, wobei das *Wort* an der physikalisch höheren Stackadresse den Segmentanteil beinhaltet, das »untere« *Wort* den Offsetanteil. Das gleiche gilt für Parameter, die nicht als Wertparameter (*call by value*) übergeben werden, sondern als Variablenparameter (*call by reference*; in Pascal durch das Präfix *var* in der Parameterliste gekennzeichnet).
- Zeichenketten vom Typ String werden nicht über den Stack übergeben, selbst dann nicht, wenn sie nur ein Zeichen enthalten! Statt dessen wird ein Zeiger (siehe letzter Punkt) auf den String auf dem Stack abgelegt.

ACHTUNG

Bei Strings ist es also unerheblich, ob der Parameter als Wertparameter oder als Variablenparameter übergeben wird. In den beiden folgenden Routinen wird jeweils der gleiche Zeiger übergeben:

```
Routine1(S : string);
Routine2(var S : string);
```

Da der Compiler keine Möglichkeit hat zu überprüfen, ob ein String in einer Routine verändert wurde, im Fall der ersten der beiden Routinen oben aber davon ausgeht, daß dies nicht erfolgt, kann es hier zu Problemen kommen, falls es dennoch geschieht! Es liegt in der Verantwortung des Programmierers, Strings, die als Wertparameter übergeben wurden, nicht zu verändern. Sollen sie verändert werden, so muß eine lokale Kopie angefertigt werden.

TIP Übergeben Sie daher, um Probleme zu vermeiden, Strings immer als Variablenparameter, selbst wenn das nicht nötig wäre.

- Daten von Typ *record* und *array* werden unterschiedlich behandelt. Falls sie genau 1, 2 oder 4 Bytes groß sind, werden sie wie Daten vom Typ *byte*, *word* oder *longint* behandelt und direkt über den Stack übergeben. Andernfalls wird ein Zeiger auf den *record* oder das *array* übergeben.

ACHTUNG Auch hier ergibt sich, wenn ein Zeiger übergeben wird, das gleiche Problem wie bei Strings! Solche Daten dürfen nicht verändert werden und es müssen ggf. lokale Kopien angefertigt werden, falls sie als Wertparameter übergeben werden.

- Wertparameter vom Typ *set* werden analog zu Strings niemals über den Stack übergeben. Vielmehr wird auch hier ein Zeiger übergeben, womit sich das gleiche Problem wie bei Strings ergibt. Das gilt, wie gesagt, immer – mit einer einzigen Ausnahme: *set of boolean* wird wie ein *record* behandelt, wenn bis zu 8, 16 oder 32 boolesche Werte in der Menge zusammengefaßt sind, da diese dann nur 1, 2 oder 4 Bytes belegen!

Demonstrieren wir das noch kurz anhand eines Listings und des Compilats von Turbo Pascal:

```
type  pWord  = ^Word;
      Rec1   = Record B1, B2 : Byte; end;
      Rec2   = Record W1, W2, W3 : Word; end;
      Feld   = Array[1..60] of Byte;
      Menge1 = Set of Boolean;
      Menge2 = Set of Byte;


var   aByte    : Byte;
      aWord    : Word;
      aLongInt : LongInt;
      aString  : String;
      aReal    : Real;
      aDouble  : Double;
```

```
      aPointer : Pointer;
      anotherP : pWord;
      aRecord  : Rec1;
      anotherR : Rec2;
      anArray  : Feld;
      aSet     : Menge1;
      anotherS : Menge2;

procedure Test(B  : Byte; var B_ : Byte;
               W  : Word; var W_ : Word;
               L  : LongInt; var L_ : LongInt;
               S  : String; var S_ : String;
               R  : Real; var R_ : Real;
               D  : Double; var D_ : Double;
               P1 : Pointer; var P1_: Pointer;
               P2 : pWord; var P2_: pWord;
               R1 : Rec1; var R1_: Rec1;
               R2 : Rec2; var R2_: Rec2;
               A  : Feld; var A_ : Feld;
               M  : Menge1; var M_ : Menge1;
               N  : Menge2; var N_ : Menge2);
begin
end;


begin
  aByte := 1;
  aWord := 2;
  aLongInt := 3;
  aString := 'Ein String';
  aReal := 4.0;
  aDouble := 5.0;
  aPointer := pointer(aByte);
  anotherP := @aWord;
  aRecord.B1 := 6; aRecord.B2 := 7;
  anotherR.W1 := 8; anotherR.W2 := 9; anotherR.W3 := 10;

  Test( aByte, aByte,
        aWord, aWord,
        aLongInt, aLongInt,
        aString, aString,
        aReal, aReal,
        aDouble, aDouble,
        aPointer, aPointer,
```

```
        anotherP, anotherP,
        aRecord, aRecord,
        anotherR, anotherR,
        anArray, anArray,
        aSet, aSet);

end.
```

In diesem Programm werden ein paar Typen deklariert und einige Variablen angelegt. Die Routine ist eine einfache Prozedur, der mehrere Parameter übergeben werden. Zum Vergleich und des besseren Überblicks wegen wurde hierbei die Zeile mit den Parametern umbrochen und in jeder Zeile die betreffende Variable einmal als Wertparameter (*call by value*) und einmal als Variablenparameter (*call by reference*) übergeben. Die Prozedur selbst tut gar nichts! Sie ist eine Dummy-Prozedur. Das Hauptprogramm weist nun den Variablen einige Werte zu und ruft dann die Prozedur auf. Wenn wir das Compilat dieser Wertzuweisungen einmal außer acht lassen, so erzeugt der Compiler folgenden Code:

```
CS : 00D7 A05000     MOV     AL,[0050]  aByte
     00DA 50         PUSH    AX         wortweise!

     00DB BF5000     MOV     DI,0050    var aByte
     00DE 1E         PUSH    DS
     00DF 57         PUSH    DI

     00E0 FF365200   PUSH    [0052]     aWord

     00E4 BF5200     MOV     DI,0052    var aWord
     00E7 1E         PUSH    DS
     00E8 57         PUSH    DI

     00E9 FF365600   PUSH    [0056]     aLongInt (hi)
     00ED FF365400   PUSH    [0054]              (lo)

     00F1 BF5400     MOV     DI,0054    var aLongInt
     00F4 1E         PUSH    DS
     00F5 57         PUSH    DI

     00F6 BF5800     MOV     DI,0058    aString
     00F9 1E         PUSH    DS         Zeiger!
     00FA 57         PUSH    DI

     00FB BF5800     MOV     DI,0058    var aString
     00FE 1E         PUSH    DS
```

```
00FF 57          PUSH    DI

0100 FF365C01    PUSH    [015C]     aReal (hi)
0104 FF365A01    PUSH    [015A]           (med)
0108 FF365801    PUSH    [0158]           (lo)

010C BF5801      MOV     DI,0158    var aReal
010F 1E          PUSH    DS
0110 57          PUSH    DI

0111 FF366401    PUSH    [0164]     aDouble (hi)
0115 FF366201    PUSH    [0162]             (med1)
0119 FF366001    PUSH    [0160]             (med2)
011D FF365E01    PUSH    [015E]             (lo)

0121 BF5E01      MOV     DI,015E    var aDouble
0124 1E          PUSH    DS
0125 57          PUSH    DI

0126 FF366801    PUSH    [0168]     aPointer (seg)
012A FF366601    PUSH    [0166]              (ofs)

012E BF6601      MOV     DI,0166    var aPointer
0131 1E          PUSH    DS
0132 57          PUSH    DI

0133 FF366C01    PUSH    [016C]     anotherP (seg)
0137 FF366A01    PUSH    [016A]              (ofs)

013B BF6A01      MOV     DI,016A    var anotherP
013E 1E          PUSH    DS
013F 57          PUSH    DI

0140 FF366E01    PUSH    [016E]     aRecord 2 Bytes!

0144 BF6E01      MOV     DI,016E    var aRecord
0147 1E          PUSH    DS
0148 57          PUSH    DI

0149 BF7001      MOV     DI,0170    anotherR 6 Bytes
014C 1E          PUSH    DS
014D 57          PUSH    DI

014E BF7001      MOV     DI,0170    var anotherR
0151 1E          PUSH    DS
```

```
0152 57          PUSH    DI

0153 BF7601      MOV     DI,0176    anArray
0156 1E          PUSH    DS
0157 57          PUSH    DI

0158 BF7601      MOV     DI,0176    var anArray
015B 1E          PUSH    DS
015C 57          PUSH    DI

015D A0B201      MOV     AL,[01B2]  aSet
0160 50          PUSH    AX         Boolesche Menge

0161 BFB201      MOV     DI,01B2    var aSet
0164 1E          PUSH    DS
0165 57          PUSH    DI

0166 BFB201      MOV     DI,01B4    anotherS
0169 1E          PUSH    DS
016A 57          PUSH    DI

016B BFB201      MOV     DI,01B4    var anotherS
016E 1E          PUSH    DS
016F 57          PUSH    DI

0170 E897FE      CALL    0000
```

Beachten Sie bitte hierbei die fett gedruckten Anmerkungen: Sie kennzeichnen die Ablage der Komponenten eines Datums in umgekehrter Reihenfolge gemäß Intelkonvention sowie die grundsätzliche Übergabe von Zeigern bei Strings. Auch die unterschiedliche Behandlung von Records mit Byte-, Wort- oder Doppelwortgröße ist zu erkennen.

Um auf die Verantwortlichkeit des Programmierers zurückzukommen, wenn Strings *by value* übergeben werden, so löst das unsere Dummy-Prozedur, die ja nichts anderes tut:

```
→    CS:0000 55      PUSH    BP           Stackrahmen!
→    0001 89E5       MOV     BP,SP
     0003 81EC6201   SUB     SP,0162      304 Bytes lokal
     0007 8CD3       MOV     BX,SS
     0009 8EC3       MOV     ES,BX
     000B 8CDB       MOV     BX,DS
     000D FC         CLD
     000E 8DBE00FF   LEA     DI,[BP+FF00] Ziel:Stack
     0012 C57652     LDS     SI,[BP+52]   Quelle:aString
```

```
     0015 AC         LODSB               Stringgröße
     0016 AA         STOSB
     0017 91         XCHG    CX,AX
     0018 30ED       XOR     CH,CH
     001A F3         REPZ
     001B A4         MOVSB               lokal kopieren!
     001C 8DBEFAFE   LEA     DI,[BP+FEFA] Ziel:Stack
     0020 C5761E     LDS     SI,[BP+1E]   Quelle:anotherR
     0023 B90600     MOV     CX,0006      6 Bytes Größe
     0026 F3         REPZ
     0027 A4         MOVSB               lokal kopieren!
     0028 8DBEBEFE   LEA     DI,[BP+FEBE] Ziel:Stack
     002C C57616     LDS     SI,[BP+16]   Quelle:anArray
     002F B93C00     MOV     CX,003C      60 Bytes Größe
     0032 F3         REPZ
     0033 A4         MOVSB               lokal kopieren!
     0034 8DBE9EFE   LEA     DI,[BP+FE9E] Ziel:Stack
     0038 C57608     LDS     SI,[BP+08]   Quelle:anotherS
     003B B92000     MOV     CX,0020      32 Bytes Größe
     003E F3         REPZ
     003F A4         MOVSB               lokal kopieren!
     0040 8EDB       MOV     DS,BX
→    0042 89EC       MOV     SP,BP        Stackrahmen weg
→    0044 5D         POP     BP
→    0045 C26600     RET     0066
```

Eine »normale« Dummy-Prozedur würde lediglich die mit einem Pfeil markierten Zeilen beinhalten! Man sieht, daß sich Pascal an seine eigenen Vereinbarungen hält und ggf. lokale Kopien erzeugt.

26.2 Funktionswerte in Pascal

Kommen wir nun noch zur Übergabe von Funktionswerten. Dazu betrachten wir noch einmal unser kleines Programm:

```
var j : Byte;

function Test(i:Byte):byte;
begin
  Test := i;
end;

begin
  j := Test(2);
end.
```

Wenn wir uns das Compilat noch einmal genauer anschauen, so fällt uns auf, daß die Routine zunächst den Stackrahmen und dann Platz für eine lokale Variable erzeugt. Schließlich wird der übergebene Parameter zunächst in AL kopiert, um in der lokalen Variablen abgelegt zu werden. Daß dann im nächsten Schritt diese Variable wieder ausgelesen wird, ist ein etwas verwunderlicher Aspekt, den ich an anderer Stelle kommentiere (»›Optimierte‹ Compiler oder ›optimiertes‹ Denken«). Nur soviel an dieser Stelle: Dieser Mechanismus sorgt dafür, daß in Pascalprogrammen nicht nur mehrmals, sondern auch an beliebiger Stelle in der Routine das Funktionsergebnis deklariert werden kann.

Ich möchte nun Ihr Augenmerk lediglich auf die letzten vier Zeilen der Routine lenken! Die lokale Variable wird ausgelesen und der Stackrahmen entfernt. Schließlich wird mit RET der Rücksprung in den rufenden Teil eingeleitet. Aber etwas merkwürdig ist dieser RET-Befehl schon: Er hat einen Parameter, in unserem Fall den Wert 2.

```
0C58:0000 55            PUSH    BP
     0001 89E5          MOV     BP,SP
     0003 83EC02        SUB     SP,+02
     0006 8A4604        MOV     AL,[BP+04]
     0009 8846FF        MOV     [BP-01],AL
     000C 8A46FF        MOV     AL,[BP-01]
     000F 89EC          MOV     SP,BP
     0011 5D            POP     BP
     0012 C20200        RET     0002
```

Dieser Parameter gibt an, daß der Prozessor nach dem Rücksprung zwei Bytes vom Stack entfernen soll. Diese zwei Bytes sind genau diejenigen, die »über« der Rücksprungadresse liegen, da sie ja nach dem Rücksprung entfernt werden! Die zwei Bytes über der Adresse sind aber diejenigen, die der rufende Teil dort angesiedelt hat, als er vor dem CALL-Befehl den Parameter auf den Stack gelegt hat.

HINWEIS In Pascal obliegt die Verantwortung, den Stack von Übergabewerten zu befreien, der aufgerufenen Routine! Sie hat dafür zu sorgen, daß der Stack in einem Zustand vorgefunden wird, wie er vor der Übergabe der Parameter geherrscht hat!

Das heißt, daß der RET-Befehl genau die Anzahl von Bytes als Parameter erhält, die von den Übergabewerten zusammen benutzt werden. Für die zuvor besprochene Routine mit den vielen Parametern sind dies 102 Bytes, weshalb der RET-Befehl den Parameter ($)66 erhält.

Auch hier möchte ich Ihnen und mir ersparen, herauszufinden, in welchen Registern nun unterschiedliche Funktionswerte zurückgegeben werden.

Ich resümiere einfach:

- Funktionsergebnisse vom Typ *boolean*, *byte*, *char* und *shortint*, die also nur ein Byte belegen, werden in AL zurückgegeben.
- Funktionsergebnisse vom Typ *integer* und *word* werden in AX zurückgegeben.
- *Longints* als Ergebnis werden in DX:AX übergeben, wobei in DX das höherwertige und in AX das niederwertige Wort steht.
- Auch *reals*, also Fließkommazahlen, die nicht coprozessorkompatibel sind, werden in Registern übergeben: DX:BX:AX, wobei wiederum das höherwertige Wort in DX, das niederwertige in AX steht.
- Funktionsergebnisse vom Typ *single*, *double*, *extended* und *comp* werden über den TOS des Coprozessorstacks übergeben.
- Zeiger jeglicher Art, also Parameter vom Typ *pointer* oder jedem anderen typisierten Zeigertyp, werden in DX:AX zurückgegeben, wobei DX den Segment- und AX den Offsetanteil enthält.
- Zeichenketten vom Typ *String* werden in temporären Variablen zurückgegeben, deren Adresse über den *Stack* der Routine übergeben wurde.

ACHTUNG

Diese Adresse darf durch den RET-Befehl nicht entfernt werden. Sie wird vom rufenden Teil benötigt, um auf den temporären Bereich wieder zugreifen zu können. Ein Beispiel:

```
var S : String;

function TestString : String;
begin
  TestString := 'Dies ist der Teststring!';
end;

begin
  S := TestString;
end.
```

Das Compilat sieht dann so aus:

```
cs:0019 55            PUSH    BP
   001A 89E5          MOV     BP,SP
   001C BF0000        MOV     DI,0000
   001F 0E            PUSH    CS
   0020 57            PUSH    DI
   0021 C47E04        LES     DI,[BP+04]
   0024 06            PUSH    ES
   0025 57            PUSH    DI
```

```
0026 B8FF00        MOV     AX,00FF
0029 50            PUSH    AX
002A 9A5804D812    CALL    12D8:0458
002F 5D            POP     BP
0030 C3            RET
```

Wie Sie sehen können, lädt die Funktion *TestString* in ES:DI eine Adresse, die sie über den Stack übergeben bekommt. Bei dieser Adresse handelt es sich um die angesprochene Adresse des temporären Speicherbereichs. Der Aufruf der Routine an der Adresse $12D0:0450 ist der Aufruf einer Turbo-Pascal-Systemroutine, die für das Kopieren von Strings verantwortlich ist und den String 'Dies ist der Teststring!' in diesen temporären Bereich kopiert. Das eigentlich Wichtige ist hierbei, daß der RET-Befehl keinen Parameter hat, also keine Stackbereinigung durchführt! Dies ist auch logisch, da *TestString* formal kein Parameter übergeben wird! Die Adresse des temporären Bereichs ist also ein »versteckter« Parameter, den nur derjenige entfernen darf, der ihn angelegt hat. Sie bleibt daher auf dem Stack als Adresse des Funktionsergebnisses.

HINWEIS

Falls der Routine andere Parameter zusätzlich übergeben worden wären, so läge diese Adresse physikalisch »über« allen übergebenen Parametern, so daß der RET-Befehl die anderen Parameter entfernen kann (und muß!). Der RET-Befehl erhält dann als Wert die Gesamtgröße aller übergebenen Parameter – ohne die Adresse!

Verantwortlich für den temporären Bereich ist der *rufende* Teil. Dies hat auch ganz logische Gründe:

```
0031 9A0000D812    CALL    12D8:0000
0036 55            PUSH    BP
0037 89E5          MOV     BP,SP
0039 81EC0001      SUB     SP,0100
003D 8DBE00FF      LEA     DI,[BP+FF00]
0041 16            PUSH    SS
0042 57            PUSH    DI
0043 E8D3FF        CALL    0019
0046 BF5000        MOV     DI,0050
0049 1E            PUSH    DS
004A 57            PUSH    DI
004B B8FF00        MOV     AX,00FF
004E 50            PUSH    AX
004F 9A5804D812    CALL    12D8:0458
0054 89EC          MOV     SP,BP
0056 5D            POP     BP
0057 31C0          XOR     AX,AX
0059 9A1601D812    CALL    12D8:0116
```

Nach dem Aufruf einer uns nicht interessierenden Systemroutine erzeugt das Hauptprogramm wie immer einen Stackrahmen. Anders als sonst kreiert jedoch hier das Hauptprogramm eine lokale Variable, indem es von SP $100 = 256 Bytes abzieht – genau so viel, wie ein String maximal groß sein kann! Die Adresse dieses temporären Bereichs wird nun auf den Stack gelegt und die Funktion dann aufgerufen. Nach der Rückkehr wird nun eine weitere Adresse auf den Stack gelegt: nämlich die der Variablen *S*, die ja das Funktionsergebnis aufnehmen soll! Schließlich wird wieder die Systemroutine aufgerufen, die Strings kopiert. Hier liegt der Grund dafür, daß die übergebene Adresse des temporären Bereichs nicht gelöscht werden darf: Die Systemroutine erwartet zwei Adressen, Quellstring und Zielstring! Sie entfernt dann beide Adressen vom Stack!

Würden Sie nun Ihrerseits in *TestString* die temporäre Adresse entfernen, so entfernte auch die Kopierroutine etwas – und das wäre fatal, ganz abgesehen davon, daß sie ja keine Quelladresse hätte, also irgendwelche zufälligen Werte vom Stack als Quelle interpretierte. Probleme sind damit vorprogrammiert!

- Daten von Typ *record, array* und *set* können nicht als Funktionsergebnis zurückgegeben werden.

Professional Pascal

Achtung! Unterschiedliche Pascal-Compiler halten sich an unterschiedliche Konventionen. So zeigt der *Professional-Pascal-Compiler* von Microsoft ein auf den ersten Blick recht merkwürdiges Verhalten bei Funktionswerten vom Typ *real, real4* und *real8*, die ja unter Turbo Pascal den Typen *single* und *double* entsprechen.

Dieser Compiler legt nämlich einen (zusätzlichen) Parameter auf dem Stack ab, bevor die Funktion aufgerufen wird. *WhoAmI*, wie ich den seltsamen Parameter einmal nennen möchte, ist der Offset eines Zeigers, der auf einen temporären Stackbereich zeigt, auf dem das Funktionsergebnis abgelegt werden kann. Microsoft bezeichnet dies als *Long-return*-Methode und wendet diese nur bei den genannten Daten an. Man muß sich dies in etwa so vorstellen, als würde eine Parameterliste im Prozedurkopf auf diese Weise verlängert:

```
function FPFunction([VarList,]WhoAmI:NearPointer):Real8;
```

ACHTUNG

Damit ist bei diesen Datentypen Vorsicht angebracht, wenn Hochsprachenmodule »gemixt« werden sollen. So kann man davon ausgehen, daß die Sprachen der Firma Microsoft diese Eigenheit von Professional Pascal berücksichtigen (können), was auch der Fall ist, wie wir im Kapitel über C noch sehen werden. Mit Sicherheit jedoch werden Sie Probleme bekommen, wenn Sie solche Module in Turbo C++ einbinden wollen!

HINWEIS *MASM* und *TASM* dagegen verhalten sich friedlich! Denn welche und wie viele Parameter wem übergeben werden und warum, müssen *Sie* ja wissen und dem Assembler mitteilen! Somit liegt die Verantwortung, diesen »Zusatzparameter« zu berücksichtigen, bei *Ihnen*!

TIP Planen Sie daher rechtzeitig genug, für welche Pascal-Dialekte Sie Assemblermodule erstellen wollen (sollen, müssen). Entwickeln Sie dann die Module für den Dialekt, der möglichst in allen anderen Dialekten »enthalten« ist. Erst zum Schluß sollten Sie dann auf die Eigenheiten der einzelnen Dialekte eingehen. Sie ersparen sich auf diese Weise viel Mühe.

Quick Pascal Der andere Pascal-Compiler aus dem Haus Microsoft dagegen verhält sich ganz Turbo-freundlich. Sie werden wenig Probleme bekommen, Module dieser beiden Sprachen auszutauschen.

26.3 Einbindung von Assemblermodulen in Pascal-Programme

Nachdem wir nun wissen, wie Parameter an Routinen und Ergebnisse von Funktionen übergeben werden müssen, können wir an den letzten Punkt gehen: die Einbindung von Assemblermodulen in Pascal-Programme.

§ 1 *Die Definition und Implementation von Assemblerroutinen im Assemblermodul reicht nicht aus.* Sie müssen dem Assembler mitteilen, daß er in die zu erzeugende *Objektdatei* auch die Information aufnimmt, an welcher Adresse im Modul sich die zu importierende Routine befindet. Schließlich wissen Sie ja nun, daß alle Namen in Assembler nur Platzhalter für irgend etwas sind: Routinennamen repräsentieren eben Adressen. Sie erreichen dies jedoch ganz einfach: Irgendwo im Assemblerquelltext muß lediglich die Anweisung

```
PUBLIC Xyz
```

stehen. Dies veranlaßt den Assembler, die Adresse der Routine *Xyz* in eine Tabelle aufzunehmen, aus der sich der Hochsprachenteil bedient! Ohne diese Anweisung ist *Xyz* nur im Assemblermodul bekannt!

TIP Um den Überblick zu behalten, empfehle ich Ihnen, dies grundsätzlich unmittelbar vor der Deklaration der Routine zu tun:

```
PUBLIC ImportedRoutine
ImportedRoutine PROC FAR
          :
          :
          :
ImportedRoutine ENDP
```

HINWEIS

Dies gilt nicht nur für Routinen, sondern auch für Daten, die im Modul deklariert und exportiert (bzw. aus dem Blickwinkel der Hochsprache: importiert) werden sollen. Das bedeutet: Alles, was im Assemblermodul definiert und exportiert werden soll, muß mit PUBLIC markiert werden!

§ 2

Im Pascal-Modul muß die Routine mit einem Routinenrumpf angemeldet werden. Dies geschieht ganz analog zu *forward*-Deklarationen. Die Syntax ist hierbei die gleiche wie bei ganz »normalen« Pascal-Routinen, erweitert um das Schlüsselwort *external*:

```
Function Xyz(var X:Byte; Y,Z:Word):Boolean; External;
```

Auf diese Weise teilen Sie jedoch dem Compiler lediglich mit, daß er eine Routine (hier: Funktion) einbinden soll, die extern deklariert ist. *Eingebunden wird die Routine hierdurch noch nicht!* Die Angaben über die Variablen benötigt der Compiler, um, wie wir schon gesehen haben, die Parameter vor dem CALL-Befehl auf den Stack zu legen! Ebenso erzeugt er durch das Schlüsselwort *function* nach dem CALL-Befehl Code, der das Funktionsergebnis anhand des Funktionstyps weiterverarbeitet.

ACHTUNG

Beachten Sie an dieser Stelle, daß

- *Art*,
- *Anzahl* und
- *Typ* der Variablen sowie die
- *Reihenfolge* ihrer Deklaration korrekt angegeben werden!

Hier finden sich häufig Fehler, die dann zu Abstürzen oder instabil ablaufenden Programmen führen. Denken Sie daran, daß der Pascal-Compiler die Parameter von links nach rechts auf den Stack legt!

§ 3

Das eigentliche Einbinden des Assemblermoduls ist Aufgabe des Compilers. Daher erfolgt dies auch durch eine Compileranweisung:

```
{$L Xyz}
```

Achten Sie hierbei darauf, daß der Compiler die Objektdatei *XYZ.OBJ* auch tatsächlich findet! Ggf. müssen Sie den Dateinamen um einen Pfad erweitern. Die Namenserweiterung *.OBJ* benötigen Sie dagegen nicht, der Compiler nimmt dies automatisch an.

TIP

Auch hier empfehle ich Ihnen, die Assembleranweisung unmittelbar an die Deklaration anzuschließen. Dies können Sie, falls Sie mehrere Routinen eines Moduls importieren, nach dem Deklarationsblock tun:

```
Function  Xyz(var X:Byte; Y,Z:Word):Boolean; External;
Procedure Abc(A:String); External;
Function  Uvw:Byte; External;
{$L Modul}
```

In diesem Fall befinden sich alle Routinen im Assemblermodul MODUL.ASM bzw. in der hieraus erzeugten OBJ-Datei.

§ 4 *Vergewissern Sie sich, daß die einzubindenden Routinen in der richtigen Adressierungsart assembliert wurden!* Die meisten Schwierigkeiten beim Verwenden von Assemblerteilen bestehen darin, daß die Assemblerroutine nicht korrekt vom Hochsprachenmodul angesprungen wird. *Far*-deklarierte Assemblerroutinen (*Def PROC FAR*) müssen von der Hochsprache aus auch *far* angesprungen werden (P*rocedure Def; Far; External;* oder *{$F+} Procedure Def; External; {$F-}*).

ACHTUNG Ein gewaltiger Stolperstein ist hier die sonst sehr gute UNIT-Technologie verschiedener Pascal-Compiler! So sind Routinen in Units, die über den INTERFACE-Teil exportiert werden können, grundsätzlich *far*! Wollen Sie daher eine Assemblerroutine im Rahmen einer Unit exportieren, so muß diese Routine *immer* als *PROC FAR* deklariert werden! Anders verhält es sich mit Routinen, die innerhalb der Unit verwendet werden. Diese können (müssen aber nicht!) *near* angesprungen werden, weshalb sie als *PROC NEAR* deklariert werden können. Werden sie dagegen als *far* deklariert, so müssen sie auch *far* eingebunden werden!

TIP Wenn dieser Sachverhalt Sie verwirrt, dann denken Sie bitte an folgendes. Eine Unit belegt üblicherweise ein eigenes Segment. Damit werden alle Routinen einer Unit aus anderen Programmteilen grundsätzlich aus anderen Segmenten, also *far* angesprungen! Dies ist der Grund dafür, daß alle im INTERFACE-Teil angegebenen Routinen *immer far* sein müssen. Alle unit-internen Routinen dagegen, die nicht exportiert werden, sind, falls man nichts anderes vorgibt, immer *near*, da sie die Unit »niemals verlassen (können)«. Bei dieser Betrachtungsweise ist es vollkommen unerheblich, ob die Routinen selbst in der Hochsprache, also Pascal, oder als Assemblerroutine implementiert werden: Schließlich wird die Assemblerroutine ja in die Unit eingebunden, »verschmilzt« also mit dem Hochsprachencode (Sie können im Prinzip die Zeile, mit der das Assemblermodul eingebunden wird, als »*Makro*« ansehen, das an dieser Stelle den entsprechenden Code einfügt). Ein Beispiel:

```
Unit MyUnit;

INTERFACE

Procedure Abc(var A:Byte);
Function  Def:Word;
```

`Abc` und `Def` sind Routinen, die im INTERFACE-Teil der Unit auftauchen, also exportiert werden und somit von Programmteilen »außerhalb« der Unit angesprungen werden. Sie müssen also als *far* deklariert werden!

Bisher mußten Sie sich darüber keine Gedanken machen, da der Compiler dies automatisch berücksichtigt hat. Doch nun gilt es aufzupassen!

```
IMPLEMENTATION

Procedure Abc(var A:Byte); FAR; External;
Function  Ghi:Byte; External;
{$L MyModule}

Function  Jkl:Byte;
begin
  Jkl := 0;
end;

Function  Def:Word;
begin
  if Semaphore = 1 then Def := Ghi
  else Def := Jkl;
end;

end.
```

`Abc` ist eine Routine aus dem Assemblermodul und *muß* daher als *far* deklariert und gemäß §1 – §3 eingebunden werden, da sie aus der Unit exportiert wird. `Def` dagegen ist eine Hochsprachenroutine. Da auch sie im INTERFACE-Teil erscheint, erzeugt der Compiler selbst automatisch eine *far*-Adressierung. `Def` ruft nun ihrerseits die Assemblerroutine `Ghi` auf. Diese steht ebenfalls im Assemblermodul, befindet sich also »innerhalb« der Unit. Da sie nicht im INTERFACE-Teil erscheint, gibt es keinen zwingenden Grund, sie als *far* zu deklarieren, da sie nicht »von außen« angesprungen werden kann. Daher wurde sie (standardmäßig) als *near*-Routine deklariert und eingebunden! Das gleiche tut übrigens der Pascal-Compiler mit der Routine `Jkl`. Da sie nicht exportiert wird, erzeugt der Compiler hier *near*-deklarierten Routinencode.

ACHTUNG

Was für Units gilt, gilt nicht (immer) für Programme! Falls Sie nämlich eine Assemblerroutine in einem Pascal-Programm einbinden, so wird sie (wie üblich) in das entsprechende Segment aufgenommen! Handelt es sich nun um das Hauptprogramm, etwa

```
Program Test;

Procedure Abc; External;
{$L ABC}

begin
```

```
  :
  Abc;
  :
end.
```

so befindet sich `Abc` im gleichen Segment wie das gesamte Programm und muß daher entweder als *near* deklariert oder *far* eingebunden werden! Also: entweder im Assemblerteil »*Abc PROC NEAR*« und »*procedure Abc; external;*« im Hauptprogramm oder »*Abc PROC FAR*« im Assemblermodul und »*procedure Abc; Far; External;*« im Programm!

TIP Wenn Sie im Zweifel sind, ob Sie nun *near* oder *far* deklarieren müssen, wählen Sie besser grundsätzlich die *far*-Adressierung! Sie kostet zwar ein paar Bytes Code und Stack mehr, ist aber sicherer.

§ 5 *Halten Sie die Assemblermodule möglichst klein!* Heutzutage verfügen die meisten Pascal-Dialekte über sogenannte »intelligente« Compiler, die erkennen, welche Routinen im Programm verwendet werden, und nur diese dann auch benutzen. Allerdings können sie dies nicht mit Assemblermodulen tun. Sie können nicht außerhalb ihres Wirkungskreises erkennen, welche Assemblerroutine überflüssig und daher zu eliminieren ist. Denn schließlich ist ja alles, was sie über die Assemblerroutinen in Erfahrung bringen können, die Adresse der exportierten Routinen in der OBJ-Datei. Wie die interne Struktur im Assemblerteil ist, weiß nur der Assembler, und der wurde zu diesem Zeitpunkt schon beendet. Allerdings kann der Compiler sehr wohl ganze Assemblermodule ignorieren, wenn sie nicht benutzt werden.

TIP Es ist daher guter (und optimierender!) Programmstil, in einzelne Assemblermodule nur die Routinen zu packen, die für die Tätigkeit der aus ihnen exportierten Routine(n) unbedingt erforderlich sind (im besten Fall also eine exportierte Routine pro Modul!). Demonstriert wird dies am Beispiel der Unit *Mathe*, die aus mehreren Assemblermodulen besteht. So wird sichergestellt, daß z.B. das gesamte Assemblermodul mit den trigonometrischen Funktionen nur dann in das fertige Programm eingebaut wird, wenn eine Routine aus diesem Modul verwendet wird. Wird dann aber auch nur eine einzelne Routine auch nur einmal im Programm benutzt, so werden alle anderen trigonometrischen Funktionen dieses Moduls ebenfalls eingebunden (nicht aber die der anderen Module!).

§ 6 *Es ist möglich, in einem Assemblermodul Teile aus der Hochsprache zu benutzen.* Sie müssen in diesem Fall lediglich dem Assembler mitteilen, daß er nun Adressen zu verwenden hat, die der Compiler zur Verfügung stellt. Im Grunde gilt hier das eben für die Richtung Assembler→Hochsprache Gesagte in umgekehrter Weise! Daher gibt es auch hier ein Schlüsselwort:

EXTRN, das Sie in den Assemblerteilen verwenden. Den Compiler selbst brauchen Sie hier nicht anzuweisen, etwas zur Verfügung zu stellen. Das erfolgt automatisch.

Üblicherweise werden Sie in Assemblermodulen nur auf Daten im Datensegment zurückgreifen, seltener auf Routinen. Falls Sie also z.B. auf das Wort PascalDate zurückgreifen wollen, das im Pascal-Teil definiert wird, melden Sie dieses Datum lediglich im Assembler unter

```
Data SEGMENT WORD PUBLIC 'Data'
EXTRN PascalDate:WORD
:
Data ENDS
```

an. Da der Assembler lediglich die Adresse des Datums erhält, nicht aber die Größe, müssen *Sie* dies als Programmierer kundtun, indem Sie hinter den Namen den Datentyp (den der Assembler kennt, also nicht etwa den Pascal-Typ!) durch einen »:« getrennt anführen. Sie können nun ganz normal mit dieser Variablen im Assemblertext arbeiten.

ACHTUNG

Beachten Sie, daß dann in DS das Turbo Pascal-Datensegment stehen und mit ASSUME angemeldet worden sein muß! Allerdings wird dies in der Regel der Fall sein, wenn Sie die Assemblerroutinen einbinden. Denn eines der ersten Dinge, die Pascal in seinem *Start-up-Code* erledigt, ist, das korrekte Datensegment in DS zu speichern. Daher müssen Sie sich nicht mehr darum kümmern!

HINWEIS

Es gibt eine Ausnahme: Wenn Sie Assemblerroutinen programmieren, die Teil des *Start-up-Codes* sind, kann es vorkommen, daß DS noch nicht korrekt belegt wurde! Das ist beispielsweise immer dann der Fall, wenn Sie solche Routinen in dem Teil von Units verwenden, der die Unit initialisiert, also dem Teil, der zwischen einem eventuell vorhandenen *Begin – End* der Unit steht. Da Pascal sich in seinen *Start-up*-Routinen selbst nicht so genau an die Regeln hält, können Sie nicht sicher sein, daß in DS immer das Datensegment eingetragen ist! In diesem Fall sollten Sie auf Nummer Sicher gehen und wie üblich das Datensegment setzen:

```
ASSUME DS:Data
mov      ax, SEG DATA
push     ds
mov      ds,ax
:
:
pop      ds
```

DATA ist ein reserviertes Wort und liefert immer die Adresse des Datensegments zurück. Mit Routinen geht das ebenso:

```
Code SEGMENT BYTE PUBLIC 'Code'
EXTRN PascalRoutine:NEAR

PUBLIC Test
Test PROC FAR
         :
         call  PascalRoutine
         :
Test ENDP

Code ENDS
END
```

Die (exportierbare) Routine `Test` will auf die Pascal-Routine `PascalRoutine` zurückgreifen. Daher muß diese dem Assembler in Form ihrer Adresse (die der Compiler zur Verfügung stellt) bekannt gemacht werden. Da sich diese Routine im gleichen Segment wie der Assemblerteil befindet, kann sie *near* angesprungen und daher auch als solche deklariert sein, was analog zu den Daten mit *NEAR* hinter einem Doppelpunkt erfolgt! Das war alles!

§ 7 *Legen Sie Daten niemals im Codesegment ab, wenn Sie Programme unter Windows programmieren und nutzen wollen!* Windows läuft üblicherweise im Protected-Mode des Prozessors und verbietet einen schreibenden Zugriff auf das Codesegment! Nutzen Sie das Datensegment, oder erzeugen Sie ein eigenes.

§ 8 Pascal hat seine eigenen Ansichten über extern definierte Daten! Sie können zwar Daten im Assemblermodul definieren, die ins gemeinsame Datensegment kommen, jedoch sind diese Daten nur lokal im Assemblermodul bekannt (oder *privat*, wie das Handbuch sagt!). Das bedeutet, daß Pascal-Programmteile auf diese Daten nicht zurückgreifen können!

TIP Definieren Sie die Daten im Pascal-Teil, und importieren Sie sie mittels EXTRN in das Assemblermodul.

§ 9 *Daten, die im Assemblermodul definiert werden (auch wenn sie im Datensegment von* Pascal *stehen!), können nicht vorbelegt werden*. Sie werden zwar erzeugt und haben eine (korrekte) Adresse, aber sie besitzen nach ihrer Erzeugung nicht den Inhalt, den Sie ihnen eventuell im Assemblermodul zugewiesen haben!

```
Data SEGMENT WORD PUBLIC 'Data'
MyWord DW 04711h
Data ENDS

Code SEGMENT BYTE PUBLIC 'Code'
```

```
ASSUME CS:Code, DS:Data

PUBLIC MyProc
MyProc PROC FAR
         mov   ax, MyWord
         ret
MyProc ENDP
Code ENDS
END
```

Falls Sie `MyProc` in ein Pascal-Programm einbinden, so wird es nur zufälligerweise den korrekten Wert zurückgeben, da der Compiler zwar `MyWord` im Datensegement erzeugt, nicht aber vorbelegt! Abhilfe: Weisen Sie den Wert in der Routine zu:

```
Data SEGMENT WORD PUBLIC 'Data'
MyWord DW ?
Data ENDS

Code SEGMENT BYTE PUBLIC 'Code'
ASSUME CS:Code, DS:Data

PUBLIC MyProc
MyProc PROC FAR
         mov   MyWord, 04711h
         mov   ax, MyWord
         ret
MyProc ENDP
Code ENDS
END
```

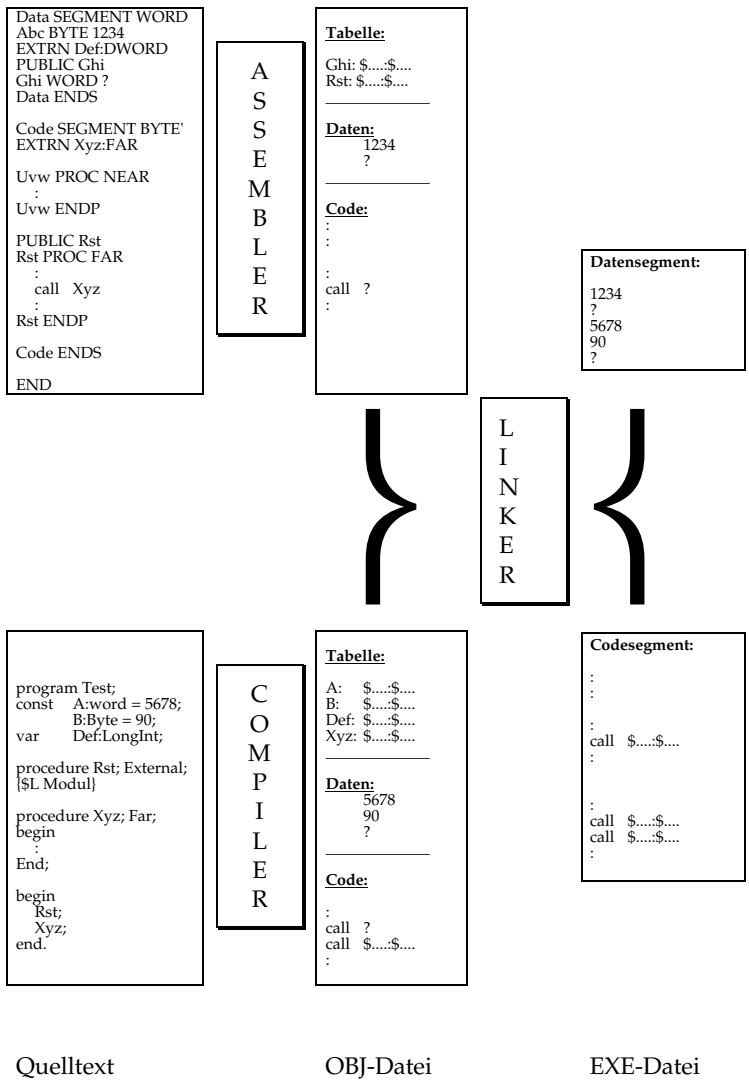
Natürlich macht dieses Vorgehen nur Sinn, wenn Sie auch innerhalb des Assemblermoduls auf `MyWord` zurückgreifen. Ansonsten könnten Sie gleich programmieren: `mov ax, 04711h`.

26.4 Wie arbeitet der Compiler?

Beim Pascal-Compiler handelt es sich eigentlich um einen Zwitter. Er ist nämlich ein Compiler im engeren Sinne, ergänzt um einen sogenannten Linker, der für das Zusammenfassen einzelner Module zuständig ist. Dieser Linker ist auch der Teil, dem alle Adressen bekannt sein müssen, da er die Verbindungen zwischen den Modulen herstellt (*link*t!).

Der eigentliche Compiler von Pascal selbst macht im Prinzip nichts anderes als der Assembler auch: Er übersetzt den Quellcode in eine Folge von Opcodes für den Prozessor. Dies erfolgt allerdings gemäß

den Richtlinien der Hochsprache, hier Pascal. Das Ergebnis der Tätigkeit des Compilers ist ganz analog zu der des Assemblers eine Objektdatei, in der der kompilierte Code steht. Allerdings werden Sie, anders als bei *C*, diese Objektdatei niemals zu Gesicht bekommen, da der Compiler (genauer gesagt: der integrierte Linker) sie gleich weiterverwendet, um das lauffähige Programm zu erzeugen.



Diese temporäre OBJ-Datei enthält im Prinzip die gleichen Informationen wie die vom Assembler erzeugte: den eigentlichen Code nebst Daten sowie eine Tabelle mit den Adressen exportierbarer Routinen und Daten. Auf gut deutsch: Assembler und Compiler erzeugen beide je ein Daten- und ein Codemodul sowie eine Tabelle mit den »nach außen« zugänglichen Daten- und Routinenadressen. Zusammengeführt wird dies durch den Linker, der nun jeweils die Adressen aus den Tabellen entnimmt, sie dort einsetzt, wo sie gebraucht werden und Daten und Code in die unabhängigen Segmente kombiniert.

Dies ist übrigens auch der Grund dafür, warum die Segmente in den Assemblermodulen die gleichen Namen haben müssen wie die der Hochsprache: Schließlich sollen ja alle Daten ins gleiche (und bei Pascal einzige) Datensegment. Ferner sollen die Routinen ebenfalls möglichst »aufgeräumt« verfügbar sein. Schauen Sie sich dies einmal in der Abbildung auf der vorhergehenden Seite an.

Beachten Sie bitte, daß unter Turbo Pascal der »Linker« im Compiler integriert ist, also keine eigenständige OBJ-Datei erstellt wird. Dennoch ist der Prozeß der gleiche!

27 Assembler und C

Auf den folgenden Seiten werden die gleichen Überlegungen für C angestellt, die eben für Pascal abgeklärt wurden. Selbst wenn Sie kein C-Programmierer sind, lohnt es sich, das nun folgende zu lesen. Denn trotz aller Unterschiede zwischen den Programmiersprachen: *so sehr*, wie manche einem glaubhaft machen möchten, unterscheiden sie sich auch nicht!

27.1 Parameter in C

Gehen wir analog zu Pascal vor: Kompilieren wir einfach ein kleines C-Programm, das einer Funktion einen Parameter übergibt, also z.B.:

```
char Test (char i);
char j;

int main(void)
{
  j = Test (2);
  return(0)
}
```

```
char Test (char i)
{
  return(i);
}
```

Der Compiler macht hieraus:

```
12D1:02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 B002          MOV     AL,02
     02C7 50            PUSH    AX
     02C8 E80A00        CALL    02D5
     02CB 59            POP     CX
     02CC A28C02        MOV     [028C],AL
     02CF 33C0          XOR     AX,AX
     02D1 EB00          JMP     02D3
     02D3 5D            POP     BP
     02D4 C3            RET
     02D5 55            PUSH    BP
     02D6 8BEC          MOV     BP,SP
     02D8 8A4604        MOV     AL,[BP+04]
     02DB EB00          JMP     02DD
     02DD 5D            POP     BP
     02DE C3            RET
```

An $12D1:02D5 beginnt die Funktion `Test`. Hier wird ebenfalls zunächst ein Stackrahmen eingerichtet. Allerdings erzeugt der Compiler offensichtlich keine lokale Variable für das Funktionsergebnis! Der Parameter wird auch hier, wie im Falle von Pascal, an der Stelle [BP+4] gefunden. An [BP+2] steht, da die Routine *near* aufgerufen wird, nur der Offset der Rücksprungadresse und an [BP+0] der alte BP-Inhalt. Auch hier soll uns zunächst an der Funktion nichts weiter interessieren.

Das Hauptprogramm, das an CS:02C2 beginnt, richtet zunächst ebenfalls einen Stackrahmen ein und schiebt dann die Konstante 2 auf den Stack. Anschließend wird die Funktion `Test` aufgerufen. Soweit läuft also alles vollkommen gleich zu Pascal ab.

Betrachten wir nun den Fall der Übergabe mehrerer Parameter:

```
char Test (char i, char k);
char j;

int main(void)
{
  j = Test (1,2);
  return(0);
}
```

```
char Test (char i, char k)
{
  return(i);
}
```

Auch hier erwartet die Funktion `Test` zwei Parameter und bekommt diese auch übergeben. Das Compilat sieht dann so aus:

```
12D1:02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 B002          MOV     AL,02
     02C7 50            PUSH    AX
     02C8 B001          MOV     AL,01
     02CA 50            PUSH    AX
     02CB E80B00        CALL    02D9
     02CE 59            POP     CX
     02CF 59            POP     CX
     02D0 A28C02        MOV     [028C],AL
     02D3 33C0          XOR     AX,AX
     02D5 EB00          JMP     02D7
     02D7 5D            POP     BP
     02D8 C3            RET
     02D9 55            PUSH    BP
     02DA 8BEC          MOV     BP,SP
     02DC 8A4604        MOV     AL,[BP+04]
     02DF EB00          JMP     02E1
     02E1 5D            POP     BP
     02E2 C3            RET
```

Verglichen mit dem Compilat von oben, stellen wir fest, daß auch in diesem Fall in der Funktion an der Adresse CS:02DC die Variable *i* an der Stackposition [BP+4] gefunden wird. Das bedeutet aber, daß der Parameter *k* an [BP+6] stehen muß.

Dies bestätigt sich dadurch, daß das Hauptprogramm an Adresse CS:02C5 zuerst den Wert 2 (also *k*) auf den Stack schiebt, bevor 1 repräsentativ für die Variable *i* auf dem Stack abgelegt wird. Offensichtlich arbeitet sich also der Compiler von rechts nach links durch die Parameter, also genau anders herum als Pascal.

HINWEIS

C arbeitet die Parameter, die Routinen übergeben werden, von rechts nach links ab. Das heißt, daß der erste Parameter der Parameterliste einer Routine an der physikalisch untersten, aus Blickwinkel des Stacks betrachtet, an oberster Position auf dem Stack liegt, während der letzte an der physikalisch obersten, also untersten Stackposition liegt.

Wenn Sie nun für alle möglichen Parametertypen und -arten solche Miniprogramme entwickeln und das Compilat analysieren, so kommen Sie zu folgenden Vereinbarungen unter *C*:

- Parameter, die *by value* übergeben werden (Wertparameter), werden mit Ausnahme von *arrays* immer als Wert über den Stack übergeben (in Pascal ist dies bei komplexeren Daten nicht der Fall). *By-reference*-Parameter werden mittels Zeiger übergeben, die entweder *near* oder *far* angegeben sein können (modellabhängig). Pascal verwendet hier grundsätzlich *far*-Zeiger.
- Wertparameter vom Typ *signed* und *unsigned char*, die also nur ein Byte belegen, werden als Wort auf dem wortorientierten Stack abgelegt, wobei das »obere« Byte undefiniert ist.
- Wertparameter vom Typ *signed* und *unsigned int* werden ebenfalls über den Stack übergeben. Sie belegen ebenfalls ein Wort.
- Auch Wertparameter größerer Länge, also *signed* und *unsigned long* (4 Bytes), werden auf dem Stack abgelegt. Das jeweils »obere« Wort wird hierbei gemäß Intel-Konvention zuerst abgelegt, so daß es an physikalisch höherer Stackadresse vorgefunden wird.
- Ebenfalls über den Stack übergeben werden alle Coprozessordaten, also Parameter vom Typ *float*, *double* und *long double*.
- Zeiger jeglicher Art werden über den Stack *mit zwei Worten* übergeben, wobei das Wort an der physikalisch höheren Stackadresse den Segmentanteil beinhaltet, das »untere« Wort den Offsetanteil. Dies gilt allerdings nur, falls mit *far*-Zeigern gearbeitet wird. Bei Zeigern, bei denen nur die Offsets der Adressen Verwendung finden, wird ein Wort über den Stack übergeben. Das gleiche gilt für Parameter, die nicht als Wertparameter (*call by value*) übergeben werden, sondern als Variablenparameter (*call by reference*; in *C* durch das Präfix * in der Parameterliste gekennzeichnet).
- Zeichenketten vom Typ *string* werden nicht gesondert behandelt. Sie werden wie alle anderen Daten vom Typ *array* übergeben.
- Bei Daten von Typ *union*, *struct* und *array* werden Zeiger auf die Struktur oder das *array* übergeben. Ausnahme: Bei Strukturen werden Daten im Falle von *by-value*-Übergabe selbst auf den Stack gelegt.

Demonstrieren wir das ebenfalls kurz anhand eines Listings und des Compilats von Turbo C++:

```
typedef struct Rec1 { char B1, B2; };
typedef struct Rec2 { int  W1, W2, W3; };
typedef char   String[21];
typedef long   *pointer;
```

```
typedef int    *pInt;
typedef char   Feld[60];

char        aChar;
int         anInt;
long        aLong;
String      aString = "Ein String";
long double aLongDouble;
pointer     aPointer;
pInt        anotherP;
Rec1        aRecord;
Rec2        anotherR;
Feld        anArray;

char Test (char C,        char *C_,
           int  I,         int  *I_,
           long L,         long *L_,
           String S,       String *S_,
           long double D, long double *D_,
           pointer P1,    pointer *P1_,
           pInt P2,        pInt *P2_,
           Rec1 R1,        Rec1 *R1_,
           Rec2 R2,        Rec2 *R2_,
           Feld A,         Feld *A_);

int main(void)
{
  aChar = 1;
  anInt = 2;
  aLong = 3;
  aLongDouble = 5.0;
  anotherP = &anInt;
  aRecord.B1 = 6; aRecord.B2 = 7;
  anotherR.W1 = 8; anotherR.W2 = 9; anotherR.W3 = 10;

  aChar = Test(aChar,       &aChar,
               anInt,        &anInt,
               aLong,        &aLong,
               aString,      &aString,
               aLongDouble, &aLongDouble,
               aPointer,     &aPointer,
               anotherP,     &anotherP,
               aRecord,      &aRecord,
               anotherR,     &anotherR,
               anArray,      &anArray);
```

```
  return(0);
}

char Test (char C,        char *C_,
           int  I,         int  *I_,
           long L,         long *L_,
           String S,       String *S_,
           long double D, long double *D_,
           pointer P1,    pointer *P1_,
           pInt P2,       pInt *P2_,
           Rec1 R1,       Rec1 *R1_,
           Rec2 R2,       Rec2 *R2_,
           Feld A,        Feld *A_)
{
}
```

Analog zu dem weiter oben abgedruckten Pascal-Programm werden in diesem Programm ebenfalls ein paar Typen deklariert und einige Variablen angelegt. Die Routine ist eine einfache Prozedur, der nun mehrere Parameter übergeben werden.

Zum Vergleich und des besseren Überblicks wegen wurde auch hier die Zeile mit den Parametern umbrochen und in jeder Zeile die betreffende Variable einmal als Wertparameter (*call by value*) und einmal als Variablenparameter (*call by reference*) übergeben. Die Prozedur selbst tut gar nichts! Sie ist wie bei der Pascal-Variante eine Dummy-Prozedur. Das Hauptprogramm weist nun den Variablen einige Werte zu und ruft dann die Prozedur auf.

Wenn wir das Compilat dieser Wertzuweisungen außer acht lassen, so erzeugt der Compiler folgenden Code:

```
CS : 030A B8FB06        MOV     AX,06FB    *anArray
     030D 50            PUSH    AX         Zeiger near!

     030E B8FB06        MOV     AX,06FB    anArray
     0311 50            PUSH    AX         Zeiger!

     0312 B8F506        MOV     AX,06F5   *anotherR
     0315 50            PUSH    AX

     0316 B8F506        MOV     AX,06F5    anotherR
     0319 8CDA          MOV     DX,DS      kein Zeiger!
     031B B90600        MOV     CX,0006
     031E E8740C        CALL    0F95
```

```
0321 B8F306        MOV     AX,06F3    *aRecord
0324 50                  PUSH    AX

0325 FF36F306      PUSH    [06F3]     aRecord
                                      kein Zeiger!

0329 B8F106        MOV     AX,06F1    *anotherP
032C 50            PUSH    AX

032D FF36F106      PUSH    [06F1]     anotherP

0331 B8EF06        MOV     AX,06EF    *aPointer
0334 50            PUSH    AX

0335 FF36EF06      PUSH    [06EF]     aPointer

0339 B8E506        MOV     AX,06E5    *aLongDouble
033C 50            PUSH    AX

033D CD372EE506    FLD     [06E5]     aLongDouble
0342 83EC0A        SUB     SP,+0A
0345 CD377EDC      FSTP    [bp-24]

0349 B8AA00        MOV     AX,00AA    *aString
034C 50            PUSH    AX

034D B8AA00        MOV     AX,00AA    aString
0350 50            PUSH    AX

0351 B8E106        MOV     AX,06E1    *aLong
0354 50            PUSH    AX
0355 CD3D          FWAIT

0357 FF36E306      PUSH    [06E3]     aLong (hi)
035B FF36E106      PUSH    [06E1]           (lo)

035F B8DF06        MOV     AX,06DF    *anInt
0362 50            PUSH    AX

0363 FF36DF06      PUSH    [06DF]     anInt

0367 B8DE06        MOV     AX,06DE    *aChar
036A 50            PUSH    AX
```

```
036B A0DE06        MOV     AL,[06DE]  aChar
036E 50            PUSH    AX         wortweise!

036F E80C00        CALL    037E
```

Beachten Sie auch hier die fett gedruckten Anmerkungen: Ablage der Komponenten eines Datums in umgekehrter Reihenfolge gemäß Intel-Konvention sowie die grundsätzliche Übergabe von Zeigern bei *arrays*. Beachten Sie bitte auch, daß bei der Übergabe von Zeigern lediglich die Offsetteile auf den Stack gebracht werden! Bei *records* werden die Daten selbst auf den Stack gelegt, falls diese *by value* übergeben werden. Im Unterschied zu Pascal erfolgt dies grundsätzlich! Bei *arrays* dagegen werden grundsätzlich Zeiger übergeben.

Daß übrigens in C Strings keine Sonderbehandlung wie in Pascal erfahren, ersehen Sie aus der Kürze der Routine `Test`, die hier tatsächlich nur den Umfang hat, der in der Pascal-Version mit den Pfeilen markiert war:

```
037E 55            PUSH    BP
037F 8BEC          MOV     BP,SP
0381 5D            POP     BP
0382 C3            RET
```

Strings sind in C ganz »normale« *arrays*.

27.2 Funktionswerte in C

Kommen wir auch für *C* noch zur Übergabe von Funktionswerten. Dazu betrachten wir noch einmal unser kleines Programm:

```
char Test (char i);
char j;

int main(void)
{
  j = Test (2);
  return(0)
}

char Test (char i)
{
  return(i);
}
```

Ich möchte Ihr Augenmerk lediglich auf die letzten zwei Zeilen des Disassemblats lenken. Da C keinen Speicher für lokale Variablen eingerichtet hat, darf uns auch nicht wundern, daß der Stackrahmen nur

mit einem *pop bp* entfernt wird. Da ja SP nicht verändert wurde, enfällt die Notwendigkeit zu einem *mov sp,bp*.

```
12D1:02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 B002          MOV     AL,02
     02C7 50            PUSH    AX
     02C8 E80A00        CALL    02D5
     02CB 59            POP     CX
     02CC A28C02        MOV     [028C],AL
     02CF 33C0          XOR     AX,AX
     02D1 EB00          JMP     02D3
     02D3 5D            POP     BP
     02D4 C3            RET
     02D5 55            PUSH    BP
     02D6 8BEC          MOV     BP,SP
     02D8 8A4604        MOV     AL,[BP+04]
     02DB EB00          JMP     02DD
     02DD 5D            POP     BP
     02DE C3            RET
```

Auffällig sind, verglichen mit dem Compilat des analogen Pascal-Programms, zwei Dinge:

- Die Funktion `Test` endet trotz Übergabe eines Parameters über den Stack nur mit einem *ret*, das keine Stackbereinigung durchführt.
- Nach dem CALL-Befehl im Hauptprogramm, der `Test` aufruft, folgt unmittelbar ein POP-Befehl, der den Stackinhalt in CX poppt.

Dies scheint zu bedeuten, daß in C nicht die Routine für das Wiederherstellen des Stacks zuständig ist, sondern der Teil, der die Routine ruft. Dies ist sehr geradlinig gedacht. Denn schließlich weiß niemand anderes besser über die Anzahl der übergebenen Parameter und ihren Typ Bescheid als der Programmteil, der die Parameter auf den Stack legt. Das dem CALL-Befehl folgende Poppen in das CX-Register ist dann der Teil, der den Stack restauriert.

In diesem Fall sollte in unserem zweiten Programm ein zweifaches *pop cx* dem CALL-Befehl folgen. Dem ist tatsächlich so.

HINWEIS

In C obliegt die Verantwortung, den Stack von Übergabewerten zu befreien, der rufenden Routine! Sie hat dafür zu sorgen, daß alle Parameter, die sie vor dem Unterprogrammaufruf auf den Stack legt, nach der Rückkehr wieder vom Stack genommen werden.

Heißt das nun, daß alle übergebenen Parameter vom Stack gepoppt werden? Das würde ja bedeuten, daß nach einem *Call* einer Routine,

der 6 *long doubles* übergeben werden, 60 Bytes via *pop cx* entfernt werden müßten: Also ein Rattenschwanz von 30 *pop cx*! Nein. Die Methode mit *pop cx* wird von *C* nur so lange verwendet, bis die Codegröße dieser Sequenz die eines anderen Befehls überschreitet: *add sp, xx*. Nachdem ein *pop cx* ein Byte umfaßt, das *add sp, xx* aber drei, heißt das, daß drei Worte via *pop cx* vom Stack geholt werden. Ab dann greift die *add*-Variante:

```
int Test (int i, int j, int k, int l, int m, int n);
int o;

int main(void)
{
  o = Test (2,2,2,2,2,2);
  return(0);
}

int Test (int i, int j, int k, int l, int m, int n)
{
  return(i);
}
```

Das Disassemblat sieht so aus:

```
CS : 02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 B80200        MOV     AX,0002
     02C8 50            PUSH    AX
     02C9 B80200        MOV     AX,0002
     02CC 50            PUSH    AX
     02CD B80200        MOV     AX,0002
     02D0 50            PUSH    AX
     02D1 B80200        MOV     AX,0002
     02D4 50            PUSH    AX
     02D5 B80200        MOV     AX,0002
     02D8 50            PUSH    AX
     02D9 B80200        MOV     AX,0002
     02DC 50            PUSH    AX
     02DD E80C00        CALL    02EC
     02E0 83C40C        ADD     SP,+0C
     02E3 A38C02        MOV     [028C],AX
     02E6 33C0          XOR     AX,AX
     02E8 EB00          JMP     02EA
     02EA 5D            POP     BP
     02EB C3            RET
     02EC 55            PUSH    BP
```

```
02ED 8BEC          MOV     BP,SP
02EF 8B4604        MOV     AX,[BP+04]
02F2 EB00          JMP     02F4
02F4 5D            POP     BP
02F5 C3            RET
```

Die unterschiedlichen Funktionswerte werden in folgenden Registern übergeben:

- Funktionsergebnisse vom Typ *unsigned char* und *char*, die also nur ein Byte belegen, werden in AL zurückgegeben.
- Funktionsergebnisse vom Typ *unsigned int* und *int* werden in AX zurückgegeben.
- *Unsigned long und long* als Ergebnis werden in DX:AX übergeben, wobei in DX das höherwertige und in AX das niederwertige Wort steht.
- Funktionsergebnisse vom Typ *float*, *double* und *long double* werden über den TOS des Coprozessorstacks übergeben.
- Zeiger jeglicher Art werden in DX:AX zurückgegeben, falls sie *far*-Adressen beherbergen, wobei DX den Segment- und AX den Offsetanteil enthält. Wird nur der Offset einer Adresse übermittelt, wie dies bei allen *near*-Adressen der Fall ist, steht dieser in AX.
- Daten aller anderen, auch selbstdefinierter Typen können nicht als Funktionsergebnis zurückgegeben werden. In diesem Fall müssen Argumente *by reference* als Parameter übergeben werden. Alternativ kann die Funktion auch einen Zeiger auf ein solches Datum übermitteln.

27.3 Parameter und Funktionen in C, Teil 2

Pascal und C haben also recht unterschiedliche, ja fast sogar kontroverse Regeln, an die sie sich halten. Nun würde dies bedeuten, daß man Assemblerroutinen speziell für die jeweilige Hochsprache entwickeln müßte. Denn schließlich kann z.B. C mit dem Befehl *ret 10*, mit dem Pascal eine Routine abschließen könnte, nichts anfangen!

Turbo Pascal läßt, im Gegensatz zu anderen Pascal-Dialekten wie Microsoft Professional Pascal oder Quick Pascal, keinerlei Veränderungen seiner Übergaberegeln zu. Nicht so C! Dankenswerterweise gibt es in C eine Möglichkeit, Routinen genau so aufzubauen, wie Pascal dies tut. Sie müssen fast gar nichts dafür tun:

```
char pascal Test (char i);
char j;
```

```
int main(void)
{
  j = Test (2);
  return(0);
}

char pascal Test (char i)
{
  return(i);
}
```

Fügen Sie bei der Definition der Routine lediglich das Schlüsselwort PASCAL ein – der Compiler macht daraus:

```
CS : 02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 B002          MOV     AL,02
     02C7 50            PUSH    AX
     02C8 E80900        CALL    02D4
     02CB A28C02        MOV     [028C],AL
     02CE 33C0          XOR     AX,AX
     02D0 EB00          JMP     02D2
     02D2 5D            POP     BP
     02D3 C3            RET
     02D4 55            PUSH    BP
     02D5 8BEC          MOV     BP,SP
     02D7 8A4604        MOV     AL,[BP+04]
     02DA EB00          JMP     02DC
     02DC 5D            POP     BP
     02DD C20200        RET     0002
```

Bemerken Sie bitte, daß der Compiler hier die Regeln von Pascal berücksichtigt hat: Die Routine wird mit einem RET-Befehl abgeschlossen, der den Parameter vom Stack nimmt. Dafür wird nach Aufruf der Routine im Hauptprogramm das Funktionsergebnis sofort gespeichert, ohne daß mittels *pop cx* der Stack erst bereinigt wird. Auch die Abarbeitung der Parameter erfolgt dann Pascal-konform, wie das zweite, modifizierte Programm zeigt:

```
char pascal Test (char i, char k);
char j;

int main(void)
{
  j = Test (1,2);
  return(0);
}
```

```
char pascal Test (char i, char k)
{
  return(i);
}
```

In diesem Fall werden die Parameter von links nach rechts auf den Stack gelegt:

```
CS : 02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 B001          MOV     AL,01
     02C7 50            PUSH    AX
     02C8 B002          MOV     AL,02
     02CA 50            PUSH    AX
     02CB E80900        CALL    02D7
     02CE A28C02        MOV     [028C],AL
     02D1 33C0          XOR     AX,AX
     02D3 EB00          JMP     02D5
     02D5 5D            POP     BP
     02D6 C3            RET
     02D7 55            PUSH    BP
     02D8 8BEC          MOV     BP,SP
     02DA 8A4606        MOV     AL,[BP+06]
     02DD EB00          JMP     02DF
     02DF 5D            POP     BP
     02E0 C20400        RET     0004
```

Dank dieses netten Zuges von C können Sie Assemblerroutinen erstellen, die sowohl in Pascal- als auch in C-Programmen eingesetzt werden können!

ACHTUNG

Hierbei ist jedoch Vorsicht geboten. Denn wie im Kapitel über Pascal schon erwähnt wurde, haben die Pascal-Compiler unterschiedlicher Hersteller unterschiedliche Übergabekonventionen. Dies kommt in den unterschiedlichen C-Dialekten zum Tragen.

Visual C++

Der Visual-C++-Compiler hält sich an das, was eben über die Rückgabe von Funktionswerten für Turbo C++ gesagt wurde. Allerdings überrascht er mit einem seltsamen Aufbau des Stacks, falls eine Funktion Daten vom Typ *float*, *double* und *long double* zurückgibt.

Sehen wir uns dies am Beispiel einer Assemblerroutine im Pascal-Format an, die extern deklariert ist und eingebunden wird. Sie macht nichts anderes, als den Übergabeparameter *X* als Funktionsergebnis zurückzugeben, und ist daher selbst nicht weiter interessant.

Danach müßte die Einbindung wie folgt geschehen:

```
extern long double far pascal ATest(long double X);

long double i;

void main()
{
  i = ATest(1.23);
}
```

Kompiliert man dies mit Turbo C++, so erhält man folgendes Disassemblat, wobei wir uns lediglich auf den Teil beschränken, der die eingebundene Routine aufruft:

```
CS : 02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 CD372EAA00    FLD     TBYTE PTR [00AA]
     02CA 83EC0A        SUB     SP, 000A
     02CD CD377EF6      FSTP    TBYTE PTR [BP-0A]
     02D1 CD3D          FWAIT
     02D3 90            NOP
     02D4 0E            PUSH    CS
     02D5 E80A00        CALL    02E2
     02D8 CD373E9605    FSTP    TBYTE PTR [_i]
     02DD CD3D          FWAIT
     02DF 5D            POP     BP
     02E0 C3            RET
```

Ganz geradlinig richtet das Hauptprogramm `main` einen Stackrahmen ein und lädt die `long double`-Konstante aus der Speicherstelle $000A in den TOS. Dann wird 10 Bytes Platz auf dem Stack für das Argument geschaffen und der auf dem TOS liegende Wert dort zur Übergabe an `ATest` abgelegt. (Es muß ja nicht immer über das Pushen erfolgen! Dies ist eine andere und effektivere Methode. Das Resultat ist das gleiche.) Das folgende NOP übersehen wir einfach!

Das Modul mit `ATest` wurde ganz offensichtlich in das gleiche Segment wie `main` eingebaut, da `ATest` durch einen *Near Call* gerufen wird. Da wir `ATest` jedoch als *far*-Routine deklariert haben, legt der Compiler vor dem CALL noch das Codesegment auf den Stack, damit das *Far Return* der Routine einen vollständigen Zeiger mit Segmentanteil laden kann!

Nach dem Routinenaufruf erfolgt wie erwartet lediglich das Abspeichern des über den TOS übergebenen Funktionsergebnisses, das Entfernen des Stackrahmens und das Beenden von `main`. Wie aber kompiliert Visual C++ das gleiche Programm?

```
CS : 0010 55            PUSH    BP
     0011 8BEC          MOV     BP,SP
     0013 B80C00        MOV     AX,000C
     0016 E8F302        CALL    030C
     0019 56            PUSH    SI
     001A 57            PUSH    DI
     001B 83EC0A        SUB     SP,+0A
     001E CD3906B802    FLD     QWORD PTR [02B8]
     0023 CD3D          FWAIT
     0025 8BDC          MOV     BX,SP
     0027 CD373F        FSTP    TBYTE PTR [BX]
     002A CD3D          FWAIT
     002C 8D46F4        LEA     AX,[BP-0C]
     002F 50            PUSH    AX
     0030 9A4800E80E    CALL    0048
     0035 8BD8          MOV     BX,AX
     0037 CD372F        FLD     TBYTE PTR [BX]
     003A CD373EA004    FSTP    TBYTE PTR [04A0]
     003F CD3D          FWAIT
```

Auch Visual C++ legt nach dem Einrichten eines Stackrahmens und dem Aufruf einer nicht weiter interessanten Routine einen Speicherplatz der Größe von 10 Bytes zur Parameterübergabe auf dem Stack an. Dann wird aus DS:$02B8 der Übergabewert in den TOS geladen und auf dem eben eingerichteten Platz abgelegt. Wie Sie im fett gedruckten Teil sehen können, legt Visual C++ vor dem Aufruf von `ATest` noch einen Wert auf den Stack. Es handelt sich hierbei um den *Near*-Zeiger, den ich im Kapitel über Pascal *WhoAmI* getauft habe und der von Professional Pascal auf dem Stack abgelegt wird, wenn Fließkommazahlen als Funktionsergebnis übergeben werden.

ACHTUNG

Wie dieses Beispiel zeigt, hält sich Visual C++ Punkt für Punkt an die Pascal-Übergabekonventionen von Professional Pascal. Das bedeutet, daß nur Routinen eingebunden werden können, die dieser Konvention folgen. Anders ausgedrückt: Wann immer Sie ein Assemblermodul mit Pascal-Namens- und Übergabekonvention für Visual C++ erstellen, müssen Sie *WhoAmI* berücksichtigen! Denn Microsoft versteht unter der Pascal-Übergabekonvention bei diesem Compiler etwas anderes als z.B. Borland. *Mischen Sie daher keinesfalls Module dieser beiden Hersteller.* Falls dies unumgänglich ist, achten Sie darauf, daß die übergebenen Parameter auch von der aufgerufenen Routine korrekt gefunden werden können, indem Sie die Symbole zum Ansprechen der Übergabewerte an die neuen Adressen des Stacks anpassen.

TIP

Falls Sie mit Visual C++ arbeiten, sollten Sie besser die einzubindenden Routinen in C-Konvention bringen. Dies ist meistens wesentlich

einfacher und schneller erledigt als das mühsame Anpassen und Prüfen der Routinen nach Pascal-Konvention. Denn schließlich brauchen Sie, wenn Sie symbolische Namen mittels EQU definieren, nur die Parameterliste umzudrehen, jedes öffentliche Label (also Daten- und Routinennamen) mit einem Unterstrich »_« beginnen zu lassen und beim RET-Befehl auf die Angabe der zu entfernenden Bytes zu verzichten (ein Beispiel hierfür finden Sie bei den beiden Versionen der Bibliothek MATHE.LIB).

27.4 Einbinden von Assemblermodulen in C-Programme

Die Einbindung von Assemblerteilen in C-Programme funktioniert nicht ganz so einfach wie in Pascal! Auch gibt es hier Unterschiede bei den Dialekten, so daß im Rahmen dieses Buches nicht jeder Aspekt berücksichtigt werden kann.

§ 1 *Die Definition und Implementation von Assemblerroutinen im Assemblermodul reicht nicht aus!* Sie müssen dem Assembler mitteilen, daß er in die zu erzeugende Objektdatei auch die Information aufnimmt, an welcher Adresse im Modul sich die zu importierende (= anzuspringende) Routine befindet. Schließlich wissen Sie ja, daß alle Namen in Assembler nur Platzhalter für irgend etwas sind: Routinennamen repräsentieren eben Adressen! Sie erreichen dies jedoch ganz einfach: Irgendwo im Assemblerquelltext muß lediglich die Anweisung

```
PUBLIC _Xyz
```

stehen. Dies veranlaßt den Assembler, die Adresse der Routine *_Xyz* in eine Tabelle aufzunehmen, aus der sich der Hochsprachenteil bedient. Ohne diese Anweisung ist *_Xyz* nur im Assemblermodul bekannt!

TIP Um den Überblick zu behalten, empfehle ich Ihnen, dies grundsätzlich unmittelbar vor der Deklaration der Routine zu tun:

```
PUBLIC _ImportedRoutine
_ImportedRoutine PROC FAR
          :
          :
          :
_ImportedRoutine ENDP
```

HINWEIS Dies gilt nicht nur für Routinen, sondern auch für Daten, die im Modul deklariert und exportiert (bzw. aus dem Blickwinkel der Hochsprache importiert) werden sollen. Das bedeutet: Alles, was im Assemblermodul definiert und exportiert wird, muß mit PUBLIC markiert werden!

Im C-Modul muß die Routine mit einem Routinenrumpf angemeldet werden. § 2
Die Syntax ist hierbei die gleiche wie bei ganz »normalen« C-Routinen, erweitert um das Schlüsselwort *extern*:

```
extern char Xyz(char X, int Y,Z);
```

Auf diese Weise teilen Sie jedoch dem Compiler lediglich mit, daß er eine Routine (hier: Funktion) einbinden soll, die extern deklariert ist. *Eingebunden wird die Routine hierdurch noch nicht!* Die Angaben über die Variablen benötigt der Compiler, um die Parameter vor dem CALL-Befehl auf den Stack zu legen. Ebenso erzeugt er durch die Typzuweisung *char* und den Routinennamen nach dem CALL-Befehl Code, der das Funktionsergebnis anhand des Funktionstyps weiterverarbeitet.

Beachten Sie an dieser Stelle, daß **ACHTUNG**

- *Art*,
- *Anzahl* und
- *Typ* der Variablen sowie die
- *Reihenfolge* ihrer Deklaration korrekt angegeben werden müssen.

Hier finden sich häufig Fehler, die dann zu Abstürzen oder instabil ablaufenden Programmen führen. Denken Sie daran, daß der C-Compiler die Parameter von rechts nach links auf den Stack legt! *Denken Sie im Falle von Visual C++ auch daran, daß eventuell ein versteckter Parameter mehr übergeben wird!*

Das eigentliche Einbinden des Assemblermoduls ist eine Aufgabe des § 3
Linkers. Daher erfolgt dies erst beim Linken der durch den Compiler erzeugten Module.

Bei der Einbindung von Assemblermodulen in C-Programme ist darauf zu achten, daß beide Teile das gleiche Speichermodell verwenden. § 4

Dahinter verbirgt sich die Flexibilität von C, den Speicher bestimmten Bedingungen anpassen zu können. Als C-Programmierer werden Ihnen die Modelle TINY, SMALL, MEDIUM, COMPACT, LARGE und HUGE bekannt sein. Ich möchte nicht weiter auf dieses Thema eingehen. Wichtig bei der Verwendung von Assemblerteilen ist, daß diese in dem Speichermodell assembliert werden, das auch das C-Modul verwendet. Wie man dies in Assembler realisiert, werden wir in einem folgenden Kapitel noch sehen!

Dialektabhängig können C-Programme typsicher gelinkt werden. § 5
Hierunter versteht man, daß Compiler und Linker in C überwachen, ob die übergebenen Parameter die korrekten Typen verwenden. *Falls Sie diese Fähigkeit nutzen wollen, müssen Sie den Assemblerroutinen und*

-daten bestimmte Namen geben, die der Linker *erwartet. Dies ist nicht trivial, da die Namen Informationen über die verwendeten Typen enthalten!* Am besten erzeugen Sie hierzu einen Assemblerquelltext mit Hilfe von *C*, indem Sie die Funktionsprototypen mit der Compileroption »ASM-Datei erstellen« (in Turbo C++: »-S«, in Visual C++: »-Fa«) erzeugen. Diesen Assemblerquelltext können Sie dann entsprechend Ihren Vorstellungen ergänzen und verändern. Hierzu das Beispiel, das im Handbuch zum Turbo Assembler von Borland steht. Wenn Sie z.B. die folgenden C-Prototypen erstellen:

```
void test() {}
void test(int) {}
void test(int, int) {}
void test(float, double) {}
```

und mit der Option »-S« kompilieren, so erhalten Sie folgende Assemblerdatei:

```
@test$qv PROC NEAR
         push  bp
         mov   sp,bp
         pop   bp
         ret
@test$qv ENDP
@test$qi PROC NEAR
         push  bp
         mov   sp,bp
         pop   bp
         ret
@test$qi ENDP
@test$qii PROC NEAR
         push  bp
         mov   sp,bp
         pop   bp
         ret
@test$qii ENDP
@test$qfd PROC NEAR
         push  bp
         mov   sp,bp
         pop   bp
         ret
@test$qfd ENDP
```

Sie könnten nun zwischen die Zeilen, die jeweils den Stackrahmen erzeugen und wieder entfernen, Ihren Assemblerquelltext einbauen. Der Vorteil dieses Vorgehens liegt auf der Hand.

Andernfalls können Sie »normal« linken. *Wichtig ist dann jedoch, daß jede Routine und jedes Datum, das Sie in das C-Modul exportieren wollen, mit einem Unterstrich »_« beginnt.* § 6

```
Data SEGMENT WORD PUBLIC 'Data'
PUBLIC _AsmDate
_AsmDate DW 04711h
Data ENDS

Code SEGMENT WORD PUBLIC 'Code'
PUBLIC _AsmProc
_AsmProc PROC NEAR
         push  bp
         mov   sp, bp
         pop   bp
         ret
_AsmProc ENDP
Code ENDS
END
```

Solchermaßen erzeugte Assemblerroutinen können mittels der EXTERN-Anweisung eingebunden werden, *wobei der Unterstrich nicht erscheinen darf*:

```
extern   int   AsmProc(void)
extern   int   AsmDate
```

HINWEIS

Wir werden noch sehen, daß Sie an den explizit notwendigen Unterstrich nicht denken müssen, wenn Sie die vereinfachten Segmentanweisungen verwenden, die in einem späteren Kapitel (»Vereinfachungen und Ergänzungen«) erläutert werden. Denn in diesem Fall erledigt das der Assembler für Sie! Für den Moment jedoch sei festgehalten: Zu exportierende Labels müssen in C mit einem Unterstrich beginnen.

Vergewissern Sie sich, daß die einzubindenden Routinen in der richtigen Adressierungsart assembliert wurden! Ich kann es nur nochmals betonen: Die meisten Schwierigkeiten beim Verwenden von Assemblerteilen entstehen dadurch, daß die Assemblerroutine nicht korrekt vom Hochsprachenmodul angesprungen wird. *Far*-deklarierte Assemblerroutinen (*_TestRoutine PROC FAR*) müssen von der Hochsprache aus auch *far* angesprungen werden (*extern void far TestRoutine*). § 7

Halten Sie die Assemblermodule möglichst klein! Heutzutage verfügen die meisten C-Dialekte über sogenannte »intelligente« Compiler, die erkennen, welche Routinen im Programm verwendet werden, und nur diese dann auch benutzen. Allerdings können Sie dies nicht mit Assemblermodulen tun. Sie können nicht außerhalb ihres Wirkungskreises erkennen, welche Assemblerroutine überflüssig und daher zu eli- § 8

minieren ist. Denn schließlich ist ja alles, was sie über die Assemblerroutinen in Erfahrung bringen können, die Adresse der exportierten Routinen in der OBJ-Datei. Wie die interne Struktur im Assemblerteil ist, weiß nur der Assembler, und der hat seinen Part schon erledigt! Allerdings kann der Compiler sehr wohl ganze Assemblermodule ignorieren, wenn sie nicht benutzt werden.

TIP Es ist daher guter (und optimierender!) Programmierstil, in einzelne Assemblermodule nur die Routinen zu packen, die für die Tätigkeit der aus ihm exportierten Routine(n) unbedingt erforderlich sind (im besten Fall also eine exportierte Routine pro Modul). Solche Module können Sie dann, wie die *Library* MATHE zeigt, in Bibliotheken sammlen.

§ 9 *Beachten Sie unbedingt, daß der* C-Compiler *zwischen Groß- und Kleinschreibung streng unterscheidet!* Der Assembler tut dies in der Regel nicht, kann jedoch durch Assemblerschalter oder die Nutzung vereinfachter Segmentanweisungen dazu gezwungen werden.

§ 10 *Es ist möglich, in einem Assemblermodul Teile aus der Hochsprache zu benutzen*. Sie müssen in diesem Fall lediglich dem Assembler mitteilen, daß er nun Adressen zu verwenden hat, die der Compiler zur Verfügung stellt. Im Grunde gilt hier das eben für die Richtung Assembler →Hochsprache Gesagte umgekehrt! Daher gibt es auch hier ein Schlüsselwort: EXTRN, das Sie in den Assemblerteilen verwenden. Den Compiler selbst brauchen Sie hier nicht anzuweisen, etwas zur Verfügung zu stellen. Das erfolgt automatisch.

Üblicherweise werden Sie in Assemblermodulen nur auf Daten im Datensegment zurückgreifen, seltener auf Routinen. Falls Sie also z.B. auf das Wort *CDate* zurückgreifen wollen, das im C-Modul definiert wird, melden Sie dieses Datum lediglich im Assembler an unter:

```
Data SEGMENT WORD PUBLIC 'Data'
EXTRN _CDate:WORD
:
Data ENDS
```

Beachten Sie bitte den Unterstrich, der im C*-Teil nicht erscheint!* Da der Assembler ja lediglich die Adresse des Datums erhält, nicht aber die Größe, müssen Sie dies als Programmierer kundtun, indem Sie hinter den Namen den Datentyp (den der Assembler kennt, also nicht etwa den C-Typ!) durch einen »:« getrennt anführen. Sie können nun ganz normal mit dieser Variablen im Assemblertext arbeiten.

ACHTUNG Denken Sie daran, daß dann in DS das korrekte Datensegment stehen und mit ASSUME angemeldet worden sein muß! Allerdings wird dies in der Regel der Fall sein, wenn Sie die Assemblerroutinen einbinden.

Denn eines der ersten Dinge, die C in seinem *Start-up-Code* erledigt, ist, das korrekte Datensegment in DS zu speichern.

HINWEIS Es gibt eine Ausnahme: Wenn Sie Assemblerroutinen programmieren, die Teil des *Start-up-Codes* sind, kann es vorkommen, daß DS noch nicht korrekt belegt wurde! In diesem Fall sollten Sie auf Nummer Sicher gehen und wie üblich das Datensegment setzen:

```
ASSUME DS:Data
mov       ax, SEG DATA
push      ds
mov       ds,ax
:
:
pop       ds
```

DATA ist ein reserviertes Wort und liefert immer die Adresse des Datensegments zurück. Natürlich können auch C-Routinen in Assemblerteilen verwendet werden. Auf die im folgenden definierte C-Routine

```
int       CProc(void);
```

kann in Assemblermodulen zurückgegriffen werden:

```
Code SEGMENT WORD PUBLIC 'Code'
extrn     _CProc:NEAR

PUBLIC _AsmProc
_AsmProc PROC NEAR
          mov   ax, _CDate
          call  _CProc
          ret
_AsmProc ENDP
Code ENDS
END
```

Legen Sie Daten niemals im Codesegment ab, wenn Sie Programme unter Windows *programmieren und nutzen wollen!* Windows läuft üblicherweise im Protected-Mode des Prozessors und verbietet einen schreibenden Zugriff auf das Codesegment. Nutzen Sie das Datensegment, oder erzeugen Sie ein eigenes. **§ 11**

Wenn Sie externe Dateien einbinden wollen, dann tun Sie dies auch! Wie eben schon erwähnt, reicht das Einbinden der Prototypen aus *Headerdateien* oder die EXTERN-Angabe nicht aus. Auch gibt es keine Möglichkeit, etwa analog zu Turbo Pascal die einzubindenden Module im Quelltext anzugeben. Sie müssen dem Linker mitteilen, welche Module er linken soll! Das können Sie entweder aus der Entwick- **§12**

lungsumgebung heraus, indem Sie ein Projekt eröffnen, in dem Sie alle zu verwendenden Routinen eintragen. Alternativ können Sie auch den Kommandozeilenversionen der Compiler bzw. dem betreffenden Linker selbst alle Module angeben.

§ 13 *Beachten Sie bei* C *die Möglichkeit, Parameter von Routinen anstelle über den Stack auch über Register übergeben zu können (__fastcall!).* Hierbei kann es zu Datenverlusten kommen, wenn eine Assemblerroutine unmittelbar nach ihrem Aufruf Register überschreibt, die C zur Wertübergabe verwendet.

TIP Deshalb rät auch Microsoft, die *__fastcall*-Methode nicht zu verwenden, falls Sie Daten an Assemblerroutinen übergeben wollen.

§ 14 *Keine der Bibliotheken, die zur Emulation von Coprozessorbefehlen dienen, emuliert die Befehle FLDENV, FSTENV, FSAVE, FRSTOR, FBLD, FBSTP und FNOP.* Die Verwendung dieser Befehle, auch mit dem *integrierten Assembler*, führt zum Programmabbruch.

27.5 Wie arbeitet der Compiler?

Anders als bei Turbo Pascal ist der C-Compiler tatsächlich nur ein Compiler. Das bedeutet, daß er nur den C-Quelltext in den Code »übersetzt«. Das Ergebnis dieses Vorgangs ist ganz analog zur Aktivität des Assemblers eine OBJ-Datei, in der neben den Tabellen mit den Adressen öffentlicher Daten und Routinen der Code und die Daten verzeichnet sind. Bei C geht diese Analogie zum Assembler so weit, daß Sie den Compiler auch anweisen können, aus dem C-Quelltext Assembler-Quelltext zu erzeugen, der mit dem Assembler dann assembliert werden kann. So könnte man durchaus behaupten, daß der C-Compiler lediglich ein sehr guter Assembler mit riesiger Makro-Bibliothek und automatischer »Suche-und-Ersetze«-Funktion ist. Tatsächlich ruft der C-Compiler auch während seiner Tätigkeit den Assembler auf.

Grundsätzlich jedoch kann auch hier das Schaubild von Seite 358 angeführt werden, mit dem Unterschied, daß in diesem Fall Compiler und Linker tatsächlich getrennt vorliegen! Das bedeutet für die Einbindung, *daß dem* Linker *jedes Modul, also auch die OBJ-Dateien des* Assemblers, *explizit mitgeteilt werden muß*, das er zur Erzeugung der endgültigen Datei benötigt. Hierbei spielt keine Rolle, ob dies über die Kommandozeilenversion des Compilers erfolgt oder im Rahmen der integrierten Entwicklungsumgebung einiger C-Dialekte. Unter Turbo Pascal ist dies nicht nötig, da Compiler und Linker hier eine Einheit bilden und die Informationen über die zu verwendenden Assemblermodule im Quelltext gegeben werden.

28 Assembler und Delphi

Delphi ist ein Entwicklungssystem, das auf der Programmiersprache *Object* Pascal, einem Pascal-Dialekt, basiert – es ist eine konsequente Weiterentwicklung von *Borland* Pascal *7.0* / *Borland* Pascal *for Windows*. Somit ist zu erwarten, daß alles, was zu Assembler und Pascal – genauer Turbo Pascal – gesagt wurde (siehe ab Seite 336), auch für Delphi gültig ist.

In der Tat können die an dieser Stelle gemachten Angaben zu Übergabekonventionen von Parametern und Funktionsergebnissen fast wörtlich übernommen werden. Dies bedeutet, daß alle in Delphi definierten Variablentypen, die in Turbo/Borland Pascal ein Pendant besitzen, sich genau gleich verhalten. Somit gibt es lediglich einige Ergänzungen, die die neu definierten Typen betreffen oder auf Erweiterungen der Sprache basieren. Und dies sind, vor allem wenn man an Delphi 2.0 und 32-Bit-Betriebssysteme denkt, doch einige.

Im folgenden werde ich von Delphi sprechen, wenn die Version Delphi 1.x betroffen ist, und von Delphi 2, wenn es sich um Delphi 2.x handelt. Beide Versionen unterscheiden sich nämlich nicht unerheblich voneinander. Falls notwendig, wird noch zwischen Delphi16 und Delphi 2 unterschieden. Bei Delphi16 handelt es sich um eine Delphi-1.x-Version, in der zwar teilweise die Spracherweiterungen aus Delphi 2 (mit notwendigen Anpassungen) Eingang gefunden haben, die aber zur Erzeugung von 16-Bit-Programmen unter Windows 3.xx benutzt wird und sich somit von dem »echten« Delphi 2.0 wesentlich unterscheidet[10].

Delphi

Als neue Typen sind nun zu berücksichtigen: *ShortInt*, *ByteBool*, *WordBool* und *LongBool*. ShortInt und ByteBool sind 8-Bit-Typen, so daß davon ausgegangen werden kann, daß auf sie auch das zutrifft, was für 8-Bit-Variablen vom Typ *Byte* oder *Char* gesagt wurde. Ebenso verhält es sich mit *WordBool* (entsprechend Word oder Integer) und LongBool (entsprechend *LongInt*).

---

[10] Wenn Sie so wollen, ist Delphi16 ein »aufgebohrtes« Delphi 1.x, das, soweit sinnvoll, die Funktionalität von Delphi 2 besitzt. Sie können es auch als »Schmalspur-Delphi 2« zur Programmierung von Programmen für »alte« Windows-Versionen unter Windows 9x betrachten. Wie auch immer, Delphi 1.x und Delphi16 unterscheiden sich voneinander, auch wenn es sich nicht so offensichtlich darstellt. Woher wissen Sie, daß Sie Delphi16 besitzen? Tip: Rufen Sie die Online-Hilfe des vermeintlichen Delphi 1.x auf, und suchen Sie nach dem Begriff „Cardinal“. Zeigt sich die Delphi-Hilfe kooperativ und gibt Ihnen Informationen zu diesem neuen Typ, haben Sie Delphi16 vorliegen!

Ein weiterer Typ, der zwar schon in Borland Pascal 7.0 definiert wurde, aber erst im Zusammenhang mit Delphi (und den Strings) an dieser Stelle besprochen werden soll, ist *pChar*. pChar ist etwas interessanter als die neuen Typen, ist es doch einerseits per definitionem ein Zeiger auf ein Datum vom Typ Char:

```
Type pChar = ^Char;
```

Andererseits kann pChar aber auch sogenannte »null-terminierte Zeichenketten« referenzieren. Das bedeutet, daß in Delphi eine Zuordnung der Form

```
var S : pChar;

begin
  S := 'Dies ist ein null-terminierter String';
end;
```

möglich ist. Nimmt man die Definition von pChar für bare Münze, so handelt es sich hier trotz der offensichtlich möglichen, direkten Zuweisung einer Zeichenkette an eine Zeigervariable um einen Zeiger, auf den das zutreffen sollte, was auf Seite 339 über Zeiger gesagt wurde. Dies ist auch so, wie sich an folgendem Codefragment zeigen läßt:

```
var aString : pChar;

procedure Test(S : pChar; var S_ : pChar);
begin
  do_something;
end;

begin
  aString := 'Dies ist ein null-terminierter String';
  Test(aString, aString);
end;
```

Beachten Sie bitte, daß hier die strikte Konvention, die man von Pascal gewohnt ist, aufgeweicht wurde! Unter aktivierter Compileroption {$X+} (= erweiterte Syntax) ist ein Datum vom Typ pChar kompatibel zu einem Array[0..n] of Char. Doch soll an dieser Stelle keine Einführung in Delphi gegeben werden!

Das Disassemblat dieses Fragments sieht dann wie folgt aus:

```
CS : 0175 B84000          MOV     AX,0040
     0178 8CDA            MOV     DX,DS
     017A A3480A          MOV     [0A48],AX
     017D 89164A0A        MOV     [0A4A],DX
     0181 FF364A0A        PUSH    [0A4A]
```

```
     0185 FF36480A       PUSH    [0A48]
     0189 BF480A         MOV     DI,0A48
     018C 1E             PUSH    DS
     018D 57             PUSH    DI
```

Die ersten vier Zeilen sind die Zuweisung der absoluten Adresse der Zeichenkette (DS:0040) an die Variable S vom Typ pChar. Wie man sieht, enthält die Speicherstelle [0A48] (= S) tatsächlich den Zeiger (zwei Worte, deshalb 0A48 – 0A4A) auf die Zeichenkette. Mit den nächsten beiden Zeilen wird der Inhalt von S auf den Stack geschoben, was einer Übergabe eines Zeigers *by value* entspricht. Achtung: Der Inhalt von S (selbst ein Zeiger), nicht etwa die Adresse von S (auch ein Zeiger!) wird gepusht! Die letzten drei Zeilen demonstrieren die Übergabe der Variablen S selbst (hier wird tatsächlich der Zeiger auf S gepusht, also DS:0A48). Es ist also alles ganz geradlinig.

ACHTUNG

Das aber bedeutet, daß die Übergabe einer Zeichenkette wie »Dies ist ein String« sich wesentlich davon unterscheidet, ob sie als Datum vom Typ String oder vom Typ pChar übergeben wurde! Denn während Turbo Pascal / Borland Pascal bei der Übergabe von Strings *by value* lokal in der Routine Platz schafft und den String dann kopiert, unterbleibt dies bei »null-terminierten Zeichenketten« (da hier ja de facto ein Zeiger auf die Zeichenkette übergeben wird!).

Die Überprüfung der im Kapitel über Turbo Pascal geschilderten Sachverhalte für Delphi brachte aber eine weitere, erstaunliche Änderung beim Themenkreis Strings ans Licht. Betrachten Sie einmal das folgende Fragment:

```
var aString : String;

procedure Test(S : String; var S_ : String);
begin
  do_something;
end;

begin
  aString := 'Dies ist ein String-String';
  Test(aString, aString);
end;
```

Delphi macht daraus:

```
CS : 01B3 BF360A         MOV     DI,0A36
     01B6 1E             PUSH    DS
     01B7 57             PUSH    DI
     01B8 BF360A         MOV     DI,0A36
     01BB 1E             PUSH    DS
```

```
01BC 57             PUSH    DI
01BD 68FF00         PUSH    00FF
```

Bei der Übergabe *by value* scheint ja noch alles ganz wie erwartet abzulaufen, nämlich Pushen der Adresse des Strings (DS:0A36) auf den *Stack*. Auch die (hier nicht gezeigte) Reservierung von lokalem Platz mit Umkopieren, wie auf Seite 344 demonstriert, findet sich wieder. Doch neu ist, daß bei Übergabe *by reference* ein weiterer Wert auf den Stack geschoben wird. Wie es aussieht, handelt es sich um eine Konstante mit dem Wert 255.

HINWEIS Dies bedeutet, daß offensichtlich bei der Übergabe von Strings *by reference* zusätzlich zur Übergabe der Adresse der Stringvariablen auch die Stringgröße übergeben wird.

Nun stellt sich die spannende Frage, ob dies grundsätzlich bei Strings der Fall ist. Die nicht weniger interessante Antwort lautet: nein! Definiert man

```
Type S1 = String[32];
     S2 = String[255];
     S3 = String;

Var  aString1 : S1;
     aString2 : S2;
     aString3 : S3;

procedure Test(var X : S1; var Y : S2; var Z : S3);
begin
  do_something;
end

begin
  Test(aString1, aString2, aString3);
end;
```

so wird in keinem Fall die Stringgröße mit übergeben, auch nicht bei Variablen vom Typ S3, der ja seinerseits als Typ String definiert wurde. Halten wir also fest, daß dann und nur dann, wenn in der Parameterliste des Prozedurkopfes eine Variable vom Typ String *by reference* ohne Größenangabe übergeben wird, also der Form

```
procedure XYZ(var ABC : String);
```

zusätzlich zum Zeiger auch die Größe des Strings auf den Stack gebracht wird.

Dieses Verhalten ist merkwürdig und läßt sich nur in Verbindung mit einer weiteren Neuerung in Delphi erklären, den »open arrays«. Bei

diesen offenen Feldern kann im Prozedurkopf eine Arrayvariable angegeben werden, die keine Angaben über die Größe des Arrays besitzt. Die Definition lautet somit:

```
procedure UseOpenArray(var X : array of char);
```

Dies ist sehr nützlich, um der gleichen Routine Arrays unterschiedlicher Größe übergeben zu können:

```
Var anArray   : Array[1..100] of Integer;
    anotherA  : Array[0..999] of Integer;
    anInteger : Integer;

function Test(var A : Array of Integer):integer;
begin
  Test := A[High(A)div 2]; {High(A) liefert die Arraygröße}
end;

begin
  anInteger := Test(anArray);
  anInteger := Test(anotherA);
end;
```

Die Funktion `Test` kann, da *A* als offenes Feld definiert wurde, nicht wissen, wie groß das übergebene Array letztendlich ist! Allerdings hat der Programmteil, der `Test` dann aufruft und die entsprechende Variable übergibt, diese Information sehr wohl. Dies ist der Grund, warum nach Ablegen des Zeigers auf dem *Stack* auch die Größe des übergebenen Arrays gepusht wird, bevor in das Unterprogramm verzweigt wird:

```
CS : 01BF BF360A         MOV     DI,0A36
     01C2 1E             PUSH    DS
     01C3 57             PUSH    DI
     01C4 6A63           PUSH    0063    ; $63 = 99
     01C6 E8C5FF         CALL    Test
     01CC BF360A         MOV     DI,0A36
     01CF 1E             PUSH    DS
     01D0 57             PUSH    DI
     01D1 68E703         PUSH    03E7    ; $3E7 = 999
     01D4 E8B7FF         CALL    Test
```

Nun läßt sich prüfen, ob die weiter oben gemachten Annahmen zu Strings auch tatsächlich stimmen. Ersetzt man nämlich die Arraydeklarationen durch Stringdeklarationen, etwa

```
Var aString   : String[50];
    anotherS  : String[150];
```

```
procedure Test(var S : String);
begin
  do_something_with_S;
end;

begin
  Test(aString);
  Test(anotherS);
end;
```

so findet man auch hier, ohne es an dieser Stelle durch ein Disassemblat belegen zu wollen, daß die Stringgrößen 50 und 150 nach der Ablage der Stringadressen und vor dem Unterprogrammaufruf auf den Stack geschoben werden.

TIP Da Arrays in Pascal grundsätzlich »null-basierend« sind (egal, ob sie als solche auch definiert werden!), d.h. der Index auf das erste Element immer 0 ist, wird bei offenen Arrays immer die um 1 verminderte Arraygröße übergeben (also 99 und 999, obwohl die Arrays 100 bzw. 1000 Elemente besitzen). Bei Strings dagegen, so sie echte Pascal-Strings und nicht etwa »null-terminierte« pChars sind, befindet sich an Position 0 des Char-Arrays die Stringgröße. Daher steht der erste echte Index an Position 1, und der übergebene Wert ist die tatsächliche Größe der Zeichenkette, also 50 bzw. 150.

ACHTUNG Das alles bedeutet, daß Sie bei der Nutzung von Arrays und Strings als Parameter in Assemblermodulen aufpassen müssen, ob offene Arrays/Strings verwendet werden oder nicht. Dementsprechend ändern sich die Stelle auf dem Stack, an der der Zeiger auf das Array steht, und auch die Anzahl zu entfernender Parameter im RET-Befehl. Und noch etwas! Logischerweise macht dies alles nur bei einer Übergabe *by reference* Sinn und findet daher auch nur dort statt.

Noch ein Wort zu den Funktionswerten in Delphi. Hier ist dem auf Seite 347 Gesagten nur soviel hinzuzufügen, als daß sich die neuen Typen ShortInt, ByteBool, WordBool und LongBool absolut in die Reihe der bekannten Typen gleicher Größe einreihen lassen. Das heißt: ShortInt und ByteBool werden in AL, WordBool in AX und LongBool in DX:AX zurückgegeben.

Nun noch ein paar Verwaltungsdetails, falls Sie Assemblerteile in Delphi-Programme einbinden wollen. Beachten Sie bitte, daß

- der Code in Segmenten mit den Namen CODE, CSEG oder xxx_TEXT steht, wobei xxx ein beliebiger Name sein kann (z.B. MyModule_TEXT).

- gleiches für Daten und Konstanten gilt: Daten werden in Segmenten namens DATA, DSEG oder xxx_BSS abgelegt, Konstanten (d.h. initialisierte Daten) in CONST oder xxx_DATA. *Bitte beachten Sie die Unterschiede zwischen den Segmenten DATA und xxx_DATA!*
- keine GROUP-Anweisung im Quelltext vorhanden sein darf!
- das Alignment für Codesegmente Byte zu sein hat und für Daten Word.
- Methodenbezeichner nicht durch einen Punkt, sondern durch das »@«-Zeichen definiert werden. Die Routine MyObject.Init wird in Assembler also durch MyObject@Init angesprochen. Grund: der Punkt hat im Assembler eine andere Bedeutung.

HINWEIS

Delphi ignoriert ebenso wie Turbo Pascal / Borland Pascal jegliche Initialisierung im Datensegment! Sie sollten daher von vornherein bei der Definition von Daten diese durch »?« als uninitialisiert markieren:

```
.DATA

ThatsOK DW ?
Ignored DW 4711
```

`Ignored` ist für den Compiler einfach nicht existent! In Code- und Konstantensegmenten dagegen ist eine Wertzuweisung erlaubt.

Delphi16

Es existieren verschiedene Versionen von Delphi 1.x, die sich aber nicht grundlegend voneinander unterscheiden und deshalb auch nicht genauer betrachtet werden müssen. Während die vor Delphi 2.0 aktuelle und ausgelieferte Version von Delphi[11] nur die oben beschriebenen zusätzlichen Typen kennt, so wird mit Delphi 2.0 (Verzeichnis »Delphi16« auf meiner CD-ROM) eine Version ausgeliefert[12], die zusätzlich noch folgende Typen kennt: Cardinal und SmallInt. Diese Typen werden wir im nächsten Absatz zusammen mit Delphi 2 besprechen. Der Grund für diese »überarbeitete« Version von Delphi, die Borland als Delphi16 bezeichnet, ist eine gewisse »Aufwärtskompatibilität« zu Delphi 2, wird sie doch zusammen mit diesem ausgeliefert. Auf diese Weise können Quelltexte, die für Delphi 2 und somit Windows NT/95 geschrieben wurden, auch für das 16-Bit-Windows 3.xx kompiliert werden. Leider konnte sich Borland aber nicht zu vollständiger Aufwärtskompatibilität durchringen: Alle weiteren neuen Typen von Delphi 2 provozieren bei Verwendung in Delphi16 einen Fehler des Compilers!

---

[11] Meine DELPHI.EXE ist vom 17.02.95 und 1.195.168 Bytes groß.

[12] DELPHI.EXE ist hier vom 14.09.95 und 1.97.280 Bytes groß.

Delphi 2

Trifft alles, was zu Delphi festgestellt wurde, denn nun auch auf das 32-Bit-Entwicklungssystem Delphi 2 zu? Antwort von Radio Eriwan: »Im Prinzip ja, doch es wird noch etwas komplizierter!« Denn zum einen wurden in Delphi 2 notwendige Erweiterungen der Sprache vorgenommen, um wirklich für 32-Bit-Betriebssysteme tauglich zu sein – Stichwort: »Maximale Größe von 64 kByte aufgrund der 64-kByte-Segmente entfällt.« Andererseits wurden verschiedene sinnvolle Veränderungen vorgenommen, die in 32-Bit-Systemen sinnvoll sind – Stichwort: »Warum sollte eine ganz normale Integer in einem Prozessor, der 32-Bit-Register hat, nur 16 Bit breit sein? Nur aus Tradition?« Ferner fanden einige neue *Features* Einzug in Delphi, die sehr sinnvoll sind. Schließlich der Zwang zur Kompatibilität, da Delphi 2, respektive die auf Delphi 2-Syntax aufgesetzte Version von Delphi ja auch 16-Bit-Versionen der Programme für Windows 3.xx erstellen können soll – Stichwort: »Nicht jedes unter Windows 95 entwickelte Programm soll nur unter Windows 95 lauffähig sein!«

Gehen wir also das Thema Variable an. Welche Neuigkeiten gibt es da unter Delphi 2? Neue Datentypen sind:

- Cardinal: Dieser Typ ist wohl am besten mit dem Typ Word in Delphi zu vergleichen. Es ist eine vorzeichenlose Zahl mit Werten zwischen 0 und 2.147.483.647 (= 2^{31}). Frage: Warum nur 2^{31} und nicht 2^{32}? Antwort: Da Windows 95 dem Benutzer nur 2 GByte Adreßraum zur Verfügung stellt, darf er auch nur in solchen Größenordnungen mit Integerzahlen[13] rechnen – er könnte ja sonst etwas Böses anfangen wollen. Daher haben die Delphi-Entwickler dem Compiler hier Schranken auferlegt! Falls Sie größere Integers brauchen, müssen Sie auf Comp ausweichen.

HINWEIS

In Variablen vom Typ Cardinal können tatsächlich 32-Bit-Daten gespeichert werden! Falls Sie ein Feld mit 32 Bits brauchen und mit den Bitschiebebefehlen wie ROR oder SHL manipulieren müssen – kein Problem! Auch das Abspeichern von 32-Bit-Zahlen ist möglich. Nur »arithmetisch rechnen« läßt sich mit 32 Bit nicht. Das ist ein Eingriff in die Freiheit des Programmierers, der, wie ich meine, Borland nicht zusteht und der auch nicht sein muß! Hoffentlich ist dies in der nächsten Version bereinigt.

- SmallInt: Eine vorzeichenbehaftete 16-Bit-Zahl im Bereich zwischen -32768 und 32767. Somit ist SmallInt mit dem Typen Integer aus Delphi und Pascal identisch.

---

[13] »Integer« ist hier im Gegensatz zu den Fließkommazahlen der FPU (floating point unit; »Coprozessor«) zu sehen. Auch eine nicht vorzeichenbehaftete Zahl ist eine Integerzahl, wenn auch im weiteren Sinne.

- Currency: Ein neuer Typ, der kaufmännische Berechnungen genauer machen soll (und es sicherlich auch tut!). Es handelt sich hierbei im Prinzip um den Typ Comp, bei dem durch den Compiler lediglich Skalierungen vorgenommen werden, um vier feste Nachkommastellen zur Verfügung zu stellen. Vorteil: Currency-Daten werden in der FPU (Floating-Point-Unit) manipuliert, die mit genauerer Arithmetik arbeitet!
- AnsiChar: Das ist der aus Pascal und Delphi gewohnte Typ Char. Es gibt diesen neuen Typ deshalb, um ihn von einem weiteren neuen Typ zu unterscheiden, der sehr sinnvoll ist:
- WideChar: Dieser Typ dient zur Darstellung von Zeichen nach dem sog. Unicode, bei dem ein Zeichen nicht mit 7 (ASCII) bzw. 8 (ANSI) Bits dargestellt wird, sondern mit 16. Auf diese Weise können auch Zeichen verarbeitet werden, die in anderen Schriftsätzen verwendet werden (Arabisch, Hebräisch, Japanisch etc.).

  TIP Kein Mensch weiß heute, welche Delphi-Version standardmäßig etwas unter Char verstehen wird. Es kann also durchaus sein – nein, es wird ziemlich sicher so kommen, daß – analog zu Integer – Char demnächst das gleiche wie WideChar ist. Gehen Sie also auf Nummer Sicher, wenn Sie definitiv AnsiChars verwenden wollen, und deklarieren Sie Variablen dieses Typs entsprechend. Falls Sie aber »Auf- und Abwärtskompatibilität« benötigen, greifen Sie auf Char zurück.
- ShortString: Ist die neue Bezeichnung für den guten alten Pascal-String, also ein Array of Char, bei dem in Position 0 des Arrays die Stringgröße verzeichnet ist.
- AnsiString: Verläßt man die Pascal-Tradition der Längenangabe für Strings an der Stelle 0 des Arrays und führt die aus C übernommene Angabe von »null-terminierten Strings« mit dem ersten Zeichen an der ersten Stelle des Strings ein, so ist ein aus AnsiChar bestehender, null-terminierter String ein AnsiString – und somit kompatibel zu pChar in Delphi. Dennoch ist ein AnsiString auch irgendwie ein »echter« Pascal-String (siehe Anhang J).

Was passierte mit Integer, LongInt, Word, ShortInt, Byte und String? Zum Teil gibt es sie noch in ihrer üblichen Definition (LongInt, ShortInt, Word, Byte), zum Teil gibt es sie mit veränderten Eigenschaften (Integer, String):

- Integer ist ein Zwitter! Abhängig vom gewählten Zielbetriebssystem ist Integer entweder eine vorzeichenbehaftete Zahl im Bereich -32.786 bis 32.767 (16 Bit, Delphi16) und somit zu Integer aus Delphi kompatibel oder im Bereich -2.147.493.648 bis 2.147.493.647 (32 Bit, Delphi 2) und damit zu LongInt kompatibel.

- Ähnlich janusköpfig ist String! Abhängig vom Zustand des Compilerschalters {$H} ist String nun der gute alte Pascal-String ({$H-}) oder ein null-terminierter, neuer AnsiString ({$H+}). Allerdings funktioniert das nicht unter Delphi16.

Ich versuche nun, da der Mensch ein optisch orientiertes Wesen ist, das Ganze in der folgenden Tabelle zusammenzufassen:

Typ	Delphi 2	Delphi 16	Delphi
LongInt[1]	-2.147.483.648	-2.147.483.648	-2.147.483.648
	+2.147.483.647	+2.147.483.647	+2.147.483.647
Integer	-2.147.483.648	-32.768	-32.768
	+2.147.483.647	+32.767	+32.767
SmallInt	-32.768	-32.768	-
	+32.767	+32.767	
ShortInt	-128	-128	-128
	+127	+127	+127
Currency	-922.337.203.685.477,5808		-
	+922.337.203.685.477,5897-		
Cardinal	0	0	-
	2.147.483.647	65.535	
Word	0	0	0
	65.535	65.535	65.535
Byte	0	0	0
	255	255	255
String	{$H-}: 255 Zeichen[2]	255 Zeichen[2]	255 Zeichen[2]
	{$H+}: unbegrenzt[3]	-	-
ShortString	255 Zeichen[2]	-	-
AnsiString	unbegrenzt[3]	-	-
Char	8 Bit	8 Bit	8 Bit
AnsiChar	8 Bit	-	-
WideChar	16 Bit	-	-

[1] Bitte beachten Sie, daß in Delphi 2 jede 32-Bit-Zahl, also auch LongInts, in ein 32-Bit-Register (z.B. EAX) geschrieben werden, während in Delphi 1.x und Delphi16 zwei 16-Bit-Register (z.B. DX:AX) verwendet werden.

[2] Dies ist der »alte« Pascal-String, den wir aus allen bisherigen Pascal-Versionen kennen.

[3] Dies ist der »neue« Pascal-String, ein Hybrid aus Pascal- und null-terminiertem String.

Auf der nächsten Seite finden Sie eine Tabelle, die die Kompatibilität der einzelnen Typen in den verschiedenen Delphi-Versionen aufzeigen soll.

Vor allem bei Verwendung von Daten des Typs String, Cardinal und Integer müssen Sie also in Zukunft sehr genau aufpassen, für welche Plattform gerade entwickelt wird. In Anhang J gehe ich etwas detaillierter auf die neuen Stringtypen ein.

Was heißt dies nun für die Übergabekonventionen? Uns braucht hier nur zu interessieren, was mit den neuen Stringtypen passiert, alles andere dürfte aus dem bisher Gesagten logisch ableitbar sein. Wichtig ist

jedoch bei Delphi 2, daß ganz konsequent 32-Bit-Adressen zum Einsatz kommen, was bedeutet, daß unter anderem konsequent von den 32-Bit-Registern Gebrauch gemacht wird!

Delphi 2	Delphi16	Delphi
LongInt	LongInt	LongInt
Integer		LongInt
	Integer	Integer
SmallInt	SmallInt	Integer
ShortInt	ShortInt	ShortInt
Cardinal		-
	Cardinal	Word
Word	Word	Word
Byte	Byte	Byte
String ({$N-})	String	String
({$N+})	pChar/String	pChar/String
ShortString	-	String
AnsiString	-	pChar/String

Vergessen Sie ab jetzt AX, DL oder SI. Unter Windows 95 wird nur noch in absoluten Ausnahmefällen mit Byte- oder Wortregistern gearbeitet. Ab jetzt zählen nur noch EAX, EBP, EFlags und Konsorten – einschließlich eines doppelwortorientierten Stacks! Das bedeutet z.B., daß nun drei Bytes undefiniert sind, wenn Sie ein Byte auf den Stack schieben: Nur das niedrigstwertige Byte des Doppelworts auf dem Stack enthält einen gültigen Wert. Nicht anders verhält es sich mit Worten – die oberen 2 Bytes auf dem Stack sind nicht definiert. Aber Analoges kennen wir ja schon vom wortorienterten Stack aller bisherigen Prozessoren, wenn Bytes gepusht/gepoppt werden.

Schauen wir uns ein kleines Beispiel an, in dem ein paar neue und alte Parametertypen an eine Routine übergeben werden:

```
var W : word;
    D : Double;
    S : String;
    A : AnsiString;
    C : Cardinal;

procedure Test(    W : Word; var D : Double; var S : String;
               var A : AnsiString; var C : Cardinal);
begin
  do_something;
end;


begin
```

```
  W := 1;
  D := 2;
  S := 'String';
  A := 'ANSI-String';
  C := 3;
  Test(W,D,S,A,C);
end;
```

Nun folgt das, was Delphi 2 daraus macht. Beginnen wir mit den Zuweisungen, und halten wir die Adressen fest, die die Variablen repräsentieren. Über die Zuweisung der Werte ist dies recht einfach möglich:

```
:0041F320 66C70564164200+mov  word ptr [00421664],0001
:0041F329 33C0           xor  eax,eax
:0041F32B 890568164200   mov  [00421668],eax
:0041F331 C7056C16420000+mov  dword ptr [0042166C],40000000
:0041F33B B870164200     mov  eax,00421670
:0041F340 BA8CF34100     mov  edx,0041F38C
:0041F345 E8E63EFEFF     call @LStrAsg
:0041F34A B874164200     mov  eax,00421674
:0041F34F BA9CF34100     mov  edx,0041F39C
:0041F354 E8D73EFEFF     call @LStrAsg
:0041F359 C7057816420003+mov  word ptr [00421678],00000003
```

Zunächst einmal: Fällt Ihnen an diesem Listing etwas auf? Schauen Sie sich doch einmal die Adressen oben an! Lineare 32-Bit-Adressen! Wir arbeiten mit dem 32-Bit-Betriebssystem Windows 9x! Sonst etwas Ungewöhnliches? Nein? Na sehen Sie – es bleibt also auch unter Windows 9x vieles beim alten, auch wenn man nun vermehrt 32-Bit-Register sieht wie EAX, EDX etc.! Doch nun zum Wesentlichen.

Zeile 1: Zuweisung an `W`, `W` hat die Adresse `$00421664`; Zeile 2 bis 4: Zuweisung an `D`, `D` hat die Adresse `$00421668`; Zeile 5 bis 7: Die String-Konstante wird an Adresse `$00421670`, die Variable `S`, zugewiesen und in Zeile 8 bis 11 der andere String an Adresse `$00421674`, die Variable `A` (offensichtlich ein unter {$H-} kompilierter »alter« Pascal-String). Schließlich erfolgt in Zeile 12 die Zuweisung an die Variable `C` an Adresse `$00421678`. Wozu das Ganze – wozu die Adressen merken? Schauen Sie sich nun einmal den Aufruf von Test an:

```
:0041F363 6874164200     push   00421674
:0041F368 6878164200     push   00421678
:0041F36D B970164200     mov    ecx,00421670
:0041F372 BA68164200     mov    edx,00421668
:0041F377 66A164164200   mov    ax,[00421664]
:0041F37D E88EFFFFFF     call   Test
```

Die Adresse der Variablen A wird auf den Stack gepusht, dann die Adresse der Variablen C. Mehr kommt nicht auf den Stack! In ECX ist die Adresse von S und in EDX die von D. AX enthält den Wert von W. Dann der Sprung zu Test. Das ist absolut nicht mit dem in Einklang zu bringen, was wir bisher über die Übergabekonventionen gehört haben. Eigentlich wäre zu erwarten gewesen, daß in der Reihenfolge W, D, S, A und C die Werte/Adressen auf den Stack geschoben werden – und nun das!

ACHTUNG

Delphi 2 erlaubt als erste Pascal-orientierte Sprache aus dem Hause Borland die Übergabe von Parametern an Routinen über Register. Mehr noch, dies ist sogar die Grundeinstellung. Das bedeutet, daß, wann immer ein Wert in ein Register paßt, dieses zur Übergabe verwendet wird. Insgesamt drei Register stehen hierzu zur Verfügung: EAX, ECX und EDX.

Das bedeutet für das Beispiel oben, daß – von rechts nach links gehend, geprüft wird, ob ein Wert bzw. eine Adresse über eines der Register übergeben werden kann. Vollziehen wir dies nach: W wird *by value* übergeben, also ein Wort. Dieses paßt sicherlich in ein Register, so daß EAX dafür verwendet wird – das erste der möglichen »Übergaberegister« (da W nur ein Wort belegt, wird auch nur AX verwendet!). D ist eine Variable vom Typ Double, die *by reference* übergeben wird, also (als Zeiger) ebenfalls in ein 32-Bit-Register paßt und daher auch über Register übergeben werden kann – in EDX. Schließlich die Adresse von S, ebenfalls ein pointer, der ebenfalls in ein Register, ECX, paßt. Alle anderen Parameter müssen über den Stack übergeben werden, was auch erfolgt.

Hätte man diese neue Übergabekonvention explizit ausgeschaltet und die Pascal-Konvention gewählt, so hätten wir nur Vertrautes vorgefunden:

```
:0041F363 66A164164200  mov   ax,[00421664]
:0041F369 50            push  eax          ; Inhalt von W
:0041F36A 6868164200    push  00421668     ; Adresse von D
:0041F36F 6870164200    push  00421670     ; Adresse von S
:0041F374 6874164200    push  00421674     ; Adresse von A
:0041F379 6878164200    push  00421678     ; Adresse von C
:0041F37E E88DFFFFFF    call  Test
```

HINWEIS

Beachten Sie also ganz genau, unter welcher Übergabekonvention Sie in Delphi 2 die Parameter an Routinen übergeben. Es ist nämlich neben der Pascalschen Methode und der, bei der Register möglichst intelligent benutzt werden, auch die Möglichkeit in C gegeben, bei der die Parameter in umgekehrter Reihenfolge auf den Stack geschrieben

werden. Denken Sie auch an die Verantwortlichkeit, den Stack nach dem Austritt aus der Routine zu bereinigen; wir haben diese Problematik in den vorangehenden Kapiteln über Pascal und C schon ausgiebig erörtert. Falls Sie nicht gewaltig aufpassen, liegen hier leicht übersehbare Programmierfehler!

TIP Es ist somit ratsam, bei Routinen, die in irgendeiner Weise mit Assemblerteilen zusammenarbeiten müssen, die Automatik auszuschalten und eine bestimmte Übergabekonvention durch Angabe des entsprechenden Schlüsselwortes im Prozedurkopf explizit vorzugeben. Falls tatsächlich die Notwendigkeit zur Optimierung bestehen sollte, können dann in einem zweiten Schritt gezielt Register zur Übergabe verwendet werden – nachdem alles andere fehlerfrei funktioniert!

Doch kommen wir wieder zur Übergabe von Parametern zurück, falls diese über den Stack übergeben werden. Falls Sie dies nicht schon getan haben, sollten Sie an dieser Stelle kurz unterbrechen und in Anhang J ab Seite 960 das Kapitel über die neuen Stringtypen in Delphi lesen, es hilft, das folgende besser zu verstehen. Dazu betrachten wir folgendes Programm (beachten Sie bitte die Pascal-Konvention durch Angabe des Schlüsselwortes im Funktionskopf!):

```
procedure Test(C : Cardinal;    var _C : Cardinal;
               X : Currency;    var _X : Currency;
               P : pChar;       var _P : pChar;
               S : ShortString; var _S : ShortString;
               A : AnsiString;  var _A : AnsiString;
               V : Variant;     var _V : Variant);    pascal;
begin
  do_something;
end;

begin
  aCardinal := 1;
  aCurrency := 12.345587;
  aChar :='S';
  aPChar := 'Dies ist ein pChar';
  aString := 'Dies ist ein kurzer String';
  anAnsiS := 'Dies ist ein AnsiString';
  Test(aCardinal, aCardinal,
       aCurrency, aCurrency,
       aPChar,    aPChar,
       aString,   aString,
       anAnsiS,   anAnsiS,
       aVariant,  aVariant);
end;
```

Schauen wir nun, was Delphi 2 daraus macht: Beginnen wir mit der Zuweisung von Werten an die Parameter. In der ersten der folgenden Zeilen wird der Wert 1 an die Variable vom Typ Cardinal zugewiesen, in den anschließenden drei Zeilen der Wert 123456 an die Variable vom Typ Currency. Beachten Sie bitte, daß der Compiler

- erstens den Wert automatisch auf vier Nachkommastellen rundet und
- zweitens die mit dem Faktor 10,000 multiplizierte Zahl ($00001E240 = 123456) zuweist:

```
:0041F3C2 C7056416420001+mov  dword ptr 00421664],00000001
:0041F3CC C7056816420040+mov  dword ptr [00421668],0001E240
:0041F3D6 33C0           xor  eax,eax
:0041F3D8 89056C164200   mov  [0042166C],eax
```

HINWEIS

Das »+«-Zeichen hinter den Bytes der Anweisungen in Zeile 1 und 2 bedeutet, daß hier eigentlich noch Zeichen darzustellen wären, die aber (dank des *Flat-Models* mit 32-Bit-Adressen = 8 Ziffern pro Adresse) nicht vollständig in die Zeile paßten. So hieße Zeile 2 beispielsweise vollständig

```
:0041F3CC C7056816420040E20100   mov dword ptr ...
```

Um das Gesamtbild aber nicht durch zwei vollständige 32-Bit-Adressen zu stören, schneidet der Turbo Debugger die Adresse also ab. Aufpassen!

Als nächstes wird in der ersten Zeile das ASCII-Zeichen »S« an `aChar` zugewiesen, dann die Adresse der null-terminierten Zeichenkonstante an `aPChar`:

```
:0041F3DE C6057016420053 mov    byte ptr [00421670],53
:0041F3E5 B868F44100     mov    eax,0041F468
:0041F3EA A374164200     mov    [00421674],eax
```

Diesen Teil kennen Sie nun schon: die Zuweisung eines (echten, alten Pascal-) Strings an eine Variable. Hier sieht man schön, wie optimierend der Compiler arbeitet. Statt stumpfsinnig 27 Bytes (1 Längenbyte und 26 Zeichen) mittels `MOVSB` zu kopieren, wird sechsmal `MOVSD`, einmal `MOVSW` und einmal `MOVSB` ausgeführt:

```
:0041F3EF BE7CF44100     mov    esi,0041F47C
:0041F3F4 BF78164200     mov    edi,00421678
:0041F3F9 B906000000     mov    ecx,00000006
:0041F3FE F3A5           rep movsd
:0041F400 66A5           movsw
:0041F402 A4             movsb
```

Schließlich erfolgt die Zuweisung des AnsiStrings an die Variable:

```
:0041F403 B878174200      mov    eax,00421778
:0041F408 BAA0F44100      mov    edx,0041F4A0
:0041F40D E81E3EFEFF      call   @LStrAsg
```

Nach den Zuweisungen nun der Aufruf der Routine `Test` mit dem vorbereitenden Laden des Stacks. Erstens: Übergabe von `aCardinal` in den ersten beiden Zeilen *by value*, danach *by reference* (Adresse von `aCardinal`). Alles funktioniert sehr geradlinig:

```
:0041F412 A164164200      mov    eax,[00421664]
:0041F417 50              push   eax
:0041F418 6864164200      push   00421664
```

Zweitens: Übergabe von `aCurrency` zunächst *by value* (die ersten beiden Zeilen: 8 Bytes!), dann *by reference*, ebenso geradlinig:

```
:0041F41D FF356C164200    push   dword ptr [0042166C]
:0041F423 FF3568164200    push   dword ptr [00421668]
:0041F429 6868164200      push   00421668
```

Drittens: Übergabe von `aPChar` *by value* und *by reference* – das Zeichen `aChar` zu übergeben ist trivial und braucht nicht eigens beschrieben zu werden. Beachten Sie bitte, daß bei einer Übergabe *by value* die Adresse der Stringkonstanten übergeben wird, bei einer Übergabe *by reference* die der Variablen `aPChar`. Achtung: Beides sind Adressen!

```
:0041F42E A174164200      mov    eax,[00421674]
:0041F433 50              push   eax
:0041F434 6874164200      push   00421674
```

Viertens: `aString` ist an der Reihe. Hier sehen wir keinen Unterschied zu dem, was schon für Delphi festgestellt wurde – Übergabe der Adresse von `aString`, auch wenn dieses *by value* übergeben wird – die aufgerufene Routine `Test` sorgt dann für das Kopieren des Strings in eine lokale Variable. Anschließend erfolgt die Übergabe der Adresse von `aString` mit der Stringgröße (»Offene Arrays«!) bei Übergabe *by reference*.

```
:0041F439 6878164200      push   00421678
:0041F43E 6878164200      push   00421678
:0041F443 68FF000000      push   000000FF
```

Bei den neuen Pascal-Strings sieht das Ganze aus wie bei pChars, also zeigt sich auch hier intern, daß AnsiStrings Chimären aus »echten« Pascal-Strings und pChars sind. Zunächst erfolgt (*by value*) die Übergabe der Adresse des Strings, gespeichert in `anAnsiS`, dann die Adresse von `anAnsiS` selbst (*by reference*).

```
:0041F448 A178174200      mov    eax,[00421778]
:0041F44D 50              push   eax
:0041F44E 6878174200      push   00421778
```

Schließlich noch die Übergabe der Variante `aVariant`. Wie bei Records und anderen strukturierten Parametern auch, erfolgt hier, unabhängig davon, ob der Parameter *by value* oder *by reference* übergeben wird, die Übergabe grundsätzlich mit der Adresse der Variablen selbst:

```
:0041F453 687C174200      push   0042177C
:0041F458 687C174200      push   0042177C

:0041F45D E8D2FEFFFF      call   Test
```

Das Unterprogramm selbst bietet nichts wesentlich Neues.

Noch ein Wort zu den Funktionsergebnissen: Wie bei allen anderen Pascal-Versionen aus der Borland-Schmiede erfolgt die Rückgabe von Funktionswerten wie gehabt (siehe Turbo/Borland Pascal); mit einer kleinen Ausnahme:

ACHTUNG

Beachten Sie bitte, daß 32-Bit-Daten, also Zeiger und LongInts, nicht mehr in den zwei 16-Bit-Registern DX:AX zurückgegeben werden, sondern im 32-Bit-Register EAX!

Fassen wir kurz zusammen:

- Gewöhnliche Typen wie Bytes, Words, Integers, Cardinals etc. werden je nach Größe in AL, AX oder EAX zurückgegeben. Hierzu zählen selbstverständlich auch Booleans und Chars.
- Realtypen (also Singles, Doubles und Extended) werden im TOS des Coprozessors zurückgegeben, Currency (die in Wirklichkeit eine Comp ist, aber eben als solche doch als Realzahl des Coprozessors gilt) wird mit 10.000 skaliert (1,2345 = 12345!). Die Umsetzung des zurückgegebenen Wertes in 1,2345 erfolgt an gegebener Stelle durch eingestreute Codeteile durch den Compiler.
- Strings (also echte Strings, pChars und AnsiStrings), Methodenzeiger und Varianten: Bei diesen Typen wird der Routine eine zusätzliche »Variable« für das Funktionsergebnis beim Aufruf übergeben.
- Alle Zeiger werden in EAX als vollständiger 32-Bit-Zeiger übergeben.
- Felder, Records und Mengen werden in AL übergeben, wenn sie aus einem Byte bestehen, in AX (2 Bytes), in EAX (4 Bytes) oder als zusätzlicher Parameter vom Typ Zeiger, falls sie größer sind.
- Der Parameter SELF, der bei Methoden eine Rolle spielt, ist immer der letzte übergebene Parameter und wird in Form eines Zeigers auf den Stack geschrieben. ACHTUNG: Bei aktiver REGISTER-

Konvention ist es immer der erste Parameter und wird somit in EAX übergeben. Nach Pascal-Konvention wird er nach allen anderen Parametern als letzter übergeben, gegebenenfalls nach zusätzlichem RESULTAT-Parameter. Unter der CDecl- und StdCall-Konvention steht er an letzter Stelle, aber vor dem zusätzlichen Ergebnisparameter (Result).

HINWEIS Prozeduren und Funktionen müssen EBX, ESI, EDI, EBP sichern, sie können aber EAX, EDX und ECX verändern. CLD wird beim Aufruf der Routine gelöscht und muß beim Rücksprung ebenfalls gelöscht sein.

Wie man sieht, ist die Nutzung der 32-Bit-Architektur kein großes Problem! Die Adressen werden meist automatisch vom Compiler berechnet, die Übergabekonventionen sind, falls nicht beabsichtigt, nicht wesentlich anders, als man bisher gewohnt ist. Lediglich die konsequente Nutzung von 32-Bit-Registern muß beachtet werden.

29 Der integrierte Assembler

Die meisten Hochsprachen bieten die Möglichkeit, Assemblerquelltext auch direkt in den Hochsprachenquelltext aufzunehmen. Hierzu verfügt die Hochsprache über Schlüsselwörter, mit denen der Compiler angewiesen werden kann, den daran anschließenden Block nicht als Hochsprachentext anzusehen, sondern ihn vielmehr einem »eingebauten Assembler« zu übermitteln.

Grundsätzlich können Sie mit diesen *integrierten Assemblern* genauso arbeiten wie mit externen. Dennoch ergeben sich einige Unterschiede und Einschränkungen, die auf die Eigenheiten der Hochsprache Rücksicht nehmen und auf ihnen basieren.

Turbo Pascal Unter Turbo Pascal können Sie im Pascal-Text auf den internen Assembler zurückgreifen, indem Sie, analog zu *begin – end*, die Assemblerbefehle in einen *asm – end*-Block einbauen. Die Routine muß dann jedoch mittels des Schlüsselworts *assembler* als Assemblerroutine deklariert werden:

```
function AsmProc:byte; assembler;
asm
  mov    al,002h
end;

var b : byte;
begin
  b := AsmProc;
end.
```

Das Compilat des Programms zeigt das gleiche Bild, das wir erhalten hätten, wenn wir *AsmProc* extern assembliert und eingebunden hätten:

```
CS : 0000 B002          MOV       AL,02
     0002 C3            RET
     0003 9A0000D312    CALL      12D3:0000
     0008 55            PUSH      BP
     0009 89E5          MOV       BP,SP
     000B E8F2FF        CALL      0000
     000E A25000        MOV       [0050],AL
     0011 5D            POP       BP
     0012 31C0          XOR       AX,AX
     0014 9A1601D312    CALL      12D3:0116
```

Fett gedruckt sehen Sie die Assemblerroutine, die hier minimal ausfällt. Das sich anschließende Hauptprogramm führt nur einen Bibliotheksaufruf aus und richtet einen Stackrahmen ein, bevor dann die Routine aufgerufen wird. Beachten Sie bitte, daß bei dieser Art der Assemblereinbindung

- kein RET-Befehl programmiert werden darf, da dies aufgrund des *end* durch den Compiler bewerkstelligt wird
- kein Platz für ein Funktionsergebnis reserviert wird
- kein Code zum Kopieren von Inhalten von Variablen erzeugt wird, die *by value* übergeben werden und deren Größe 1, 2 oder 4 Bytes überschreitet (siehe hierzu die ausführliche Diskussioin im vorletzten Kapitel: Strings, *arrays*, *records* etc.). Sie müssen in diesem Fall die Kopierarbeit entweder selbst vornehmen oder die Parameter als *nicht zu verändernde var*-Parameter ansehen (da der Compiler davon ausgeht, daß niemand den Inhalt von Variablen verändert, die *by value* übergeben werden, selbst wenn dies analog der *by-reference*-Übergabe durch Zeiger erfolgt!)[14]
- niemand *automatisch* einen Stackrahmen einrichtet, wenn *keine* Parameter übergeben werden, Sie dies also ggf. wie bei externen Assemblermodulen auch selbst programmieren müssen
- *sehr wohl* ein Stackrahmen eingerichtet wird, wenn Sie Parameter übergeben oder lokale Variablen definieren.

---

[14] Beachten Sie in diesem Zusammenhang bitte folgenden Unterschied. Wenn Sie einer Routine z.B. einen String *by value* übergeben, so erzeugt der Compiler immer dann eine lokale *String*kopie und den Kopiercode, wenn die Routine nicht als *assembler*-Routine definiert ist und somit einen *begin - end*-Block enthält, auch wenn unmittelbar nach dem *begin* ein *asm - end*-Block folgt! Die gleiche Routine als *assembler*-Routine realisiert, erzeugt diesen Code niemals!

Spezifizieren wir diesen Punkt noch genauer! Der Compiler erzeugt bei Assembler-Routinen (also Routinen, die mittels *assembler* deklariert werden) je nach Situation unterschiedlichen Ein- und Austrittscode:

- Ein Stackrahmen wird nur dann erzeugt (*push bp – mov bp,sp*) und entfernt (*pop bp*), wenn innerhalb des Blocks lokale Variablen erzeugt werden oder Parameter über den Stack übergeben werden. Ansonsten entfällt dieser Start- und Endcode!
- Werden lokale Variablen verwendet, so ergänzt sich der Code zur Erzeugung des Stackrahmens um *sub sp,locals*; analog werden vor dem Entfernen des Rahmens mit *mov sp,bp* die reservierten Bereiche freigegeben.
- Ein abschließendes RET wird in jedem Fall erzeugt. Je nachdem, ob die Routine *far* oder *near* deklariert ist, verwendet der Compiler automatisch den richtigen RET-Befehl. Er modifiziert ihn auch ggf. so, daß übergebene Parameter vom Stack geholt werden.

HINWEIS Auf diese Weise lassen sich ganze Routinen als Assemblermodule realisieren. Sie müssen hierbei wenig selbst berücksichtigen, jedoch genau entscheiden, ob Sie einen Stackrahmen benötigen und ob er automatisch angelegt wird.

Doch auch innerhalb »normalen« Codes können Sie mit dem *asm-end*-Block Assemblerteile einstreuen:

```
function AsmProc:byte;
var local : byte;
begin
  local := 1;
  asm
      cmp local,001h
      jz  @L1
      mov local,002h
  @L1:
  end;
  AsmProc := local
end;

var b : byte;

begin
  b := AsmProc;
end.
```

In diesem Programm wurde in einer ganz normalen Pascal-Prozedur Assemblercode eingestreut. Das Compilat zeigt dies nicht mehr:

```
CS : 0000 55            PUSH    BP
     0001 89E5          MOV     BP,SP
     0003 83EC02        SUB     SP,+02
     0006 C646FE01      MOV     BYTE PTR [BP-02],01
     000A 807EFE01      CMP     BYTE PTR [BP-02],01
     000E 7404          JZ      0014
     0010 C646FE02      MOV     BYTE PTR [BP-02],02
     0014 8A46FE        MOV     AL,[BP-02]
     0017 8846FF        MOV     [BP-01],AL
     001A 8A46FF        MOV     AL,[BP-01]
     001D 89EC          MOV     SP,BP
     001F 5D            POP     BP
     0020 C3            RET
     0021 9A0000D512    CALL    12D5:0000
     0026 55            PUSH    BP
     0027 89E5          MOV     BP,SP
     0029 E8D4FF        CALL    0000
     002C A25000        MOV     [0050],AL
     002F 5D            POP     BP
     0030 31C0          XOR     AX,AX
     0032 9A1601D512    CALL    12D5:0116
```

Wie Sie sehen, beginnt die Funktion »ganz normal« mit dem Stackrahmen und der Einrichtung von Platz für Funktionsergebnis und lokale Variable. Der fett gedruckte Teil ist auch hier der Assemblerteil. Beachten Sie bitte die fast verrückt anmutende, kursiv gedruckte Sequenz, mit der die Konstante erst in der lokalen Variablen (*bp-2*) gespeichert wird, um dann gegen Ende über den Platz für das Funktionsergebnis (*bp-1*) endgültig an den Übergabeort expediert zu werden. Wir handeln dieses Verhalten des Turbo Pascal-Compilers in Kürze (»Optimierungen beim Programmieren«) ausführlich ab!

Wie Sie sehen können, ist in Pascal die Arbeit mit dem integrierten Assembler kein Problem. Sie brauchen sich, wie eben demonstriert, auch um die Variablennamen keine Sorgen mehr zu machen. Dennoch gibt es Einschränkungen:

- Labels haben nur innerhalb des aktuellen *asm – end*-Blocks Gültigkeit. Sie müssen, wie in Pascal, im Labeldeklarationsteil der Routine definiert sein, wenn es sich nicht um lokale *Labels* handelt, die mit @ beginnen!
- Der integrierte Assembler unterstützt nur 8086/8087- und 80286/80287-Code. Die erweiterten Befehle ab den 80386/80387-

Prozessoren können nicht genutzt werden! Die 80286/80287-Befehle selbst stehen allerdings auch nur dann zur Verfügung, wenn sie mittels des Compilerschalters *{$G+}* aktiviert werden.

- Der integrierte Assembler kennt nur die Deklarationsanweisungen *DB*, *DW* und *DD*. Allerdings können diese Anweisungen *nur* dazu benutzt werden, Code im Codesegment zu definieren, *nicht aber* Daten im Datensegment. Somit erübrigen sich alle anderen Anmerkungen (*DF*, *DP*, *DQ* und *DT* sowie der Operator *DUP* und *THIS*). Daten können jedoch im Pascal-Quelltext »ganz normal« deklariert werden.
- Makros können nicht verwendet werden, der integrierte Assembler ist *kein* Makro-Assembler!
- Sie können in Assemblerblöcken mit den Symbolen AL, AH, AX etc. auf die Prozessorregister zurückgreifen. Vermeiden Sie daher die Benennung von *Variablen* mit diesen Symbolen, um Konflikte zu vermeiden.
- Das Schlüsselwort *DATA*, mit dem Sie im externen Assembler die Adresse des Datensegments erhalten können, heißt im integrierten Assembler *@Data*. Es ist ein vollständiger Zeiger mit Segment- und Offsetanteil, so daß Sie den Segmentanteil über *SEG @Data* eruieren müssen. Gleiches gilt für *CODE* bzw. *@Code*.
- Sie können nur auf Pascal-Routinen zurückgreifen, die in dem aktuellen Segment (in dem die Assembleranweisungen stehen) deklariert sind. Somit können Sie Funktionen und Routinen aus CRT oder DOS z.B. nicht aufrufen!
- Sie können keine String-, Realzahl- oder Mengenkonstanten verwenden. Ferner sind Labels, die außerhalb des aktuellen Assemblerblocks deklariert sind, hier nicht bekannt!

Von diesen Änderungen abgesehen, können Sie den integrierten Assembler genauso benutzen wie den externen.

Visual C++ Auch in *C* ist ein Assembler heutzutage eingebaut. Man kann wohl erwarten, daß mit ihm mindestens ebenso einfach gearbeitet werden kann, wie mit dem Inline Assembler von Turbo Pascal, da C ja wesentlich assemblernäher ist als Pascal.

Sie haben genauso wie bei Turbo Pascal zwei Möglichkeiten, Assemblercode in Ihren Quelltext einzubauen:

```
_ _asm
{
   mov   ax,08300h
   mov   cx,01000h
   xor   dx,dx
```

```
    lds    bx,Semaphore
    int    015h
}
```

Zum einen können Sie das Schlüsselwort _ _*asm* benutzen, um mit Hilfe der geschweiften Klammern einen ganzen Assemblerblock zu definieren. Beachten Sie bitte hierbei, daß vor *asm zwei* Unterstriche stehen müssen! Dieses Beispiel lädt in DX:CX eine Zeitspanne (hier: 4000 Mikrosekunden), die der Prozessor warten soll, bevor das Byte in `Semaphore` verändert wird. Dazu wird die Funktion $83 des ROM-BIOS-Interrupts $15 benutzt. Andererseits können Sie auch vor jede Anweisung dieses Schlüsselwort stellen:

```
_ _asm    mov    ax,00002h
_ _asm    mov    dx,00007h
_ _asm    int    021h
```

Hier wird mittels der Funktion 2 des DOS-Interrupts $21 ein Piepton erzeugt. Kommen wir nun zu einigen Einschränkungen, die der *Inline* Assembler von Visual C++ besitzt.

- Auch der integrierte Assembler von Visual C++ verarbeitet maximal 80286/80287-Code. Wie auch bei Turbo Pascal müssen Sie allerdings diesen erst durch das Setzen des Compilerschalters */G2* anfordern. Andernfalls steht Ihnen nur 8086/8087-Code zur Verfügung.
- Sie können in Assemblerblöcken mit den Symbolen AL, AH, AX etc. auf die Prozessorregister zurückgreifen. Vermeiden Sie daher die Benennung von *Variablen* mit diesen Symbolen, um Konflikte zu vermeiden.
- Daten können mit dem eingebauten Visual C++-Assembler ebenfalls nicht deklariert werden. *DB, DW, DD, DQ, DT* und *DF* sowie *DUP* und *THIS* sind somit obsolet!

  ACHTUNG Im Gegensatz zu Turbo Pascal und Turbo C++ ist unter Visual C++ auch DB im integrierten Assembler verboten. Um jedoch auch Code-Bytes einstreuen zu können, gibt es die Pseudoinstruktion *_emit*. Diese kann jedoch nur jeweils ein Byte adressieren, weshalb Byte-Folgen durch mehrere *_emit*-Instruktionen erzeugt werden müssen:

  ```
  _ _asm _emit 00Fh
  _ _asm _emit 0A2h
  ```

  Diese Sequenz könnte dazu benutzt werden, die Bytes *0F – A2* in den Code zu integrieren (dies ist der Opcode für den neuen Befehl CPUID beim Pentium).

- Makros können nicht verwendet werden, der integrierte Assembler ist *kein* Makro-Assembler!

- *Spezifische* C- oder C++-Operatoren, so z.B. die Verschiebeoperatoren << und >>, sind in Assembler nicht definiert und können daher auch im integrierten Assembler nicht verwendet werden.
- Falls Sie *C*-Symbole (also Routinen und/oder Daten) in einem _ _*asm*-Block verwenden wollen, müssen diese vor dem Block definiert sein, da der integrierte Assembler sonst davon ausgeht, daß es sich um ein Label innerhalb des Blocks handelt.
- Symbole mit der gleichen Schreibweise (unabhängig von der Groß-/ Kleinschreibung) wie reservierte Assemblerwörter (z.B. PROC, END etc.) können nicht als C-Symbole verwendet werden.
- Assembler-Labels haben, wie in C auch, globale Gültigkeit. Das heißt, daß Sie aus dem C-Modul in und aus dem _ _*asm*-Block springen können. Vermeiden Sie daher für Labels Namen, die in C schon anderweitig reserviert sind, vor allem Routinen- und Datennamen aus Bibliotheken! Ein häufiger Fehler ist das beliebte EXIT-Label zum Ende einer Assemblerroutine, das den Rücksprung in den rufenden Programmteil markiert. Da EXIT eine C-Systemfunktion ist, führt die Deklaration von EXIT im Assemblerteil zu Konflikten.
- C-/C++-Routinen können von *asm*-Blöcken nur aufgerufen werden, wenn sie global verfügbar sind und im Falle der objektorientierten Variante nicht überladen werden.

ACHTUNG

- _ _*asm*-Blöcke werden vom C-Compiler nicht optimiert! Sie werden unverändert in das Compilat an der Stelle eingefügt, an der sie deklariert wurden. Register werden auch nicht, wie sonst unter C üblich, vor dem Aufruf einer Routine lokal gesichert und nach ihrer Ausführung wieder restauriert. Dies haben ggf. *Sie* zu erledigen. Denken Sie hierbei an die Inhalte von SI und DI.

Turbo C++

Natürlich kann man auch unter Turbo C++ den *Inline Assembler* (der hier BASM heißt) benutzen. Auch hier dient *asm* als Schlüsselwort. Aber ohne Unterstriche, dafür mit der offenen geschweiften Klammer *in der gleichen Zeile wie asm*! Also:

```
asm {
   mov   ax,08300h
   mov   cx,01000h
   xor   dx,dx
   lds   bx,Semaphore
   int   015h
}
```

Auch die andere, zeilenorientierte Version gibt es. Konsequenterweise ohne Unterstriche, dafür jedoch mit erlaubtem DB:

```
asm DB 00Fh
asm DB 0A2h
```

Aber es gibt, verglichen mit dem *Inline Assembler* von Turbo Pascal, einen großen Unterschied. Das, was in Turbo Pascal als Assemblerroutine bezeichnet wird (*procedure Test; assembler;*), ist in Turbo C++ nicht möglich! *Asm*-Blöcke und *asm*-Anweisungen können nur innerhalb von geschweiften Klammern der Routinen stehen. Somit werden wie bei vergleichbaren reinen *C*-Routinen ggf. ein Stackrahmen eingerichtet und Register gerettet. Schauen wir uns dies eimal an:

```
unsigned char AsmProc(void);
unsigned char B;

void main(void)
{
  B = AsmProc();
}
unsigned char AsmProc(void)
{
  asm  mov  al,0x02
}
```

Das Disassemblat sieht dann so aus:

```
CS : 02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 E80500        CALL    02CD
     02C8 A28C02        MOV     [028C],AL
     02CB 5D            POP     BP
     02CC C3            RET
     02CD 55            PUSH    BP
     02CE 8BEC          MOV     BP,SP
     02D0 B002          MOV     AL,02
     02D2 5D            POP     BP
     02D3 C3            RET
```

Die ersten sechs Zeilen sind `main()`. Nach dem Aufruf von `AsmProc` wird das Funktionsergebnis in *B* gesichert. Die fett gedruckte Zeile ist die *asm*-Direktive! Beachten Sie bitte, daß durch die geschweiften Klammern *{* und *}* ein Stackrahmen eingerichtet und entfernt wird! Es ist ein ganzer Rahmen, wenn auch keine lokalen Variablen erzeugt werden und somit SP nicht verändert wird. Betrachten wir ein anderes Beispiel, in dem ein Parameter übergeben wird:

```
unsigned char AsmProc(unsigned char I);
unsigned char B;
```

```
void main(void)
{
  B = AsmProc(1);
}

unsigned char AsmProc(unsigned char I)
{
  unsigned char local;
  local = I;
  asm {
        cmp local,0x01
        jz  L1
        mov local,0x02
      }
  L1:
  return(local);
}
```

Das Disassemblat sieht so aus:

```
CS : 02C2 55            PUSH    BP
     02C3 8BEC          MOV     BP,SP
     02C5 B001          MOV     AL,01
     02C7 50            PUSH    AX
     02C8 E80600        CALL    02D1
     02CB 59            POP     CX
     02CC A28C02        MOV     [028C],AL
     02CF 5D            POP     BP
     02D0 C3            RET
     02D1 55            PUSH    BP
     02D2 8BEC          MOV     BP,SP
     02D4 83EC02        SUB     SP,+02
     02D7 8A4604        MOV     AL,[BP+04]
     02DA 8846FF        MOV     [BP-01],AL
     02DD 807EFF01      CMP     BYTE PTR [BP-01],01
     02E1 7404          JZ      02E7
     02E3 C646FF02      MOV     BYTE PTR [BP-01],02
     02E7 8A46FF        MOV     AL,[BP-01]
     02EA EB00          JMP     02EC
     02EC 8BE5          MOV     SP,BP
     02EE 5D            POP     BP
     02EF C3            RET
```

Verglichen mit dem vorangehenden Beispiel schiebt `main()` hier lediglich noch den Parameter auf den Stack und entfernt ihn nach dem Aufruf von `AsmProc` wieder. Auch hier zeigt sich die zwanglose Inte-

gration des *asm*-Blocks, der im Disassemblat fett hervorgehoben ist. Dies soll als Anschauungsmaterial ausreichen. Wenden wir uns daher den Hinweisen zu!

- Auch der integrierte Assembler von Turbo C++ verarbeitet maximal 80286/80287-Code. Wie bei den anderen besprochenen Compilern müssen Sie dies aber durch Compilerschalter (*options*/Compiler/ *advanced code generation*) oder Kommandozeilenparameter (»-Z«) anfordern. Andernfalls steht Ihnen nur 8086/8087-Code zur Verfügung.
- Sie können in Assemblerblöcken mit den Symbolen AL, AH, AX etc. auf die Prozessorregister zurückgreifen! Vermeiden Sie daher die Benennung von Variablen mit diesen Symbolen, um Konflikte zu umgehen.
- *DQ, DT* und *DF* sowie *DUP* und *THIS* sind nicht definiert.

ACHTUNG

Das heißt jedoch nicht, daß man keine Daten deklarieren könnte! Im Gegensatz zu Visual C++ ist analog zu Turbo Pascal in Turbo C++ *DB, DW* und *DD* beim integrierten Assembler erlaubt! Werden diese Anweisungen außerhalb von gültigen Routinen benutzt, so verwendet *BASM* sie zur Deklaration der entsprechenden Bytes im Datensegment, andernfalls im Codesegment. Dies bedeutet, daß mit *BASM* auch Daten generiert werden können.

HINWEIS

Wozu diese Möglichkeit gut sein soll, ist mir allerdings ein Rätsel! Denn sicherlich dürfte es konsistenter sein, Daten im C-Quelltext in C-Manier zu deklarieren anstatt sie mittels des *Inline Assemblers* außerhalb von Funktionen über Assemblerdirektiven zu erzeugen!

- Makros können nicht verwendet werden, der *integrierte* Assembler ist *kein* Makroassembler!
- *Spezifische* C- oder C++-Operatoren, so z.B. die Verschiebeoperatoren << und >>, sind in Assembler nicht definiert und können daher auch im integrierten Assembler nicht verwendet werden.
- Symbole mit der gleichen Schreibweise (unabhängig von Groß-/Kleinschreibung) wie reservierte Assemblerwörter (z.B. PROC, END, etc.) können nicht als C-Symbole verwendet werden.
- Innerhalb von *asm*-Blöcken können *keine* Labels deklariert werden! Dies bedeutet, daß im Gegensatz zum *Inline Assembler* von Turbo Pascal und Visual C++ mehrere Blöcke generiert werden müssen, die mit C-Labels beginnen:

```
:
:
A: asm {
      dec  [Counter]
```

```
        jz   B:
        jmp  A:}
    B: asm {
        :
        : }
    :
    :
```

`A:` und `B:` sind hier ganz »normale« C-Labels, die jeweils auf einen eigenen *asm*-Block zeigen. Da C-Labels global definiert sind, gelten sie auch innerhalb des *asm*-Blocks.

- C-/C++-Routinen können von *asm*-Blöcken nur aufgerufen werden, wenn sie global verfügbar sind und im Falle der objektorientierten Variante nicht überladen werden!

TIP Ein Rat an dieser Stelle: Verwenden Sie für größere Module oder komplexere Routinen besser den externen Assembler als den integrierten. Sie sind damit erheblich flexibler und können Module unabhängig von der Hochsprache entwickeln. Außerdem sind in der Regel die externen Assembler wesentlich leistungsfähiger (Beispiel: Makros, bedingte Assemblierung!) und erzeugen kompakteren Code. Was eigentlich auch nicht verwundert, denn externe Assembler müssen sich ja nicht um die Eigenheiten der verwendeten Hochsprache kümmern! Ferner können Sie auch 80386-/80387-Code erzeugen, was mit den *Inliners* nicht geht.

Interne Assembler dagegen haben ihren unschätzbaren Wert immer dann, wenn kurze, prägnante Codefragmente in einen Hochsprachenteil integriert werden sollen oder der Aufwand für ein eigenes Assemblermodul zu groß wäre. So läßt sich z.B. mit zwei Zeilen integriertem Assembler ein Type-Casting der gehobenen Art realisieren:

```
function Exponent(X:Extended):Integer; Assembler;
var Y : Extended;
    I : Integer;
begin
  :
  Y := Log(X);
  asm
        fld   Y
        fistp I
  end;
  :
end;
```

Oder es ist z.B. sehr viel schneller eine »integrierte Assemblerroutine« verwirklicht, wie sie in der Unit MATHE verwendet wird und im folgenden abgedruckt ist. Hier ein eigenes Modul zu erstellen hieße, Daten

EXTRN zu deklarieren, Routinen PUBLIC, ein Daten- und Codesegment zu generieren, eventuell noch darauf zu achten, daß die Modelle und Übergabekonventionen stimmen usw. Sicherlich ist das für diesen Zweck mit mehr Aufwand behaftet als im »integrierten« Fall.

```
procedure SetMaxFixDisplay(Exp:Integer); Assembler;
asm
     mov        cx,[Exp]
     fild       [K_10]
     fld1
     cmp        cx,$0000
     js         @B
     jz         @D
@A:  fmul       st,st(1)
     loop       @A
     jmp        @D
@B:  neg        cx
@C:  fdiv       st,st(1)
     loop       @C
@D:  fstp       [M_Fix_]
     ffree      st
end;
```

Delphi

Der integrierte Assembler von Delphi verhält sich, wie wäre es auch anders zu erwarten, sehr ähnlich wie der von Turbo Pascal. Im Prinzip lassen sich also nur noch einige Delphi-spezifische Angaben an dieser Stelle anführen:

- Wie Turbo Pascal kann der integrierte Assembler 80286-Befehle nutzen, wenn dies per Compileroption {$G+} ausdrücklich erwünscht wird. Andernfalls ({$G-}) ist der Befehlsvorrat auf 8086-Befehle beschränkt. Analoges gilt für 80287-Befehle ({$N+} / {$N-}).
- Falls in Routinen die Register BP, SP, SS oder DS verändert werden müssen, so sind sie vorher zu sichern. Der Assembler verläßt sich darauf, daß alle »von außen« durchgeführten Manipulationen an diesen Registern wieder in Ordnung gebracht werden.

Delphi 2

Auch unter Delphi 2 gibt es keine grundsätzlichen Änderungen außer denen, die mit dem *Flat-Memory-Modell* und der 32-Bit-Architektur zu tun haben:

- Es können 80386-/80387-Befehle genutzt werden. Ein Compilerschalter ist bei Delphi 2 nicht zu setzen, da es grundsätzlich nur auf Prozessoren ab 80386 lauffähig ist (*Flat-Memory-Modell*!).
- Falls in Routinen die Register EDI, ESI, EBP oder EBX verwendet werden sollen, sind sie vorher zu sichern!

30 Optimierungen beim Programmieren

Während Hochsprachen über eine gewisse »Intelligenz« in der Weise verfügen, daß man sich als Programmierer keine allzu großen Sorgen über das »Zusammenfügen« und Zusammenwirken einzelner Routinen zu und in einem Programm machen muß, ist das Arbeiten mit dem Assembler in sehr viel stärkerem Maße auf die Intelligenz des Programmierers angewiesen. Was sinnvoll ist und was nicht, muß der Programmierer hier selbst entscheiden. In Hochsprachen wurde ihm dies in starkem Maße von den Compilerbauern abgenommen.

Die sinnvolle Programmierung in und mit Assembler ist nicht einfach – sie setzt ein gewisses Maß an Bereitschaft voraus, sich mit dem Produkt seines Wirkens auch wirklich auseinandersetzen zu wollen. Dafür belohnt einen die Assemblerprogrammierung in sehr vielen Fällen mit hervorragend optimiertem Programmcode. Assemblerprogrammierung erfordert Lust, Zeit und Intelligenz, was nicht jeder hat, der auch programmiert…

30.1 »Optimierte« Compiler oder »optimiertes Denken«?

Ich möchte diesen Abschnitt mit der Schilderung einer Begebenheit beginnen, die eine weit verbreitete Definition von Optimierung des Compilats eines Compilers am Beispiel von Turbo Pascal erläutern soll. Im November 1992 veranstaltete die Firma Borland eine Präsentation, auf der das neue Borland Pascal offiziell in Deutschland vorgestellt wurde. Nach der eigentlichen Demonstration scharten sich dann noch einige hartgesottene Freaks, Programmierer und Anwender um die versammelten Borland-Mitarbeiter, um noch das Letzte an Informationen aus diesen herauszuholen.

Auch ich gesellte mich daher zu einem dieser Kondensationskeime und wurde Zeuge der vehementen Meinungsäußerung eines Teilnehmers. Das Thema war gerade auf den Assembler gekommen. Freddi Ertl, damals einer der zuständigen Borland-Mitarbeiter, hatte eben nochmals festgestellt, daß Borland einen Assembler mit Borland Pascal standardmäßig als integriertes und *Stand-alone*-Produkt auslieferte. Die Frage von seiten des Diskutanten lautete nun, wann denn Borland nun endlich »richtige« Pascal-Compiler programmiere, die auch »optimierten Code« erzeugten, ähnlich wie in C. Recht vorwurfsvoll kritisierte er die Politik, der Hochsprache einen Assembler beizulegen, statt dem Programmierer die Notwendigkeit für den Einsatz eines solchen Hilfsmittels abzunehmen. Es half kein Argument des Borland-Mitarbeiters oder anderer Umstehender, die Assemblern gegenüber nicht so ablehnend gegenüberstanden! Dieser Zeitgenosse hatte seine *idée fixe* und ließ sich nicht irritieren.

Auch wenn ich viel Verständnis für die geäußerte Kritik habe und diesem Anti-Assemblierer teilweise sogar Recht geben muß – verstanden, was es mit Optimierung auf sich hat, hat dieser Zeitgenosse nicht! Und ob nun C »optimaleren« Code erzeugt als Pascal, ist auch noch nicht erwiesen – denn auch C ist eine Hochsprache und somit bestimmten Konventionen unterworfen. Zweifellos kann man mit C flexibler arbeiten als mit Pascal. Aber »optimierter«? Sicherlich wird es versierten Programmierern niemals schwer fallen, die Schwächen einer Programmiersprache mit den Stärken einer anderen zu vergleichen, um so zu »beweisen«, daß die eigene Lieblingssprache besser ist als alles andere auf der Welt. Da meine (traditionell bedingten) Neigungen eher in Richtung Pascal zielen (»Was gibt es in C, das ich mit Assembler und Pascal nicht auch realisieren kann? Genauso optimal? Warum also ein Umstieg?«), bilde ich mir durchaus ein, Beispiele aufführen zu können, in denen Turbo Pascal deutlich kompakteren und optimierteren Code erzeugt als Turbo C. Schließlich ist ja seit Windows z.B. die Pascalsche Übergabekonvention von Parametern auch unter C salonfähig geworden!

Sicherlich habe auch ich manchmal recht belustigt im Compilat herumgestöbert, das der Turbo Pascal-Compiler beispielsweise erzeugt. Auch ich mußte heimlich grinsen, wenn in einer Codesequenz ein Zwischenergebnis zunächst auf den Stack gepusht wurde, nach dem Abarbeiten einer Routine von dort wieder geholt wurde, zunächst dann in DI gerettet und von dort im folgenden Befehl in DX kopiert wurde, bevor es endgültig über AX der aufrufenden Routine übergeben wurde. Ein eindrucksvollens Beispiel hierfür haben wir ja auf Seite 401 kennengelernt!

Oder schauen Sie sich einmal das folgende kleine Turbo Pascal-Programm an:

```
program dummy;

var i:byte;


function return_2:byte;
begin
  return_2:=2;
end;
begin
  i:=return_2;
end.
```

Das Programm deklariert eine Byte-Variable *i* und besitzt die Funktion *return_2*, die nichts weiter tut, als das Byte 2 zurückzugeben. Im

eigentlichen Programm nun wird die Variable *i* mit dem Funktionsergebnis dieser Funktion belegt. Eigentlich nichts Schlimmes, sollte man meinen – oder? Was aber macht der Compiler daraus?

```
PUSH    BP
MOV     BP,SP
MOV     AX,2
SUB     SP,2
MOV     BYTE PTR [BP-1],AL
MOV     AL,BYTE PTR [BP-1]
MOV     SP,BP
POP     BP
RET
:
:
:
CALL    0000
MOV     [004E],AL
:
:
```

Zunächst wird die Funktion `return_2` kompiliert. Mit der Sequenz `push bp; mov bp,sp` wird ein Stackrahmen erzeugt, also ein Bereich, der dieser Funktion »gehört«. Dann wird die Konstante 2 in `AL` geladen. Mit `sub sp,2` wird auf dem Stack Platz geschaffen, in dem theoretisch eine lokale Wort- oder zwei lokale Byte-Variablen Platz finden. Praktisch verwendet die Routine tatsächlich nur ein Byte! Nun kommt das, was unser so kritischer Zeitgenosse bemängelt: der Compiler legt diese Konstante lokal ab (`mov byte ptr [bp-1],al`), um sie im nächsten Befehl wieder zurückzuholen (`mov al, byte ptr [bp-1]`). Schließlich wird der Stackrahmen wieder entfernt und die Funktion mit `ret` beendet. Weiter unten findet sich dann der Aufruf dieser Funktion und die Abspeicherung des Ergebnisses in der Variablen *i*.

Wozu diese Byteschieberei? Hier könnte man im ersten Moment tatsächlich am Sachverstand der Leute zweifeln, die Turbo Pascal entwickelt haben. Der Grund für dieses Konstrukt ist eigenlich ganz einfach: der Compiler kann nicht (und soll es auch gar nicht können!) vorausschauend und interpretierend kompilieren! Woher nun soll er dann aber wissen, daß nach dem Eintrag der Konstanten 2 in `AL` die Funktion beendet wird? Immerhin könnte `return_2` auch so weitergehen:

```
function return_2:Byte;
begin
  return_2:=2;
  if not ok then return_2:=0;
end;
```

Es ist zwar nicht besonders logisch, was da folgt, es zeigt jedoch, daß der Compiler nicht sicher sein kann, daß mit der Zuweisung eines Wertes als Funktionsergebnis die Funktion auch tatsächlich beendet wird. Im obigen Fall würde der Compiler `return_2` etwa so kompilieren:

```
push    bp
mov     bp,sp
mov     ax,2
SUB     SP,2
MOV     BYTE PTR [BP-1],AL
CMP     BYTE PTR [004F],1
JE      +4
MOV     BYTE PTR [BP-1],0
MOV     AL,BYTE PTR [BP-1]
MOV     SP,BP
POP     BP
RET
```

Das heißt also, daß der Compiler um der Flexibilität willen in manchen Fällen, so bei der ersten Deklaration von `return_2`, ein paar Bytes verschenkt. Daß dies nicht so tragisch ist, wie man nach dem geschilderten Beispiel annehmen könnte, werden wir noch sehen! Denn in Wirklichkeit produziert der Compiler im allgemeinen recht kompakten und optimierten Code – zumindest seit Turbo Pascal 6.0.[15]

An dieser Stelle könnte ich nun anfangen, C-Programme zu analysieren. Man wird auch schnell fündig! Schauen Sie sich als einfaches Beispiel an, was der Compiler macht, wenn Sie ein *return(1)* ausführen lassen: Eintrag des Wertes 1 in AL und *unbedingter Sprung zum unmittelbar folgenden Befehl* (meistens eine Ende-Sequenz der Routine). Natürlich gibt es auch hier zwingende Gründe dafür, auf die ich nun jedoch nicht weiter eingehen möchte.

Ob der geschilderte Zeitgenosse, wenn er von Optimierung des Compilats spricht, sich klar darüber ist, daß sich das weit gravierender in anderen Punkten äußert, weiß ich nicht. Ich wollte ihn danach nicht fragen – mir wurde irgendwann das Lamentieren zuviel! Ich kann mir aber vorstellen, wie er z.B. folgendes Problem gelöst hätte.

---

[15] C behandelt Funktionsergebnisse ganz anders. Hier wird üblicherweise keine lokale Variable angelegt. Dennoch werden Mechanismen benutzt, die teilweise denen aus Pascal ähneln. Es würde aber viel zu weit führen, im Rahmen dieses Buches nun Details von C im Vergleich zu Pascal zu klären! Dies ist ein Assembler-Buch! Nur so viel: Auch C-Code läßt sich ähnlich stark »optimieren« wie Pascal-Code!

Stellen Sie sich vor, Sie wollen ein Grafikprogramm herstellen, beispielsweise zur Darstellung von Kurven. Einerlei welche Gleichungen Sie dabei verwenden – Sie kommen um die Berechnung von Funktionswerten nicht herum! Nehmen wir z.B. die allgemein beliebten *Splines*, mit denen »glatte« Kurven durch Punkte gelegt werden können, die sich durch keine »einfache« Gleichung anpassen lassen. Dies erfolgt, indem alle Punkte mit Kurven verbunden werden, die der bekannten Formel für kubische Funktionen $y = ax^3 + bx^2 + cx + d$ folgen. Durch geeignete Wahl der Randbedingungen lassen sich dann Kurven so aneinandersetzen, daß sie einen nahtlosen Übergang besitzen.

Wie erfolgt nun die Berechnung der Funktionswerte zwischen den vorgegebenen Punkten? Schließlich soll ja eine Kurve auf dem Bildschirm ausgegeben werden, nicht eine Ansammlung von Punkten! Natürlich durch Berechnung von Funktionswerten nach der oben genannten Funktionsgleichung für kubische Funktionen. Also liegt doch nichts näher, als z.B. in Turbo Pascal eine Funktion der Art

```
function Cubic(a,b,c,d,x:LongInt):LongInt;
begin
  Cubic := a * x * x * x + b * x * x + c * x + d;
end;
```

zu deklarieren. LongInts werden hier verwendet, weil der Bildschirm ja pixelweise angesprochen wird, gebrochene Zahlen gar nicht vorkommen können, Realzahlen also unnötig (und rechenaufwendiger) sind. So würde wohl fast jeder das Thema angehen, zumal unter Turbo Pascal ja Potenzen nicht standardmäßig berechnet werden können. Vielleicht aber kommt jemand auf die Idee, dieses Manko »in einem Aufwasch« mit bereinigen zu können (was allerdings das Ganze nur verschlimmert!):

```
function Power(x,y:LongInt):LongInt;
var i ,s: LongInt;
begin
  s := 1;
  for i := 1 to y do s := s * x;
  Power := s;
end;
function Cubic(a,b,c,d,x:LongInt):LongInt;
begin
  Cubic := a * Power(x,3) + b * Power(x,2) + c * x + d
end;
```

Auch hier reichen uns LongInts zur Berechnung der Potenzen. Ich bin mir fast sicher, daß unser »Optimierer« nun superkritisch das Compilat durchsucht und auch Stellen findet, an denen er nachweisen kann, daß der Compiler keinen »optimierten« Code erzeugt.

Doch was soll das? Gut – wenn ein paar Bytes in einer Routine gespart werden, die häufig genug aufgerufen wird, das macht sich schon bemerkbar! Falls also der Compiler erkennen würde, daß er das Funktionsergebnis nicht lokal zu speichern braucht, wenn die Routine fertig ist, ließen sich ein paar Prozessortakte einsparen. Dennoch kann ich in einem solchen Fall nur die Amerikaner zitieren, die zu dieser Ersparnis »Peanuts!« sagen würden. Denn mit etwas Hirnschmalz läßt sich eine Routine programmieren, die mit Bruchteilen der Rechenzeit einer der beiden Lösungen von oben auskommt. Die Idee besteht, zeitaufwendige Multiplikationen zu vermeiden und statt dessen die schnelleren Additionen zu benutzen!

Fangen wir z.B. mit einer Geraden an. Sie folgt der Gleichung $y = m \cdot x + b$, wobei m die Steigung der Geraden und b ihr y-Achsenabschnitt ist. y läßt sich hier mit einer Multiplikation und einer Addition berechnen. Aber wir wollen ja zum Zeichnen der Geraden Funktionswerte an mehreren Stellen berechnen, die alle den gleichen Abstand voneinander haben: $x_{i+1} - x_i = \delta$. Das bedeutet, daß sich die x-Werte durch wiederholtes Addieren eines »Inkrements« errechnen lassen: $x_{i+1} = x_i + \delta$. Wir brauchen also nur einen Startwert und das Inkrement.

Doch was passiert mit y? Für den Startwert x_0 ist y einfach berechenbar: $y_0 = mx_0 + b$. Für y_1 ergibt sich dann $y_1 = mx_1 + b$, was aber nach dem eben Gesagten als $y_1 = m(x_0 + \delta) + b$ umgeschrieben werden kann. Multipliziert man die Klammer aus, so erhält man $y_1 = mx_0 + m\delta + b$. Wenn man die Terme etwas anders anordnet, resultiert hieraus $y_1 = mx_0 + b + m\delta$. Dies aber ist $y_1 = y_0 + m\delta$ oder ganz allgemein: $y_{i+1} = y_i + m\delta$! Das bedeutet, daß man nur noch einmal für den Startpunkt die Formel $y_0 = mx_0 + b$ benutzen sowie einmal die Konstante $m\delta$ berechnen muß. Alle anderen Funktionswerte ergeben sich dann durch pure Addition dieser Konstanten zum vorangehenden Funktionswert! Multiplikationen werden somit verhindert.

Was für Geraden recht ist, ist für Parabeln billig! Auch hier kann man mit etwas Mathematik die Funktionsgleichung $y = ax^2 + bx + c$ so abändern, daß nur noch Additionen von Konstanten auftreten.

$$y_i = ax_i^2 + bx_i + c$$

$$y_{i+1} = ax_{i+1}^2 + bx_{i+1} + c = a(x_i + \delta)^2 + b(x_i + \delta) + c$$

$$= ax_i^2 + 2a\delta x_i + a\delta^2 + bx_i + b\delta + c$$

$$= ax_i^2 + bx_i + c + 2a\delta x_i + a\delta^2 + b\delta$$

$$= y_i + z_i + a\delta^2 + b\delta$$

$$z_i = 2a\delta x_i$$

$$z_{i+1} = 2a\delta(x_i + \delta)$$

$$= 2a\delta x_i + 2a\delta^2$$

$$= z_i + 2a\delta^2$$

Wir haben nun das Problem auf zwei Gleichungen reduziert:

$$y_{i+1} = y_i + z_i + a\delta^2 + b\delta$$

$$z_{i+1} = z_i + 2a\delta^2$$

Um also Funktionswerte berechnen zu können, brauchen wir lediglich einmal den Startwert y_0 nach der »normalen« Gleichung $y = ax^2 + bx + c$ zu berechnen und den Startwert z_0 der »Hilfsfunktion« nach $z_0 = 2a\delta x_0$. Ferner brauchen ebenfalls nur einmal die Konstanten $a\delta^2$, $b\delta$ und $2a\delta^2$ berechnet zu werden. Alle weiteren Funktionswerte sind dann mit den beiden Gleichungen nur über Additionen berechenbar.

Nun werden Sie fragen, was es bringt, drei Multiplikationen und zwei Additionen ($a \cdot x \cdot x + b \cdot x + c$) durch vier Additionen zu ersetzen. Einfache Antwort: riesige Zeitersparnis! Zum Beweis finden Sie auf der beiligenden CD-ROM das Programm *function*, das Funktionswerte nach der kubischen Funktionsgleichung auf beide Arten berechnet und die Zeit mißt, die für 100.000 Berechnungen notwendig ist. Lassen Sie sich überraschen!

TIP Denken Sie bitte daran, daß solche »Optimierungen« in der Regel zu wesentlich besseren Ergebnissen führen als das Totoptimieren des

vom Compiler erzeugten Codes! Denn auch das gehört zur effektiven Programmierung von heute: erst denken, dann programmieren – und erst dann und nur dort, wo es sinnvoll ist, optimieren!

HINWEIS

Falls Sie einen Coprozessor besitzen oder die Emulation, die heutzutage von Hochsprachen zur Verfügung gestellt wird, nutzen wollen, bedenken Sie bitte folgendes. Die Hardware (in Form des Coprozessors) ist daraufhin optimiert, mit Realzahlen möglichst effektiv zu arbeiten! Das bedeutet, daß eine Multiplikation des Coprozessors in der Regel nicht langsamer ist als eine Addition (manchmal sogar schneller – siehe hierzu die Taktangaben im Teil 3 des Buches). »Optimierung« der Berechnung von Funktionswerten mit Coprozessor heißt in diesem Fall, daß die Berechnungen von kubischen Funktionswerten beispielsweise ins Gegenteil umschlägt: statt deutlich schneller, geht's nun drastisch langsamer. Denn schließlich müssen in diesem Fall nach der obigen Methode 14 Additionen durchgeführt werden, denen im »klassischen« Fall 6 Multiplikationen und 3 Additionen gegenüberstehen, also lediglich ca. 64% der (etwa gleich schnellen) Operationen!

Im Falle der Emulation dagegen müssen die Coprozessorfunktionen mit den gleichen Prozessorbefehlen nachgebildet werden, die die Integer-Berechnungen oben benutzen.

30.2 Optimieren mit dem Assembler

Ein gutes Beispiel für das Optimieren mit Assembler ist da dass *Pattern-Matching*. Unter diesem Begriff, den man ganz wörtlich mit *Musteranpassung* übersetzen kann, versteht man das Durchforsten einer Struktur nach einem bestimmten Muster. Konkret fällt hierunter z.B. das Durchsuchen eines Textes nach einer vorgegebenen Zeichenfolge.

Im allgemeinen Fall lassen sich durch *Pattern-Matching* beliebige, zum Teil sehr komplexe Muster definieren, mit denen die Struktur abgeglichen werden soll. Hierbei können dann z.B. *Wildcards* verwendet werden, wie Sie sie mit den Zeichen *?* und * aus DOS bereits kennen. Komplexe *Pattern-Matching*-Algorithmen beherrschen Syntax und Semantik wie Hochsprachen und sind entsprechend flexibel einzusetzen. Ich möchte mich hier jedoch auf den einfachsten Fall beschränken: auf die Suche eines konkret vorgegebenen Musters in einer Struktur:

Lösung 1

```
function Search1(S,P:pByteArray; SSize,PSize:word):word;
  var i,
      j  : word;
  begin
    Search := 0;
```

```
  if (SSize = 0) or (PSize = 0) or (SSize < PSize)
    then exit;
  i := 0;

  repeat
    j := 0;
    repeat
       inc(i);
    until (i > SSize - PSize + 1) or (S^[i-1] = P^[j]);
    if (i <= SSize - PSize + 1) then
    repeat
       inc(j)
    until (j > PSize - 1) or (S^[i+j-1] <> P^[j]);
  until (j > PSize - 1) or (i > SSize - PSize + 1);
  if i <= SSize - PSize + 1 then Search := i;
end;
```

Die Funktion erhält vier Parameter: je eine Adresse, an der der zu durchsuchende Text und die zu suchende Zeichenfolge stehen, sowie die Größe der beiden Strukturen. Wir realisieren die Funktion über Zeiger, damit wir unabhängig von der Art der Daten sind. Auf diese Weise können wir nicht nur Strings nach Zeichen absuchen, sondern z.B. auch in Byte-Feldern bestimmte Byte-Sequenzen auffinden.

Bei dieser Strategie wird zunächst in einem REPEAT-Block ein Index so lange inkrementiert, bis das erste Zeichen von »Muster« und »Text« übereinstimmen. Führt dies zu einem Erfolg, so wird in einer zweiten REPEAT-Schleife ein weiterer Index so lange erhöht, bis entweder alle Zeichen in »Muster« und »Text« ab Position Index1 übereinstimmen oder eben nicht. Eingebettet sind beide Schleifen in eine weitere, die abbricht, wenn das Muster gefunden wurde oder es nichts mehr zu vergleichen gibt.

So etwa würde wohl die Lösung des Problems aussehen, die am einleuchtendsten ist. Nun zum Thema »Optimieren«. Unser frustierter Freund von eben würde nun in Ermangelung eines »optimierten« Compilers ärgerlich »von Hand« das Erzeugnis seines Compilers aus diesem Quelltext auf »überflüssige« Befehlsfolgen untersuchen. Mit Sicherheit wird er auch einige Stellen finden, an der es »nachzufeilen« gilt. Aber das sind, wie schon erwähnt, *Peanuts*. Denn was nun kommt, ist so überwältigend, daß es die Nachbearbeitung jedes Hochsprachen-Compilats von Hand ad absurdum führt.

Da wir nun die Assemblerbefehle kennen, kennen wir auch zwei äußerst potente Befehle, die wir für unsere Zwecke verwenden können, die aber aus welchen Gründen auch immer in Pascal kein Hochsprachenpendant haben: CMPS und SCAS. SCAS kann sehr gut dafür ein-

gesetzt werden, das erste Zeichen aus dem Muster im Text zu finden, während man mit CMPS dann beide Zeichenfolgen vergleichen kann, wenn SCAS fündig geworden ist.

So klug waren die Entwickler von Turbo Pascal auch. Das eben geschilderte Problem ist so häufiger Natur, daß Borland seinem Pascal-Compiler die Funktion *Pos* mitgegeben hat, die ähnliches leistet und sich die gleichen Assemblerbefehle zunutze macht. Dennoch habe ich die Funktion *Pos* für unsere Zwecke erweitert! Das Original läßt nämlich »nur« Strings als Text und Muster zu. Die Informationen über die Länge der beiden Strings werden aus ihrem Längenbyte selbst ermittelt, so daß die Funktion eben auch nur zwei Parameter hat. Um jedoch flexibel bleiben und auch längere Strukturen durchsuchen zu können, realisiere ich daher folgende Mutante von *Pos*:

Lösung 2

```
function Search2(S,P:pByteArray; SSize,PSize:word):word;
assembler;
  asm
      push  ds
      lds   si,P
      cld
      mov   dx,PSize
      or    dx,dx
      jz    @No
      les   di,S
      mov   cx,SSize
      inc   cx
      sub   cx,dx
      jb    @No
```

LDS und LES werden benutzt, um die Registerkombinationen DS:SI und ES:DI mit der Adresse des Textes und des Musters zu laden. In DX wird die Länge des Musters, in CX die des Textes abgelegt. Diese wird dann anschließend um die Länge des Musters gekürzt, da das Muster ja vollständig im Text enthalten sein muß, die letzte mögliche Position für das erste Zeichen aus dem Muster also DX – CX + 1 sein kann. Bleibt noch die Prüfung, ob das Muster und der so »gekürzte« Text überhaupt noch eine definierte Länge haben, weil man sich ansonsten die Routine sparen kann.

```
@Loop:lodsb
      repnz scasb
      jnz   @No
```

Diese Sequenz ist praktisch die erste der »inneren« REPEAT-Schleifen von oben. `LODSB` lädt aus dem Muster das erste Zeichen in AL. `REPNZ SCASB` sucht nun so lange im Text nach dem in AL stehenden Zeichen, bis ent-

weder CX 0 ist, was bedeutet, daß die letzte mögliche Position für dieses Zeichen keine Übereinstimmung erbracht hat, oder es wird davor eine Position im Text gefunden, an der dieses Zeichen steht. Entscheidend ist hierbei die Stellung des Zero-Flags, da dieses ja von SCAS gesetzt wird, wenn das Zeichen gefunden wurde. So kann *Search2* mit negativem Ergebnis beendet werden, wenn das Zero-Flag gelöscht ist.

```
mov   ax,di
mov   bx,cx
mov   cx,dx
dec   cx
repz  cmpsb
jz    @Yes
```

Andernfalls sind wir fündig geworden und können die zweite REPEAT-Sequenz beginnen. Dazu kopieren wir zunächst die aktuelle Position in Text aus DI in AX sowie die noch abzuarbeitende Anzahl von Zeichen in TEXT aus CX in BX. Der Grund dafür ist, daß der nun folgende Vergleich von Muster und Text ja negativ ausgehen könnte. In diesem Fall muß mit der Suche nach »dem ersten Zeichen« an dieser Stelle weitergemacht werden.

Wie viele Zeichen müssen nun verglichen werden? So viele, wie das Muster hat, abzüglich des ersten schon gefundenen. Also wird die Musterlänge aus DX in das *Counter-Register* kopiert und um eins dekrementiert. Dann kann mit *REPZ CMPSB* praktisch die zweite REPEAT-Schleife ausgeführt werden, die entweder zu einer Übereinstimmung führt, was wiederum durch das Zero-Flag signalisiert wird, oder eben nicht – dann geht es weiter.

Bitte bemerken Sie noch folgendes: Während in der ersten REPEAT-Anweisung mittels *REPNZ SCASB* so lange repetiert wurde, wie keine Übereinstimmung herrschte, wird hier mittels *REPZ CMPS* so lange wiederholt, wie Übereinstimmung herrscht!

```
mov   di,ax
mov   cx,bx
mov   si,P
jmp   @Loop
```

Wie in der Pascal-Realisation sind die beiden »REPEAT«-Sequenzen auch hier in eine Schleife eingebettet. Falls nämlich der Vergleich eben scheiterte, muß mit *REPNZ SCASB* weitergesucht werden! Dazu holen wir uns die zuvor gesicherten Positionen im Text wieder in die korrekten Register zurück und springen an den Anfang der Suche zurück.

```
@No:  xor   ax,ax
      jmp   @Exit
```

```
@Yes: sub   ax,S
@Exit:pop   ds
  end;
```

Fehlt nur noch, das Funktionsergebnis zu bilden. Das ist einfach, wenn das Muster nicht gefunden wurde. Dann gibt *Search2* wie das Original *Pos* und *Search1* auch 0 zurück. Dies wird am Label *@No* vorbereitet, indem AX, über das in Pascal Funktionsergebnisse in Wortgröße zurückgegeben werden, gelöscht wird. Andernfalls haben wir in AX noch die Position stehen, an der das erste Zeichen gefunden wurde. Diese Position ist aber keine relative Position, also eine »Entfernung« vom Anfang von Text, sondern der Offset der absoluten Adresse! Wenn wir von diesem Offset den Offset des Anfangs von Text abziehen, erhalten wir die Position in Text! DS wird restauriert – und fertig.

Hier der Beweis, daß *Search2* funktioniert, und ein Zeitvergleich zwischen *Search1* und *Search2*. Baut man nämlich ein Programm, in dem beide Funktionen nacheinander aufgerufen werden und das die Zeit mißt, die beide Routinen benötigen, so ergeben sich interessante Aspekte. Realisiert wurde ein solches Programm in Turbo Pascal. Es liegt in der Datei SEARCH.PAS als Quelltext und in SERACH.EXE für alle, die kein Turbo Pascal haben, kompiliert vor. SEARCH sucht in einem »Text« aus 64.000 Zufallszeichen, dem am Ende der Suchstring angehängt wurde, genau diesen Suchstring. Alle Suchroutinen in SEARCH finden daher den Suchstring, allerdings erst ganz am Ende!

Ruft man SEARCH also mit einem Parameter auf, der angibt, wie oft die zu untersuchenden Routinen ausgeführt werden sollen, so ergibt sich auf einem 80386 mit 20 MHz für 100 Ausführungen von *Search1* eine Rechenzeit von 25,33 Sekunden, also ca. 250 msec pro Durchlauf. *Search2* dagegen benötigt 3,08 Sekunden — aber nicht pro Durchlauf! Für 100 Durchläufe! Also ca. 30 msec pro Aufruf. *Das sind 12,16% der Zeit, die Search1 für die gleiche Aufgabe benötigt hat.* Anders ausgedrückt: die Turbo Pascal-Routine benötigt *822%* der Rechenzeit der Assemblerroutine. Ist das nichts?

Doch dies ist zunächst nur die Assembleroptimierung einer noch ganz und gar nicht optimalen Problemlösung! Mathematiker haben auch das Pattern-Matching eingehend untersucht, und sie sind auf eine ganz plausible Lösung gekommen. Der Hintergund ist eigentlich auch ganz einfach. Wenn nämlich beim Vergleich von Muster und Text im Text ein Zeichen gefunden wird, das im Muster gar nicht vorkommt, so können wir mit der Suche hinter diesem Zeichen weitermachen, ohne weiter vergleichen zu müssen.

Vergleicht man daher Muster und Text nicht, wie oben gezeigt und eigentlich auch logisch, »von vorn nach hinten«, also vom ersten Zei-

chen des Musters bis zum letzten, sondern »von hinten nach vorn«, also beginnend mit dem letzten Zeichen von Muster, so kann man sich *musterlänge*mal den Vergleich sparen, wenn ein im Muster nicht existierendes Zeichen gefunden wird. Ein Beispiel:

```
D i e s   i s t   d e r   T e x t

M u s t e r
        M u s t e r
              M u s t e r
                    M
```

Wenn wir das Muster mit dem Text vergleichen, so stellen wir fest, daß an der letzten Position von Muster (am Zeichen *r*) keine Übereinstimmung mit dem Text (Zeichen *i*) herrscht. Mehr noch: da das Zeichen *i* in Muster überhaupt nicht vorkommt, können wir das Muster für den nächsten Vergleich bis hinter das *i* verschieben! Also auf einen Schlag um sechs Positionen. Der erneute Vergleich von Muster und Text zeigt dann zunächst ein »passendes« *r* und dann ein *e*, bis sich am *t* von Muster wieder Unterschiede ergeben. Auch hier stellen wir fest, daß das *d* im Text im Muster nicht vorkommt, so daß wir wieder verschieben können – bis hinter das *d*! Und so weiter.

Lösung 3

```
function Search3(S,P:pByteArray; SSize,PSize:word):word;
  const TabSize = 255;
  var   i,j     : word;
        Table   : Array[0..TabSize] of word;
  begin
    Search3 := 0;
    if (SSize = 0) or (PSize = 0) or (Size < PSize)
      then exit;
    for i := 0 to TabSize do Table[i] := PSize;
    for i := 1 to PSize do Table[P^[i - 1]] := PSize - i;
    i := PSize;
    j := PSize;
    repeat
      if S^[i - 1] = P^[j - 1]
         then begin
           dec(i);
           dec(j);
         end
      else begin
         if PSize - j + 1 > Table[S^[i - 1]]
           then i := i + PSize - j + 1
           else i := i + Table[S^[i - 1]];
```

```
        j := PSize;
      end;
    until (j < 1) or (i > SSize);
    if i <= SSize then Search3 := i + 1;
  end;
```

Bemerkt sei noch, daß `Table` eine Tabelle ist, in der für jedes der 256 möglichen Bytes eingetragen wurde, um wie viele Positionen das Muster »weitergeschoben« werden kann. Kommt das betreffende Zeichen im Muster nicht vor, so ist es immer die Länge des Musters, weshalb mit der ersten FOR-Sequenz dieser Wert in die Tabelle eingetragen wird. In einem zweiten Schritt erfolgt dann der Eintrag dieses »Versatzes« für Zeichen, die im Muster vorkommen. Der Rest ist dann der Suchalgorithmus.

Wenn wir SEARCH nun auch auf *Search3* anwenden, so messen wir 2,11 Sekunden Ausführungszeit! Dies ist erstaunlich! Der realisierte Algorithmus ist selbst in der Pascal-Version um 30% schneller als die Assemblerversion der »einleuchtenden« Routine! Dies schreit nach einer Assembleroptimierung:

Lösung 4

```
function Search4(S,P:pByteArray; SSize,PSize:word):word;
assembler;
  type TableArray = Array[0..255] of word;
  var  Table : TableArray;
       TabOfs: word;
       EndOfs: word;
    asm
       mov   TabOfs,SP
       add   TabOfs,4
       push  ds
       mov   ax,SSize
       or    ax,ax
       jz    @Not
       sub   ax,PSize
       JB    @Not
       cmp   PSize,0
       jz    @Not
       PUSH  SS
       pop   es
       mov   di,TabOfs
       push  di
       mov   cx,00100h;
       mov   ax,PatternSize
       cld
       rep   stosw
```

Dieser Teil entspricht der Vorbelegung der Tabelle, die hier wie im Original lokal eingerichtet wurde, mit der Länge des Musters. Vorher wurde noch geprüft, ob Text- und Musterlänge ein Abarbeiten der Routine überhaupt erlauben.

```
       pop   di
       lds   si,Pattern
       push  si
       mov   cx,ax
       mov   dx,ax
@Loop1:lodsb
       mov   bl,al
       xor   bh,bh
       shl   bx,1
       dec   cx
       mov   es:[di+bx],cx
       inc   cx
       loop  @Loop1
```

Es folgt der Eintrag in die Tabelle für die im Muster vorhandenen Zeichen. Hierzu wird das Muster in der Schleife zeichenweise ausgelesen und der Abstand vom »Ende« des Musters an die dem Zeichen entsprechende Stelle der Tabelle eingetragen.

```
       pop   si
       add   si,dx
       dec   si
       les   di,s
       add   di,dx
       dec   di
       add   ax,SSize
       mov   EndOfs,ax
       mov   ax,si
       std
@Loop2:mov   cx,dx
       repz  cmpsb
       jz    @Found
```

Nun wird mittels des CMPS-Befehls das Muster mit dem Text verglichen. Beachten Sie bitte, daß das Direction-Flag mit STD gesetzt wurde, weshalb der Vergleich »rückwärts« erfolgt. Aus diesem Grunde wurde auch zu den in ES:DI und DS:SI stehenden Adressen jeweils die Musterlänge addiert.

```
       inc   di
       mov   bl,es:[di]
       xor   bh,bh
```

```
       shl   bx,1
       add   bx,[TabOfs]
       mov   cx,ax
       sub   cx,si
       cmp   cx,ss:[bx]
       ja    @Next
       mov   cx,ss:[bx]
@Next: add   di,cx
       mov   si,ax
```

Falls das Muster nicht an der aktuellen Position gefunden wurde, wird mit Hilfe der Tabelleneinträge die neue Vergleichsposition berechnet. Dieser Schritt entspricht der »Verschiebung« des Musters um die Anzahl an Bytes nach rechts, die in der Tabelle verzeichnet sind.

```
cmp   di,EndOfs
ja    @Not
jmp   @Loop2
```

Falls damit das Ende des Textes noch nicht erreicht ist, erfolgt ein neuer Vergleich des Textes mit dem Muster an der neuen Position.

```
@Not:  xor   ax,ax
       jmp   @End
@Found:mov   ax,di
       sub   ax,s
       inc   ax
       inc   ax
@End:  pop   ds
  end;
```

Führt man auch für diese Routine eine Zeitmessung durch, so konnte durch die Assembleroptimierung nochmals eine Reduktion der Ausführungszeit um ca. 60% erreicht werden! Lösung 4 benötigt selbst auf einem 20-MHz-80386 nur 0,86 Sekunden für 100 Durchläufe der Mustersuche.

Ergebnis

Durch die sinnvolle Optimierung von Algorithmus und den Einsatz von Assembler konnte eine *Reduktion* des Zeitbedarfs *um 96,6%* erreicht werden. Das bedeutet, daß die »einsichtigste« Lösung des Problems in der Hochsprache *2945%* der Zeit der »optimierten« Realisierung benötigt. Doch selbst die algorithmusoptimierte Lösung in der Hochsprache benötigt noch ca. das 2½-fache – und das bei einer Routine, die z.B. bei der Durchmusterung von Texten in Textverarbeitungssystemen eingesetzt wird.

HALT!

Doch auch das gehört zur Optimierung: Ehrlichkeit sich selbst gegenüber und »testen, testen, testen ...«! Denn ein paar Probeläufe des Programms SEARCH mit variierenden Längen des Musters bringen et-

was ans Tageslicht, was man sich eigentlich hätte denken können und müssen und was berücksichtigt werden muß. Gemessen wurden jeweils 100 Suchdurchläufe auf einem 80386 mit 20 MHz, wobei darauf geachtet wurde, daß das Muster jeweils an der letztmöglichen Position im Text stand. Dargestellt sind die gemessenen Zeiten:

Muster	Routine:	1	2	3	4
«D«		25,27	3,08	**59,67**	**25,00**
«Di«		25,22	3,13	**30,13**	**12,74**
«Die«		25,22	3,08	**20,36**	**8,59**
«Dies«		25,27	3,13	**15,28**	**6,59**
«Dies »		25,22	3,08	**12,26**	**5,29**
«Dies i«		25,27	3,13	**10,21**	**4,43**
«Dies is«		25,27	3,13	**8,27**	**3,78**
«Dies ist«		25,27	3,19	**7,61**	3,35
«Dies ist »		25,22	3,08	**6,86**	3,02

Die Routinen 3 und 4 arbeiten mit Tabellen, die erst einmal eingerichtet werden müssen, wozu das Durchsuchen des Musters gehört, noch bevor die Suche überhaupt beginnt. Diese Vorbereitung muß ja auf Kosten der Suchzeit gehen – je länger das Muster, desto länger die Vorbereitung. Für die Suche nach *einem* Zeichen im Text hat dies nicht nur keinen Vorteil, sondern sogar Nachteile. In diesem Fall muß nämlich während der Suche die zu verschiebende Anzahl von Zeichen aus der Tabelle entnommen und zur aktuellen Position addiert werden, obwohl jeder instinktiv weiß, daß dies trivialerweise 1 ist. Dementsprechend deprimierend sieht das Resultat für unsere superoptimierte Routine aus, wenn lediglich nach einem Zeichen gesucht wird: fast eine halbe Minute – genau so viel, wie der erste Lösungsansatz benötigt!

Da weder Routine 1 noch ihr Assembleräquivalent diesen »Wasserkopf« besitzen, ist bei ihnen die Ausführungszeit im Rahmen der Meßgenauigkeit konstant, während sich die Algorithmusoptimierung mit zunehmender Musterlänge zunehmend positiv auswirkt. Ab etwa neun Zeichen Musterlänge erfolgt dann der »Durchbruch«: Ab hier »lohnt« sich diese Routine dann tatsächlich – sie ist nun schneller geworden als die assembleroptimierte »Triviallösung«. Konsequenz: Ein allgemein anwendbarer Suchalgorithmus sollte zunächst die Musterlänge prüfen und sich dann entsprechend einem zu bestimmenden Schwellenwert (hier: 9) für die eine oder die andere Routine entscheiden. Auf diese Weise kann sichergestellt werden, daß für alle Fälle der »optimale« Algorithmus verwendet wird. Realisiert wurde dies im Modul SEARCH.ASM. Die dort realisierte Routine können Sie in eigene Programme einbinden. Wie, zeigt Ihnen SEARCH2.PAS.

30.3 Noch ein Beispiel

Es müssen nicht immer so drastische Fälle sein, die die Assemblereinbindung sinnvoll machen. Ein anderes Beispiel findet sich in der Nutzung von BIOS- oder DOS-Funktionen.

Viele Hochsprachen stellen dazu Routinen zur Verfügung, so z.B. auch Turbo Pascal. Doch schauen wir uns einmal an, was passiert, wenn man tatsächlich eine BIOS-Funktion aus dieser Hochsprache heraus ausführt. Dazu schauen wir uns zunächst einmal ein Turbo Pascal-Minimalprogramm an, das lediglich den Interrupt $10 aufrufen soll, wobei in AX der Wert 0 übergeben wird:

```
uses dos;

var r:registers;

begin
  r.ax := $00;
  Intr($10,r);
end.
```

Das Programm bindet die Unit DOS ein, da dort die Prozedur *Intr* definiert ist und auch der Typ *Registers* deklariert wird. Das Hauptprogramm selbst definiert nur eine Variable *R* von diesem Typ und belegt das Feld AX des Records *R* mit 0. Dann wird mit *Intr* die Interrupt-Routine aufgerufen, wodurch die BIOS-Funktion $10 angesprungen wird. Was macht nun der Compiler daraus?

```
cs : 0000 9A0000620C    CALL    0C62:0000
     0005 55            PUSH    BP
     0006 89E5          MOV     BP,SP
     0008 31C0          XOR     AX,AX
     000A A35000        MOV     [0050],AX
     000D B010          MOV     AL,10
     000F 50            PUSH    AX
     0010 BF5000        MOV     DI,0050
     0013 1E            PUSH    DS
     0014 57            PUSH    DI
     0015 9A0B005B0C    CALL    0C5B:000B
     001A 5D            POP     BP
     001B 31C0          XOR     AX,AX
     001D 9A1601620C    CALL    0C62:0116
```

Als erstes wird eine hier nicht weiter interessierende Systemroutine aufgerufen. Dann erzeugt der Compiler einen Stackrahmen für das Hauptprogramm. Nachdem eine Variable im Datensegment (an Adresse $0050),

bei der es sich offensichtlich um unsere Variable *R* handelt, mit 0 vorbelegt wird, werden der Wert $10 und anschließend die Adresse von *R* auf den Stack gelegt. Schließlich wird eine Routine aufgerufen, die sich (hier!) an Adresse $0C5B:000B befindet. Nach der Rückkehr aus dieser Routine wird der Stackrahmen wieder entfernt, und über die Systemroutine an Adresse $0C62:0116 wird das Programm mit dem Fehlercode 0 in AX beendet.

Wo ist unser Interrupt-Aufruf, mit dem BIOS-Routinen ja genutzt werden? Antwort: im Hauptprogramm nicht! Sondern, wie wir gleich sehen werden, in der aufgerufenen Routine. Und die sieht so aus:

```
0C5B:000B 55            PUSH    BP
     000C 1E            PUSH    DS
     000D 8BEC          MOV     BP,SP
     000F 2E            CS:
     0010 FF363A00      PUSH    [003A]
     0014 8A460C        MOV     AL,[BP+0C]
     0017 2E            CS:
     0018 A23B00        MOV     [003B],AL
     001B C57608        LDS     SI,[BP+08]
     001E FC            CLD
     001F AD            LODSW
     0020 50            PUSH    AX
     0021 AD            LODSW
     0022 8BD8          MOV     BX,AX
     0024 AD            LODSW
     0025 8BC8          MOV     CX,AX
     0027 AD            LODSW
     0028 8BD0          MOV     DX,AX
     002A AD            LODSW
     002B 8BE8          MOV     BP,AX
     002D AD            LODSW
     002E 50            PUSH    AX
     002F AD            LODSW
     0030 8BF8          MOV     DI,AX
     0032 AD            LODSW
     0033 50            PUSH    AX
     0034 AD            LODSW
     0035 8EC0          MOV     ES,AX
     0037 1F            POP     DS
     0038 5E            POP     SI
     0039 58            POP     AX
     003A CD00          INT     00
     003C 9C            PUSHF
     003D 06            PUSH    ES
```

```
003E 57          PUSH    DI
003F 55          PUSH    BP
0040 8BEC        MOV     BP,SP
0042 C47E12      LES     DI,[BP+12]
0045 FC          CLD
0046 AB          STOSW
0047 8BC3        MOV     AX,BX
0049 AB          STOSW
004A 8BC1        MOV     AX,CX
004C AB          STOSW
004D 8BC2        MOV     AX,DX
004F AB          STOSW
0050 58          POP     AX
0051 AB          STOSW
0052 8BC6        MOV     AX,SI
0054 AB          STOSW
0055 58          POP     AX
0056 AB          STOSW
0057 8CD8        MOV     AX,DS
0059 AB          STOSW
005A 58          POP     AX
005B AB          STOSW
005C 58          POP     AX
005D AB          STOSW
005E 2E          CS:
005F 8F063A00    POP     [003A]
0063 1F          POP     DS
0064 5D          POP     BP
0065 CA0600      RETF    0006
```

Diese Routine macht nichts anderes als

```
int      010h
```

Der Compiler kann nicht vorhersehen, wann wer welchen Interrupt auslösen will und welche Register hierbei betroffen sind! Daher muß eine Möglichkeit gefunden werden, möglichst flexibel jede Eventualität abzudecken. Nun wissen wir aber inzwischen, daß Flexibilität nur mit Aufwand zu erreichen ist – hier eben mit aufwendiger Programmmierung einer allgemein nutzbaren INT-Routine.

Diese richtet zunächst einen Stackrahmen ein. Dann passiert etwas Seltsames, auf das wir sofort zurückkommen.

Als nächstes wird die Adresse der übergebenen Variablen *R* geladen. Mittels der folgenden LODSW-Befehle werden nun die Prozessorregister mit dem Inhalt der korrespondierenden Felder in *R* besetzt. Es

mag zwar etwas umständlich erscheinen, wie dies erfolgt, jedoch führt eine genauere Betrachtung zum Ergebnis, daß es anders nicht möglich ist, alle Register des Prozessors, also auch die, die der LODSW-Befehl benutzt (DS und SI), mit den Inhalten aus *R* zu beladen. Schließlich kann die Routine ja nicht vorhersehen, ob DS oder DI tatsächlich benötigt werden.

Nun folgt Erstaunliches: *int 00*! Eigentlich wollen wir Interrupt $10 aufrufen, und INT $00 löst die gleiche Aktion aus, die eine Division durch 0 erzeugen würde.

Es hat jedoch alles seine Richtigkeit, denn diese Zeile wird modifiziert. Und zwar von den Befehlen, die wir eben von der Betrachtung ausgeschlossen haben: den Zeilen mit den Offsets $000F bis $0018! Diese schieben nämlich zunächst den Inhalt an der Stelle CS:003A auf den Stack. An dieser Stelle aber steht *int 00*. Dann wird an die Stelle CS:003B der der Routine als Parameter übergebene Wert eingetragen. In unserem Falle also $10, was zu dem Befehl *int 10* führt.

ACHTUNG Es handelt sich hier um selbstmodifizierenden Code. Damit bezeichnet man Programme, die sich während der Laufzeit selbst modifizieren. Das ist äußerst gefährlich und sollte nur von denjenigen nachgeahmt werden, die sich über alle Folgen im klaren sind. Ich möchte an dieser Stelle ausdrücklich davor warnen, solche Tricks anzuwenden, wenn Sie nicht über ein gehöriges Maß an Kenntnissen verfügen!

Der Hintergrund, warum man zu solchen Tricks greifen muß, ist klar. Intel hat nämlich keine Möglichkeit vorgesehen, dem INT-Befehl den Operanden indirekt, also über eine Variable zu übergeben.

Nachdem die aufgerufene INT-Routine beendet wurde, passiert das ganze rückwärts: Die Inhalte der Prozessorregister werden in die korrespondierenden Felder der Variablen kopiert, und alle veränderten Dinge (in diesem Fall die nicht zur Nachahmung empfohlene Modifizierung des Codes) werden restauriert.

HINWEIS Gewöhnen Sie sich nicht daran, selbstmodifizierenden Code zu benutzen. Denn dieser funktioniert nur im Real-Mode des Prozessors reibungslos, also wenn Sie DOS-Programme entwickeln. Falls Sie dagegen Windows-Programme schreiben, in die Sie Assemblerteile einbauen wollen, so werden Sie höchstwahrscheinlich Ärger in Form von Fehlermeldungen mit und ohne Abstürzen bekommen. Denn im Protected-Mode, in dem Windows in der Regel arbeitet, sind Code-Segmente schreibgeschützt, eben *protected*. Zwar kann man auch dies umgehen, wie z.B. Borland beweist. Hierzu sind aber sehr genaue Kenntnisse der verschiedenen Mechanismen im Protected-Mode notwendig. Daher höre ich hier mit der Erklärung auf und verweise wieder einmal auf weiterführende Literatur!

Sie sehen also, welche Tricks man anwenden muß, um flexiblen Code zu erzeugen. Doch in den meisten Fällen brauchen wir eigentlich diese Flexibilität gar nicht, da wir etwa die durch die INT-Routine veränderten Register gar nicht benötigen, z.B. wenn man nur die Cursorposition verändern möchte (wozu es allerdings auch Pascal-Routinen gibt). Das führt uns dazu, das Programm von oben wie folgt zu ändern, wobei der integrierte Assembler zum Einsatz kommt:

```
begin
  asm
     mov  ax,000h
     int  010h
  end;
end.
```

Belohnt werden wir durch den erzeugten Code, der hier sehr kurz ausfällt:

```
CS : 0000 9A0000D10D    CALL    0DD1:0000
     0005 55            PUSH    BP
     0006 89E5          MOV     BP,SP
     0008 B80000        MOV     AX,0000
     000B CD10          INT     10
     000D 5D            POP     BP
     000E 31C0          XOR     AX,AX
     0010 9A1601D10D    CALL    0DD1:0116
```

30.4 Spezielle Optimierungen

Lange Zeit, und auch heute noch von einigen Hardlinern, wurde Pascal im mathematisch/wissenschaftlichen Bereich verteufelt, weil es so wenig mathematische Funktionen standardmäßig implementiert hat. Diese Klientel bevorzugt(e) aus solchen Gründen Fortran, eine Sprache, die extra für diesen Zweck erschaffen worden war (»*for*mula *tran*slator«). Wenn wir dies in bescheidenem Maße ändern wollen, so wenden wir uns nun den periodischen Funktionen zu. Pascal kennt nämlich außer *Sin*, *Cos* und *ATan* keine weiteren dieser elementaren Funktionen.

Wir wissen ja, daß der Assembler über einen Befehl verfügt, der etwas mit periodischen Funktionen zu tun hat: FPTAN. Dieser Befehl erzeugt aus einem Argument den sogenannten partiellen *Tangens*.

Was ist der partielle Tangens? Es ist ein Zwischenwert bei der Berechnung des eigentlichen Tangens. Dieser ist definiert als Quotient aus Gegenkathete und Ankathete. FPTAN macht nun nichts anderes, als das Argument, das einen Winkel beschreibt, in zwei Werte zu

zerlegen: in die zum Winkel gehörige Gegen- und Ankathete. Das aber heißt, daß durch eine einfache Division der beiden Ergebniswerte des FPTAN-Befehls der Tangens ermittelt werden kann. Mehr noch: Durch eine reziproke Division (genauer: Division der beiden Werte in umgekehrter Reihenfolge, denn eine reziproke Division ist ja eigentlich eine Multiplikation) kann der Cotangens berechnet werden.

Doch wie erhalten wir Sinus und Cosinus? Auch hier hilft die Mathematik! Seit Pythagoras nämlich gilt in rechtwinkligen Dreiecken, daß das Quadrat der Hypothenuse gleich der Summe der Quadrate über An- und Gegenkathete ist. An- und Gegenkathete kennen wir aus FPTAN – also ist es kein Problem, hieraus die Hypothenuse zu berechnen. Und der Sinus ist ja definiert als Gegenkathete dividiert durch Hypothenuse, Cosinus entsprechend.

Intuition

Wenn wir nun planen, Funktionen zu programmieren, die diese periodischen Berechnungen bewerkstelligen können, so können wir es ganz intuitiv beginnen: Für jede der geplanten Funktionen gibt es eine eigene Routine: *Sin*, *Cos*, *Tan*, *Cot* und auch *Secans* (*Sec* = 1/Cos) und *Cosecans* (*Csc* = 1/Sin). Jede dieser Funktionen ist eigenständig und vollständig in sich abgeschlossen. Jede dieser Funktionen muß zunächst den FPTAN berechnen. Bis auf *Tan* und *Cot* muß darüber hinaus auch die Hypothenuse errechnet werden. Und aufgrund der Einschränkungen des FPTAN-Befehls, zumindest beim 8087 und 80287, muß das Argument – der Winkel – auch auf einen bestimmten Bereich abgebildet werden!

Optimierung #1

Diese Aufgaben, die für alle periodischen Funktionen ausgeführt werden müssen, können wir einem gemeinsamen Programmteil übertragen! Auf diese Weise benötigen wir eine *Erste Allgemeine Transzendenz*, die wir z.B. *Trans* nennen könnten. Diese Routine wird nun von bestimmten *Laderoutinen* mit den jeweiligen Randbedingungen versorgt. Als *Laderoutinen* dienen die *nach außen* sichtbaren Funktionen *Sin*, *Cos*, *Tan*, *Cot*, *Sec* und *Csc*. Innerhalb von *Trans* wird der Winkel auf den korrekten Bereich abgebildet und FPTAN gebildet. *Trans* berechnet auch in Abhängigkeit von den übergebenen Randbedingungen die Hypothenuse und aus den nun bekannten Seiten des Dreiecks den Wert der gewünschten periodischen Funktion.

Optimierung #2

Der Bereich, den FPTAN bei allen Coprozessortypen abdecken kann, ist $0 \leq \phi < \pi/4$. FPTAN gilt also nur im ersten Oktanten zwischen 0 und 45°. Für alle anderen Winkel müssen wir rechnen... oder wiederum nachdenken. Denn auch hier hilft uns die Mathematik. So ist z.B. der Sin(60°) = Cos(30°) und somit wieder im ersten Oktanten. Solche Beziehungen lassen sich für alle Funktionen in allen Oktanten herstellen, so daß die Einschränkung von FPTAN auf den ersten Oktanten kein ernsthaftes Problem ist.

Nun könnte man auf die Idee kommen, daß *Trans* also in Abhängigkeit vom aktuellen Oktanten selbständig feststellt, ob der *Sin* oder *Cos* verwendet werden muß und ob er positiv oder negativ zu interpretieren ist. Dies bedeutete für jede der realisierbaren Funktionen acht Fallunterscheidungen, also insgesamt 48. Daß dies mit erheblichem Codeumfang verbunden ist, dürfte klar sein. Doch warum muß *Trans* dies tun? Wir wissen doch um die Zusammenhänge. Also können wir doch auch *Trans* »von außen« steuern, nämlich über die Ladefunktionen. So können wir z.B. die Funktion *Sin* anweisen, *Trans* neben dem Winkel auch ein Steuerwort zu übergeben, in dem codiert ist, daß im ersten Oktanten der Sinus, im zweiten der Cosinus des gespiegelten Winkels, im dritten der Cosinus des Winkels und im vierten wieder der Sinus des gespiegelten Winkels gebildet werden soll.

Optimierung #3

Dies führt zu einer einzigen, sehr kompakten Routine, die nun assembleroptimiert werden kann!

Optimierung #4

Die fertigen Funktionen haben aber einen kleinen Zusatz mitbekommen: Sie funktionieren sowohl im Bogenmaß als auch im Gradmaß mit Altgrad und Neugrad, durch einfaches Setzen einer Steuervariablen. Doch dies hat mit Optimierung wenig zu tun.

Für das Kapitel Tips & Tricks habe ich noch eine weitere Optimierung aufgehoben. Diese Art der Optimierung jedoch ist mit großer Vorsicht anzuwenden, denn hier werden bestimmte Konventionen, die bei Hochsprachen eingehalten werden müssen, einfach außer Kraft gesetzt. Da dies gefährlich ist, hebe ich mir diese Art der Optimierung für ein eigenständiges Kapitel auf.

Optimierung #5

Wenn Sie sich die Assemblermodule anschauen, die für die Unit *Mathe* verwendet werden, werden Sie bei sehr vielen der implementierten Routinen solche Optimierungen finden. Ich möchte es Ihnen überlassen, den Quellcode zu analysieren. Lassen Sie sich damit Zeit, vollziehen Sie die einzelnen Schritte nach, und lassen Sie sich dadurch inspirieren, eigene Optimierungsstrategien zu entwickeln. Der Assembler bietet Ihnen im Gegensatz zu Hochsprachen dazu die Möglichkeit!

30.5 Optimierungen bei 32-Bit-Betriebssystemen und -Prozessoren

Die konsequente Nutzung der 32-Bit-Technologie sowohl bei Hardware als auch bei Software führt zum Umdenken in verschiedenen Punkten! So liegt beispielsweise eine 32-Bit-Integer (oder »LongInt« im Pascal-Dialekt) auf Systemen, die mit 16 Bit adressiert werden, als Pärchen zweier 16-Bit-Register vor (z.B. DX:AX). In 32-Bit-Systemen dagegen findet man eine solche Integer in einem einzelnen Register (z.B. EAX).

Der Hintergrund ist klar. In 16-Bit-Systemen ist alles auf Werte hin ausgerichtet, die mit 16 Bit maximal darstellbar sind. Das fängt bei Registern an, geht über die Speicheradressierung und damit verbundene Klimmzüge bis hin zur »Ausrichtung« von Daten im Speicher. Eine 16-Bit-Integer beispielsweise, die an einer ungeraden Adresse im RAM liegt (d.h. deren Adresse nicht ohne Restbildung durch 2 teilbar ist), muß mit zwei hintereinander ablaufenden, byteweisen Speicherzugriffen in das Register geholt oder von dort geschrieben werden (der 8088 läßt grüßen!). Liegt sie dagegen an einer geraden Adresse, ist nur ein einziger Zugriff notwendig. Dies ist auch der Grund dafür, daß moderne Hochsprachencompiler den Code zwar byteweise, Daten jedoch immer wortweise ausrichten (*Alignment*). Auch Sie sollten bei der Assemblerprogrammierung darauf achten, wenn Sie mehr programmieren wollen als ein paar grundsätzliche Beispielprogramme wie in diesem Buch.

Bruch mit alten Zwängen (16 Bit) heißt dann aber auch Bruch mit alten Gewohnheiten (16-Bit-Register)! War es früher richtig, möglichst alles »auf 16 Bit hin auszurichten« (man denke an 16-Bit-Schleifenzähler, auch wenn die maximale Anzahl an Schleifendurchgängen in einem Byte Platz gehabt hätte), so sollte man nun konsequent auf 32 Bit »umsteigen« und auch für solche Schleifen 32-Bit-Register verwenden. Denken Sie daran: Das beste Datum ist das, für das der Prozessor ausgelegt ist: ein Byte für den 8088, ein Wort für die Prozessoren bis zum 80386 und vom 80386er an aufwärts ein Doppelwort.

Ist das nicht ein Widerspruch? Bisher liefen doch auch auf Pentium-Rechnern Windows 3.x und 16-Bit-Systeme trotz der 32-Bit-Register. Nein, es ist kein Widerspruch! Denken Sie immer daran, daß erst mit den 32-Bit-Betriebssystemen die Betriebssysteme vorliegen, die den Prozessoren ab 80386 tatsächlich gerecht werden. Alle anderen Betriebssysteme sind Flickschusterei, die die Abwärtskompatibilität der Hardware und Emulationen von virtuellen Prozessoren (Virtual-8086-Mode) etc. zur möglichst optimalen Realisierung der zweitbesten Lösung nutzen.

Denken Sie also beim Arbeiten in echten 32-Bit-Systemen wie Windows 9x/NT oder OS/2 »32-bittig«, wenn Sie Assemblermodule einbinden. Delphi 2 macht es uns vor: Ab jetzt ist eine einfache Integer dasselbe wie eine LongInt. Völlig falsch ist es, ein 32-Bit-Register als zwei 16-Bit-Register aufzufassen und sich zu freuen, daß Sie nun die doppelte Anzahl von Counter-Registern oder ähnliches haben! Solche in 16-Bit-Betriebssystemen sicherlich absolut angebrachten Optimierungen sind beispielsweise u.a. der Grund dafür, daß der Pentium Pro mit 16-Bit-Programmen und Windows 3.x schlechter abschneidet als sein Vorgänger: Er ist eben vollständig auf 32 Bit hin optimiert und wird nun ausgebremst.

Optimieren unter 32-Bit-Systemen heißt eventuell sogar, verschwenderisch zu sein! Sie sind sicherlich besser bedient, eine Boolesche Variable in einer 32-Bit-Variablen (z.B. LongBool in Delphi) zu halten und damit 31 Bits (statt 7 der »normalen« Bool) zu verschenken, als durch byteweise Zugriffe und Manipulationen teilweise umständliche Befehlsfolgen und gegebenenfalls sogar zusätzliche Befehle in Kauf zu nehmen.

30.6 Generelle Tips zum Optimieren

Hier ein paar generelle Tips, die für Sie beim Erstellen von Assemblercode nützlich sein können. Die folgende Liste erhebt natürlich keinen Anspruch auf Vollständigkeit:

- Die Codeoptimierung kann nicht (nur) auf dem Papier erfolgen! Es gibt keine andere verläßliche Methode, den Erfolg von Optimierungen zu verifizieren, als tatsächlich zu messen (Speichergrößen, Ausführungszeiten etc.). Theoretisieren Sie nicht, praktizieren Sie, prüfen Sie!
- Es gibt nicht »den« optimalen Code! Optimierter Code an der einen Stelle (z.B. einem 16-Bit-Betriebssystem) kann an anderer Stelle (z.B. einem 32-Bit-Betriebssystem) tödlich sein. Achten Sie immer darauf, in welcher und für welche Umgebung Sie Code schreiben.
- Nochmals: Es gibt nicht »den« optimalen Code! Optimierte Befehlsfolgen in einem bestimmten Kontext müssen in einem anderen Kontext noch lange nicht optimal sein. Besonders häufig trifft man vor allem in Schleifen auf Sequenzen, die allein schon dadurch optimiert werden können, daß man taktreiche Befehle möglichst weitgehend aus den Schleifen herauszieht. Manchmal wird dazu das Einfügen von zusätzlichen Befehlen notwendig, was bei oberflächlicher Betrachtung als Rückschritt und Verschwendung erscheinen könnte, in Wirklichkeit aber wesentliche Einsparungen bringen kann.
- Versuchen Sie nicht, alles mit einem Programm zu erschlagen! Programme, die auch auf 8088ern laufen sollen, sind sicherlich nicht für heutige Standards geeignet. Umgekehrt wird Windows 95 mit Sicherheit Probleme machen, wenn es auf 80286ern laufen soll. Haben Sie also den Mut, alten Ballast auch über Bord zu werfen. Erstellen Sie nötigenfalls mehrere Module, die speziell für bestimmte Situationen maßgeschneidert sind.
- Ein 80486 ist kein schneller 80386, ein Pentium ist kein schneller 80486 und, auch wenn sie ähnlich heißen, ist ein Pentium Pro kein Pentium! Jeder dieser Prozessoren hat eigene Wege, Code vorzubereiten und auszuführen – man denke an *Prefetch Queues*, *Pipelines* etc. `JCXZ` ist schneller als `AND CX,CX;JZ` auf Prozessoren bis zum 80286, ungefähr gleich schnell auf 80386ern, aber nur halb so

schnell auf dem 80486, gemessen in Takten (daß das Ganze teilweise durch die höhere Arbeitsgeschwindigkeit modernerer Prozessoren überkompensiert wird, ist eine andere Geschichte, die man durchaus im Kopf behalten sollte!).

- Optimieren Sie sich nicht zu Tode! Der Zeitaufwand, der zur Erstellung »optimierten Codes« notwendig ist, muß in einem klaren Verhältnis zum Nutzen stehen. Ein paar tausend Takte durch Optimierung eingespart zu haben mag zwar toll aussehen und das Selbstbewußtsein ungemein stärken, nützt aber nichts, wenn der Codeteil im Start-up-Code einer Unit oder im Rahmen einer Message-Routine aufgerufen wird, die 99,999% ihrer Existenz damit verbringt, auf den erlösenden Tasten- oder Mausklick des Anwenders zu warten.
- Nochmals: Optimieren Sie sich nicht zu Tode! Sicherlich können Sie noch Lösungen finden, die einen Text schneller nach einem Muster durchforschen können, als es in diesem Buch gezeigt wurde. Prüfen Sie jedoch, bevor Sie an weitere Optimierungen denken, ob beispielsweise eine Reduktion der Ausführungszeit einer Routine um vielleicht weitere 10% (eigentlich für sich betrachtet eine stolze Ersparnis!) auch wirklich »beim Anwender ankommt«. Denn ob der Text nun in 0,5 oder in 0,45 Sekunden durchsucht worden ist, dürfte den wenigsten überhaupt auffallen. Aber Sie sind eventuell Tage und Wochen mit dem Optimieren beschäftigt.
- Taktzahlen sind nicht alles! Durch bloßes Umstellen einzelner (harmlos aussehender) Assemblerbefehle sind gegebenenfalls bessere Gesamtergebnisse erzielbar als durch allzu penibles Suchen nach dem Code mit der kürzesten Gesamttaktzahl.
- Nochmals: Taktzahlen sind nicht alles! Eine Frage ist, wie schnell ein Befehl ausgeführt wird, wenn er ausgeführt werden kann. Dazu muß sein Code aber bereits im Prozessor zur Ausführung korrekt vorbereitet vorliegen, und die eventuell dazugehörigen Daten und/oder Parameter müssen vorverarbeitet sein. Eine andere Frage ist, wie lange die »Vorarbeiten«, das sog. *Instruction Fetching*, dauern, damit dieser Befehl abgearbeitet werden kann. Denn die Taktangaben der Befehle beinhalten nicht, in welcher Zeit was alles erfolgen muß, damit diese Voraussetzungen tatsächlich erfüllt sind. Eng damit verwandt ist das Füllen der *Prefetch Queue*! Alle Konstruktionen, die ein erneutes Füllen der *Prefetch Queue* unnötig werden lassen (z.B. erfolgt dies nach einer Programmverzweigung), können wahre Wunder wirken.
- Nutzen Sie Assemblerbefehle auch auf unkonventionelle Weise! Sie haben bereits gesehen, wie man die BCD-Korrekturbefehle für schöne Dinge mißbrauchen kann. Es gibt noch weitaus mehr Beispiele: »Wozu dienen die Speicheradressierungsarten der 80x86er wie z.B. MOV AL, [BX+SI]?« Michael Abrash wettet, daß Sie ant-

worten: »Zur Berechnung von Speicheradressen!« Nur? Weit gefehlt, wie er zeigt: Bei Prozessoren ab dem 80386 z.B. können alle 32 Register gleichzeitig Basis und Index bei Berechnungen indirekter Adressen zum Speicherzugriff sein! Was bringt uns das? Schauen Sie sich folgende Zeile an:

```
lea   ebx, [ebx+ebx*4]
```

Was tut dieser Befehl? Er lädt in EBX den Inhalt von EBX+4*EBX. Dies aber ist nichts anderes als das Multiplizieren von EBX mit 5! Es ersetzt die Befehlsfolge

```
mov   edx,ebx
shl   ebx,2
add   ebx,edx
```

und spart somit nicht nur Takte, sondern auch die Nutzung eines 32-Bit-Registers! Möglich sind auf diese Weise Multiplikationen mit 2, 3, 4, 5, 8 und 9, da eine skalierte Berechnung der EA mittels LEA nur die Faktoren 2, 4 und 8 zuläßt.

Michael Abrash ist Autor eines ausgezeichneten Buchs zum Thema »Optimieren«, das ich Ihnen dringend ans Herz legen möchte, da es eine Fülle an Informationen bietet. Er zeigt Ihnen an vielen Beispielen, an was Sie alles denken können (und müssen) und was man tun soll. Wenige Stichworte sind:

- Taktfresser (»*cycle eater*«). Es gibt verschiedene: den »8-Bit-Taktfresser«, den »*Prefetch-Queue*-Taktfresser«, »*Waitstates*«, den »*Display-Adapter*-Taktfresser« usw.
- Stackrahmen – wo sind sie hinderlich? Wir haben gesehen, daß es mit Stackrahmen so seine Bewandtnis hat. Wenn möglich, vermeiden Sie sie. Manchmal lassen sie sich aber nicht umgehen.
- Daten- und Code-Alignment – ein wenig beachtetes Problem.
- Pipelines und Caches u.v.a.m.

Zwar gibt es noch vieles mehr, was man berücksichtigen kann. Aber mit diesen Hinweisen und Empfehlungen kommen Sie zunächst schon ein gutes Stück weiter.

31 8087-Emulation

Wir haben uns gleich bei unserem zweiten Assemblerprogramm mit der Thematik des Coprozessors beschäftigt. Doch was machen wir, wenn wir mit Fließkommazahlen rechnen wollen, jedoch keinen Coprozessor vorfinden können? Hochsprachen haben heutzutage darauf eine Antwort: Emulation.

Damit ist man in der Lage, dem Programm vorzugaukeln, daß tatsächlich ein Coprozessor da ist. Dadurch ist gewährleistet, daß jedes Programm die Coprozessorbefehle benutzen kann, ohne sich darum kümmern zu müssen, ob sie auch verstanden werden.

Das aber bedeutet, daß irgend jemand sich darum kümmern muß, daß ggf. bestimmte Unterprogramme aufgerufen werden, wenn ein Coprozessorbefehl aufgerufen wird. Diese Unterprogramme müssen sich absolut genau so verhalten wie die *Microcodes* in den Coprozessorchips. Eine Sammlung solcher Routinen, die dem Befehlssatz eines Coprozessors entsprechen, nennt man 8087-Emulatoren.

Fast jede moderne Programmiersprache verfügt heute über so eine Bibliothek, die der Programmierer nutzen kann. Mehr noch – der Programmierer setzt lediglich ein oder zwei Compilerschalter, die den Compiler veranlassen, den Emulator einzubinden. Alles weitere macht dann das Programm.

ACHTUNG Als Assemblerprogrammierer haben Sie, wenn wir von dem standardmäßig mitgelieferten 8087-Emulator unter Windows einmal absehen, zunächst nur die Emulatoren, die die Hochsprachen zur Verfügung stellen. Also haben Sie auch nur im Rahmen von Hochsprachen darauf Zugriff. Somit ist offensichtlich, daß reine Assemblerprogramme Probleme mit der Emulation bekommen – außer, Sie programmieren selbst einen Emulator, den Sie dann einbinden.

Was aber muß man in Assemblermodulen, die ja Teil eines Hochsprachenprogramms sind und somit den Hochsprachenemulator nutzen können, machen, wenn man das tun will? Die Antwort ist einfach: Auch beim Assembler gibt es einen Schalter, nämlich die Assembleranweisung:

MASM
```
OPTION EMULATOR
```

Allerdings funktioniert das nur unter MASM! Denn TASM geht hier eigene Wege und nennt den Schalter einfach:

TASM
```
EMUL
```

Wenn Sie also unter MASM die erste und unter TASM die zweite Anweisung in ihren Quelltext einbauen[10], so können Sie sich absolut darauf verlassen, daß Ihnen Ihre Assemblermodule keinerlei Probleme machen. Vorausgesetzt, Sie binden in Ihrer Hochsprache die dazugehörige Bibliothek ein.

---

[10] Wenn Sie nun nicht wissen, welche Anweisung Sie verwenden müssen, machen Sie es sich einfach! Lassen Sie einfach den Assembler entscheiden, welche er möchte. Wie das geht, sehen Sie in den ASM-Dateien zur Routinensammlung MATHE!

Mit diesem Statement könnte ich nun das Kapitel beenden und das Thema zu den Akten legen. Dennoch möchte ich noch etwas im Rahmen dieses Kapitels abklären: Wie funktioniert der Emulator eigentlich? Hierbei werde ich jedoch nicht auf die Routinen eingehen, die den Coprozessorbefehl nachvollziehen.

Sehen wir uns zunächst eine kleine Routine an:

```
Code SEGMENT BYTE PUBLIC 'Code'

ASSUME CS:Code

Three PROC FAR
         fld1
         fadd  st,st
         fld1
         faddp st(1),st
         ret
Three ENDP

CODE ENDS

END
```

Die Prozedur *Three* dient nur dazu, die Fließkommazahl 3.0 zu erzeugen und als Funktionswert im TOS zurückzugeben. Wie sie arbeitet, sollte in der Zwischenzeit kein Geheimnis mehr sein. Was macht der Assembler daraus?

```
CS : 0000 9B            WAIT
     0001 D9E8          FLD1
     0003 9B            WAIT
     0004 D8C0          FADD    ST,ST(0)
     0006 9B            WAIT
     0007 D9E8          FLD1
     0009 9B            WAIT
     000A DEC1          FADDP   ST(1),ST
     000C CB            RETF
```

Die WAITs sind, wie wir wissen, das Tribut an die Kompatibilität mit den 8087-Coprozessoren. Mit der Anweisung *.387* sähe das Disassemblat folgendermaßen aus:

```
CS : 0000 D9E8          FLD1
     0002 D8C0          FADD       ST,ST(0)
     0004 D9E8          FLD1
     0006 DEC1          FADDP      ST(1),ST
     0008 CB            RETF
```

was schon etwas mehr Ähnlichkeit mit unserem Quelltext hat. Mit der Anweisung EMUL wird nun folgender Code erzeugt:

```
CS : 0000 9B            WAIT
     0001 D9E8          FLD1
     0003 9B            WAIT
     0004 D8C0          FADD    ST,ST(0)
     0006 9B            WAIT
     0007 D9E8          FLD1
     0009 9B            WAIT
     000A DEC1          FADDP   ST(1),ST
     000C CB            RETF
```

und zwar unabhängig davon, ob *.387* angegeben wurde oder nicht. Worin bestehen nun die Unterschiede zum ersten Disassemblat? Tatsächlich sieht man auf den ersten Blick keinen. Dennoch besitzt das Assemblermodul bzw. die erzeugte OBJ-Datei im letzten Fall ein wichtiges Detail: Informationen darüber, an welchen Stellen Coprozessorbefehle stehen. Denn bindet man *Three* ohne Emulatoroption z.B. in folgendes Pascal-Programm ein

```
{$N+,E+}

var X : extended;

function Three:extended; external;
{$L THREE.OBJ}

begin
  X := Three;
end.
```

so findet man folgenden Code:

```
CS : 0000 9A00003713    CALL    1337:0000
     0005 9AE5023713    CALL    1337:02E5
     000A 9A0D00D512    CALL    12D5:000D
     000F 55            PUSH    BP
     0010 89E5          MOV     BP,SP
     0012 31C0          XOR     AX,AX
     0014 9ACD023713    CALL    1337:02CD
     0019 E80F00        CALL    002B
     001C CD37          INT     37
     001E 3E            DS:
     001F 50            PUSH    AX
     0020 00CD          ADD     CH,CL
     0022 3D5D31        CMP     AX,315D
```

```
     0025 C0            DB      C0
     0026 9A16013713    CALL    1337:0116
     002B 9B            WAIT
     002C D9E8          FLD1
     002E 9B            WAIT
     002F D8C0          FADD    ST,ST(0)
     0031 9B            WAIT
     0032 D9E8          FLD1
     0034 9B            WAIT
     0035 DEC1          FADDP   ST(1),ST
     0037 CB            RETF
```

Die fetten Stellen sind der Aufruf der Funktion *Three* im Hauptprogramm. Außerdem erkennen wir unsere Assemblerroutine wieder.

Nun das gleiche Pascal-Programm mit dem gleichen Assemblerprogramm – nur mit dem Unterschied der Assemblierung unter der EMUL-Anweisung:

```
CS : 0000 9A00003713    CALL    1337:0000
     0005 9AE5023713    CALL    1337:02E5
     000A 9A0D00D512    CALL    12D5:000D
     000F 55            PUSH    BP
     0010 89E5          MOV     BP,SP
     0012 31C0          XOR     AX,AX
     0014 9ACD023713    CALL    1337:02CD
     0019 E80F00        CALL    002B
     001C CD37          INT     37
     001E 3E            DS:
     001F 50            PUSH    AX
     0020 00CD          ADD     CH,CL
     0022 3D5D31        CMP     AX,315D
     0025 C0            DB      C0
     0026 9A16013713    CALL    1337:0116
     002B CD35          INT     35
     002D E8CD34        CALL    34FD
     0030 C0            DB      C0
     0031 CD35          INT     35
     0033 E8CD3A        CALL    3B03
     0036 C1            DB      C1
     0037 CB            RETF
```

Auch hier interessiert uns nur der fett gedruckte Teil! Doch dort, wo ohne EMUL-Anweisung z.B. `fwait - fld1` stand, steht nun `int 35 - call 34FD`! Was hat das noch mit Coprozessorbefehlen zu tun?

Wie schon weiter oben geäußert, muß im Falle der Nichtexistenz eines Coprozessors zumindest seine Funktion simuliert werden. Dies kann nur durch Prozessorbefehle im Rahmen eines oder mehrerer Unterprogramme geschehen. Die Sammlung der Routinen, die einen Coprozessor simulieren können, nennt man, wie schon gesagt, Emulator. Dieser läßt sich über Interrupts aufrufen. Für ihn wurden die Interrupts $34 bis $3E reserviert.

Die Routinen, die über diese Interrupts aufgerufen werden, haben eine Besonderheit: Sie lesen das auf den INT-Befehl folgende Byte und interpretieren es als Code für den Coprozessorbefehl. Die Folge `int 35 - call 34FD` ist also tatsächlich die Folge `int 35 - DB E8 - int 34 - DB C0`. Sie müßte also als `fwait (emuliert) - fld1 (emuliert)`[11] gelesen werden.

Sie sehen, daß der Assemblercode also noch durch die Hochsprache verändert wird. Allerdings nur, wenn sie die Information darüber bekommt, welche Befehle verändert werden müssen. Dafür sorgt EMUL.

HINWEIS Das heißt aber auch, daß Hochsprachenmodule mit EMUL assemblierte Assemblermodule nutzen können, wenn sie keinen Emulator zur Verfügung stellen. Denn im Assemblermodul stehen ja noch die eigentlichen Coprozessorbefehle. Besitzt ein Hochsprachenmodul keinen Emulator oder soll dieser nicht benutzt werden, so wird das Assemblermodul mit den normalen Coprozessorbefehlen eingebunden. Das fertige Programm läuft dann aber nur noch auf Rechnern mit Coprozessor.

TIP Das bedeutet also, daß man es sich zur Gewohnheit machen sollte, Assemblermodule, die Coprozessorbefehle benutzen, immer mit EMUL zu assemblieren. Man verschwendet damit keinen Speicherplatz oder verlangsamt die Routinen (außer, man benutzt *.387*, denn dann werden unnötige FWAITs eingestreut) und steht dennoch auf der sicheren Seite!

Diese Zusammenhänge klären nun auch auf, warum bei eingeschalteter Emulation immer FWAITs eingestreut werden: Das Byte $9B wird benötigt, um aus Zwei-Byte-Coprozessorbefehlen Drei-Byte-Emulatoraufrufe zu machen. Die FWAITs erfüllen hier also den Zweck der Platzhalter.

Was aber passiert, wenn ein Programm mit einem Emulator erstellt wurde, der Rechner, auf dem es läuft, aber einen Coprozessor besitzt? Dann

---

[11] Womit wir die weiter oben geäußerte Bewunderung über DEBUG schon wieder etwas relativieren müssen. Denn DEBUG ist dumm genug, diese Befehlssequenzen nicht zu verstehen. Dagegen können moderne Debugger wie z.B. CodeView und TD mit solchem Code etwas anfangen! Sie übersetzen ihn vollkommen korrekt in die Mnemonics für die Coprozessorbefehle, die emuliert werden.

sollte doch, nicht zuletzt wegen der Ausführungsgeschwindigkeiten, der Coprozessor benutzt werden und nicht die Emulation. Hier haben sich die Entwickler einen tollen Trick einfallen lassen!

Das Programm besitzt in seinem *Start-up-Code*, also dem Code, der vor dem vom Programmierer erstellten Code automatisch und immer ausgeführt wird, den Aufruf einer Systemroutine. Diese Systemroutine stellt fest, ob ein Coprozessor vorhanden ist. Wird nun ein emulierter Coprozessorbefehl zum erstenmal ausgeführt, so stellt eine Emulatorroutine fest, daß dieser Befehl besser durch den (vorhandenen) Coprozessor ausgeführt wird. Sie verändert daher den Befehl, mit dem sie eben aufgerufen wurde, so, daß dort wieder der originale Coprozessorbefehl steht, und springt an seinen Beginn zurück, womit sie sich selbst aus dem Rennen geworfen hat. Denn ab diesem Zeitpunkt steht dort nur noch der Coprozessorbefehl – und unsere Harakiri-Routine wird nicht mehr angesprungen!

Wenn also z.B. der Prozessor an folgenden Befehl kommt:

```
          :
Hier:     int  35
          DB   E8
UndHier: int   34
          DB   CO
          :
```

so führt er den Interrupt aus. Die dort stehende Routine nun macht aus `int 35 - DB E8` wieder `fwait - fld1`

```
          :
Hier:     fwait
          fld1
UndHier: int   34
          DB   CO
          :
```

und springt zu `Hier` zurück. Ausgeführt wird dann also der eigentliche Coprozessorbefehl, der nun bis zum Programmende dort steht. Weiter geht es dann mit dem nächsten INT-Befehl, der das gleiche tut, so daß im nächsten Schritt

```
          :
Hier:     fwait
          fld1
UndHier: fwait
          fadd st,st
          :
```

nach `fwait fadd st,st` ausgeführt wird – ein sehr schönes Beispiel für selbstmodifizierenden Code. Ohne Coprozessor bleibt es bei den INTs, die dann für die Emulation der Prozessorbefehle sorgen müssen.

ACHTUNG Nicht alle Hochsprachenemulatoren unterstützen alle Coprozessorbefehle. So können Sie einige Befehle nur dann einsetzen, wenn tatsächlich ein Coprozessor vorhanden ist. Bitte konsultieren Sie hierzu Ihr Handbuch.

32 Strukturen

Ein Vorteil, den Hochsprachen haben, ist die Fähigkeit, Daten zu strukturieren. Hierunter versteht man die Möglichkeit, sie zu bestimmten *Strukturen* zusammenzufassen. Solche Strukturen kennen Sie aus Pascal oder C in Form von *arrays*, *records*, *sets*, *structures* oder *unions*.

Diese Strukturierbarkeit bietet offensichtliche Vorteile. Durch sie braucht man sich in der Regel keine Gedanken darüber zu machen, in welcher Reihenfolge und an welcher Stelle im Speicher die einzelnen Elemente der Struktur stehen. Allein durch die Definition der Struktur ist diese Information ein für allemal geklärt und dem Compiler bekannt. Dies entbindet den Programmierer davon, selbst »Buch führen« zu müssen. Auch unter dem Assembler gibt es eine Möglichkeit, Strukturen zu definieren und zu nutzen. Dies ist besonders dann von großem Wert, wenn einer Assemblerroutine ein Zeiger auf eine Hochsprachenstruktur übergeben wird. Betrachten wir daher nun die Strukturen, die der Assembler kennt.

32.1 STRUC

Die wohl wichtigste Struktur unter Assembler ist ein Datensatz, der mittels des Schlüsselwortes *struc* definiert werden kann. Diese Struktur ist identisch mit dem in C verwendbaren *struct* und hat in Pascal die Parallele *record*.

Damit ist eigentlich schon alles gesagt, was man über *struc* wissen muß. Es ist eine Aneinanderreihung von Daten unterschiedlicher, aber konstanter Länge. Die Definition ähnelt der eines Segments oder Makros und wird auch ebenso abgeschlossen:

```
Person STRUC
  Nachname DB 20 DUP (?)
  Vorname  DB 10 DUP (?)
  Alter    DB ?
```

```
  Strasse  DB 20 DUP (?)
  PLZ      DW ?
  Ort      DB 20 DUP (?)
Person ENDS
```

Mit dieser Definition haben wir eine Struktur definiert, die einige persönliche Daten aufnehmen kann. Sie könnte so z.B. in einfachen Datenbanken Verwendung finden.

ACHTUNG

Mit der Definition einer Struktur haben wir noch kein einziges Datum erzeugt! Wie in den Hochsprachen (vgl. *type* in Pascal und *typedef* in C) auch generiert der Assembler die Strukturen selbst erst auf Anweisung! Wir haben dem Assembler lediglich kundgetan, wie er ein Datum dieser Struktur erzeugen und verwalten soll, wenn es angelegt wird.

Einmal definiert, können Sie den Strukturnamen genauso verwenden wie die reservierten Schüsselwörter zur Datendefinition (DB, DW etc.):

```
APerson            PERSON ?
AnotherPerson      PERSON ?
StillAnotherPerson PERSON ?
```

Diese Anweisungen lassen den Assembler Speicherplatz für drei Strukturen gemäß der Definition von *Person* reservieren. Doch Achtung: Die Inhalte der einzelnen Daten sind nicht definiert, da weder bei der Definition der Struktur noch bei der Deklaration ihrer Instanzen jemals ein Defaultwert angegeben wurde. Dies können Sie ändern, indem Sie z.B. die Struktur *Person* wie folgt definieren:

```
Person STRUC
  Nachname DB 20 DUP (?)
  Vorname  DB 10 DUP (?)
  Alter    DB 18
  Strasse  DB 20 DUP (?)
  PLZ      DW ?
  Ort      DB 20 DUP (?)
Person ENDS
```

So erzeugen Sie mit

```
Wilhelm PERSON {}
```

eine Instanz von *Person*, deren Feld *Alter* den Wert 18 hat. Dies geschieht jedoch nur, wenn Sie die beiden Klammern *{}* hinter der Deklaration nicht vergessen. Geben Sie nur

```
Wilhelm PERSON ?
```

an, so ist trotz der Altersvorgabe von 18 bei *Wilhelm* das *Alter* nicht initialisiert!

Sie können aber auch anders vorgehen:

```
Wilhelm PERSON {Vorname='Wilhelm',Nachname='Wendehals'}
```

erzeugt nicht nur eine Instanz der Struktur *Person*, sondern belegt auch die Felder *Vorname* und *Name* mit den betreffenden Angaben. Da hier nicht durch »?« explizit auf eine Initialisierung verzichtet wurde, ist auch *Alter* mit 18 vorbelegt und könnte mit expliziter Angabe eines anderen Wertes überschrieben werden:

```
Eusebia PERSON {Alter=99}
```

HINWEIS Mit den geschweiften Klammern initialisieren Sie die Felder einer Struktur, die Sie explizit mit Namen ansprechen. Es geht aber auch anders. Verwenden Sie nämlich spitze Klammern, so brauchen Sie die Elementnamen nicht anzugeben. Dann aber müssen Sie, falls Sie nicht alle Elemente initialisieren wollen, Kommata verwenden, wie im folgenden Beispiel geschehen, um die Postleitzahl zu initialisieren:

```
Gabriele PERSON <'Mustermann','Gabriele',,,04711>
```

Soweit zur Deklaration und Initialisierung von Daten des Typs *struc*. Doch wie kann man in Programmen auf die einzelnen Strukturelemente zugreifen? Ganz einfach – wie in Hochsprachen auch:

```
Person STRUC
  Nachname DB 20 DUP (?)
  Vorname  DB 10 DUP (?)
  Alter    DB 18
  Strasse  DB 10 DUP (?)
  PLZ      DW ?
  Ort      DB 20 DUP (?)
Person ENDS


DATA SEGMENT WORD PUBLIC 'Data'
  EinName  DB 'Wilhelm'
  APerson  PERSON {NachName='Wendehals'}
DATA ENDS


CODE SEGMENT BYTE PUBLIC 'Code'

ASSUME CS:Code, DS:Data

Start:   mov   ax,SEG Data
         mov   ds,ax
         mov   es,ax
```

```
         mov   si,OFFSET EinName
         mov   di,OFFSET APerson.Vorname
         mov   cx,7
         rep   movsb
         mov   al,APerson.Alter
         mov   ax,04C00h
         int   021h
Code ENDS
END  Start
```

Das Disassemblat sieht dann so aus:

```
CS : 0006 B8D212        MOV     AX,12D2
     0009 8ED8          MOV     DS,AX
     000B 8EC0          MOV     ES,AX
     000D BE0000        MOV     SI,0000
     0010 BF1B00        MOV     DI,001B
     0013 B90700        MOV     CX,0007
     0016 F3            REPZ
     0017 A4            MOVSB
     0018 A02500        MOV     AL,[0025]
     001B B8004C        MOV     AX,4C00
     001E CD21          INT     21
```

Wie Sie sehen können, wurde der Ausdruck *APerson.Vorname* ganz korrekt in eine Adresse »übersetzt«! Auch das Alter ist leicht aus der richtigen Adresse auslesbar. Besonders hilfreich und elegant aber läßt sich dies bei der Übergabe von Datensätzen an Routinen einsetzen:

```
Person STRUC
  Nachname DB 20 DUP (?)
  Vorname  DB 10 DUP (?)
  Alter    DB 18
  Strasse  DB 10 DUP (?)
  PLZ      DW ?
  Ort      DB 20 DUP (á)
Person ENDS


DATA SEGMENT WORD PUBLIC 'Data'
  EinName  DB 'Wilhelm'
  APerson PERSON {NachName='Wendehals'}
DATA ENDS


CODE SEGMENT BYTE PUBLIC 'Code'
```

```
PUBLIC ReadAge
ReadAge PROC FAR
         DataSet = PERSON PTR [bp+6]
         mov   al,[DataSet.Alter]
         ret
ReadAge ENDP
```

DataSet wird als Zeiger auf eine Struktur *Person* definiert und zeigt auf die Adresse auf dem Stack, an der die Hochsprache eine Instanz der Struktur übergibt. Auf das Feld *Alter* läßt sich dann ganz einfach zugreifen.

HINWEIS Beachten Sie bitte, daß hier anstelle von EQU die Anweisung = verwendet wird! Offensichtlich nimmt der Assembler bei Verwendung von EQU eine formale Typprüfung vor, bei = jedoch nicht. Denn wenn Sie oben anstelle von = den Operator EQU verwenden, gibt der Assembler eine Fehlermeldung aus. In diesem Fall müssen Sie mittels eines *Type-Castings* angeben, um welchen Datentyp es sich innerhalb der Struktur handelt, auf die zugegriffen wird.

```
DataSet EQU PERSON PTR [bp+6]
mov      al,BYTE PTR[DataSet.Alter]
```

Aber das steht in keinem Handbuch!

Und das macht der Assembler in beiden Fällen aus dem Quelltext:

```
CS : 0006 8A4624        MOV     AL,[BP+24]
     0009 CB            RETF
```

Da *Alter* 30 Bytes hinter dem Beginn der Struktur und damit hinter dem ersten Byte von *NachName* steht, addiert der Assembler diesen Offset innerhalb der Struktur ($1C = 30 Bytes) zu der Adresse der Struktur (*bp+6*), was in der physikalischen Adresse *bp+24* (hexadezimal) resultiert.

32.2 UNION

Union ist nichts anderes als die Assemblerversion der C-Variante *union*, die ihr Pascal-Pendant in *Varianten Records* hat. Union belegt also den Speicherplatz, den das größte Feld in der Definition der Struktur innehat. Die Definition ist analog zu *struc*:

```
UnionVar UNION
         AByte       DB ?
         AWord       DW ?
         ADoubleWord DD ?
UnionVar ENDS
```

UnionVar ist eine Struktur, die in diesem Fall immer 4 Bytes groß ist, weil das größte Datum 4 Bytes groß ist. Dennoch kann es sein, daß nur ein Byte darin abgelegt wird, wenn nämlich auf die *union* über das Feld *AByte* zugegriffen wird. Alles andere ist absolut gleich wie bei *struc*. Auch *unions* können vorbelegt werden und existieren erst dann, wenn man eine Instanz erzeugt hat.

ACHTUNG

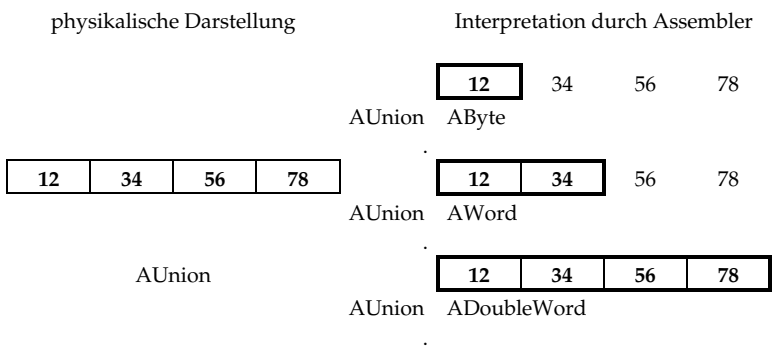
Anders als Strukturen vom Typ *struc* haben solche vom Typ *union* nur ein einziges Element als Inhalt! Daher ist eine Instantiierung der obigen Struktur *Variable* mit

```
AUnion UNIONVAR {AByte= 1, AWord=4711}
```

oder auch

```
AUnion UNIONVAR <1,4711>
```

falsch! In der obigen Definition besteht (je nach Randbedingung!) das Datum *AUnion* entweder aus einem Byte, einem Wort oder einem Doppelwort, aber nicht aus allem! *Unions* dienen dazu, in Abhängigkeit von bestimmten Situationen eine Speicherstelle unter verschiedenen Gesichtspunkten ansprechen zu können:



Dies soll noch einmal an folgendem Beispiel konkretisiert werden:

```
Pointer UNION
  Short DW ?
  Long DD ?
Pointer ENDS
```

Erzeugt man eine Instanz *APointer* von *Pointer*, so kann sie je nach Bedarf einen *far*- **oder** *near*-Zeiger aufnehmen. Je nachdem, welcher Zeigertyp dann tatsächlich enthalten ist, wird er mit *APointer.Short* oder *APointer.Long* angesprochen.

Es ist auf diese Weise sogar möglich, ein *Type-Casting* durchzuführen: Speichert man einen *far*-Zeiger in einer solchen Variablen ab, so läßt

sich mit *APointer.Short* der Offsetteil des *far*-Zeigers entnehmen, was der Umwandlung in einen *near*-Zeiger entspricht (was man allerdings auch einfacher haben kann: *lea ax, FarPointer*).

32.3 Verschachtelte Strukturen

Deutlich mehr Sinn als in den obigen Beispielen macht die Verwendung von *unions* in Strukturen, die Pascal als *Variante Records* bezeichnet. Hier sind *unions* Teil von *strucs*. Auch solche verschachtelten Strukturen sind mit Assembler möglich:

```
Person STRUC
  Vorname  DB 10 DUP (?)
  Nachname DB 20 DUP (?)
  Status   DB ?
  UNION
    STRUC
      Abteilung DB 10 DUP (?)
      Firma     DB 20 DUP (?)
      Postfach  DW ?
      PLZ1      DW ?
      Ort1      DB 20 DUP (?)
      Telefon   DB 10 DUP (?)
    ENDS
    STRUC
      Strasse   DB 20 DUP (?)
      PLZ2      DW ?
      Ort2      DB 20 DUP (?)
    ENDS
  ENDS
ENDS
```

Diese Struktur ist sehr komplex. Zunächst wird eine Struktur *Person* definiert, die neben den Feldern für Vor- und Nachnamen auch eines hat, das die Verwendung des nachgeschalteten varianten Teils steuert: *Status*. Daran schließt sich die Definition einer Varianten an, die keinen eigenen Namen erhält (erhalten kann!), da sie Teil einer Struktur ist (weshalb dann auch letztendlich *Status* notwendig wird). Der variante Teil der Struktur selbst besteht nun ebenfalls aus Strukturen. In diesem Beispiel sind es zwei, die (in Abhängigkeit vom Status) entweder eine berufliche Adresse mit Feldern für Abteilung, Firmennamen und Telefon enthalten oder die private Adresse.

Ich möchte dies hier nicht zu weit vertiefen. Sie können jedoch anhand dieser Definitionen sehen, daß auch unter Assembler struktu-

riert programmiert werden kann und komplexe Strukturen aus Hochsprachen schnell, einfach und effizient verwendet werden können.

32.4 Andere Strukturen

Neben *unions* und *structs* verfügen die meisten Assembler noch über weitere Strukturen wie Aufzählungstypen, Tabellen, Bitfelder usw. Ich möchte jedoch auf die jeweiligen Handbücher verweisen, wenn Sie zu diesen Themen weitergehende Informationen erhalten möchten.

32.5 Assembler und OOP

Auch zu diesem Thema kann ich im Rahmen dieses Buches nicht viel sagen. Dennoch sollen Sie wissen, daß die Assembler von heute ganz selbstverständlich auch die *objektorientierte Programmierung* (OOP) unterstützen. Dies umfaßt so weitgehende Möglichkeiten wie *Vererbung*, *Kapselung* und *Überschreibung* von Objekten sowie die Deklaration *statischer*, *virtueller* oder *dynamischer Methoden*. *Konstruktoren* und *Destruktoren* sind unter Assembler ebenso einfach manipulierbar wie in Hochsprachen. Auf die diversen Tabellen, die ein Objekt beinhaltet (*VMT, DMT* etc.) kann sogar besser zugegriffen werden als aus Hochsprachen (was natürlich auch seine Gefahren birgt!).

33 Vereinfachungen und Ergänzungen

33.1 Speichermodelle

Wie Sie schon gesehen haben, kommt es ganz entschieden darauf an, für welche Hochsprachen Sie die Module schreiben wollen. Zwar bestehen in allen C-Dialekten Möglichkeiten zur Einbindung von Modulen, die in »Fremdsprachen« erstellt wurden, und auch einige Pascal-Dialekte können dies in mehr oder weniger eingeschränktem Maße. Dennoch muß eine Reihe von Besonderheiten berücksichtigt werden.

Turbo Pascal-Programmierer haben es da relativ leicht. Diese Sprache kennt eigentlich nur ein Speichermodell, in dem es ein Datensegment mit bis zu 64 kByte Umfang, ein Stacksegment mit gleicher maximaler Größe und beliebig viele Codesegmente mit ebenfalls maximal 65.536 Bytes Größe geben kann. Obwohl also Datenzugriffe grundsätzlich *near* sein könnten (das DS-Register wird nur einmal mit dem Daten-

segment belegt und bleibt während des Programmlaufs immer konstant – dann braucht man nur noch die Offsets, um an die Daten heranzukommen) – werden bei allen Routinen immer *far*-Zeiger verwendet, wenn Daten *by reference* übergeben werden. Der Grund dafür besteht darin, daß man auch auf den sogenannten *Heap* zugreifen möchte, eine Struktur, die den nicht vom Programm benutzten, freien Rest des Speichers ausmacht. Dazu muß DS verändert werden.

Bei dem ausführbaren Code sieht die Sache schon etwas anders aus! Das Hauptprogramm und alle anderen Programmteile wie etwa Units stehen in eigenen Segmenten, egal wie groß sie sind. Sprünge innerhalb solcher Segmente werden mit *near*-Adressen realisiert, Aufrufe von Teilen, die nicht im gleichen Segment stehen, mittels *far*-Adressen. Segmente unterschiedlicher Art werden nicht verknüpft (Dies ist letztendlich auch der Grund dafür, daß Routinen, die aus Units exportiert werden sollen, immer *far*-Routinen sein müssen!).

C bietet hier mehr Flexibilität, damit aber auch mehr Komplexität. So kennt diese Hochsprache folgende Speichermodelle:

- TINY; Programme, die mit diesem Modell erstellt werden, beherbergen in einem Segment alles, was ein Programm benötigt: Code, Daten und Stack! Damit sind alle Adressen *near*-Adressen, egal, ob auf Daten zurückgegriffen oder an bestimmte Labels gesprungen werden soll. Mit TINY erstellte Programme lassen sich zu COM-Dateien linken. Zu mehr dient dieses Modell allerdings nicht, weshalb seine Bedeutung auch immer mehr abnimmt.
- SMALL; Programme in diesem Modell besitzen je ein Code- und ein Datensegment. Hierbei teilen sich Daten und Stack ein Segment. Aus diesem Grunde sind ebenfalls alle Adressen *near*-Adressen und alle Datenzugriffe *near*-Zugriffe, soweit nicht *far*-Zeiger explizit angefordert werden.
- MEDIUM; dieses Modell stellt mehrere Codesegmente und ein Datensegment zur Verfügung. Es ähnelt somit dem Modell unter Pascal. Wie Sie sich leicht vorstellen können, gibt es daher *near*- und *far*-Adressen, während der Datenzugriff grundsätzlich *near* erfolgt.
- COMPACT; die Umkehrung von MEDIUM heißt COMPACT. Dieses Modell besitzt mehrere Daten- und ein Codesegment.
- LARGE; kombiniert man MEDIUM und COMPACT, so erhält man das LARGE-Modell, in dem es mehrere Code- und Datensegmente gibt. In diesem Modell werden grundsätzlich alle Datenzugriffe und Adressierungen mit *far*-Zeigern erledigt, selbst wenn sie innerhalb eines Segments anzusiedeln sind.
- HUGE; dieses Modell ist eine Erweiterung des LARGE-Modells, bei dem Daten in Segmenten stehen können, die größer als 64

kByte sind. Dies hat gewaltige Vorteile, wenn Sie an Felder denken, aber auch Nachteile: die Zeigerarithmetik kann sich nun nicht mehr nur auf den Offsetanteil beschränken, sondern muß nun auch den Segmentanteil berücksichtigen. Das führt zur Verlangsamung der Ausführungsgeschwindigkeit.

- FLAT; in diesem Modell sind nun wiederum alle Adressen *near*! Es ähnelt dem Modell TINY, hat aber einen gravierenden Unterschied: Während bei TINY nur 16-Bit-Adressen verwendet werden können, sind es im FLAT-Modell 32-Bit-Adressen! Damit können Sie 2^{32} = 4 GByte direkt und linear adressieren. Sie wissen auch, daß dies erst ab dem 80386 möglich ist.
- THUGE; Turbo Assembler ermöglicht auch noch eine Abart des HUGE-Modells mit geänderten ASSUME-Anweisungen sowie **TASM**
- TPASCAL; dieses Modell wird bei älteren Turbo Pascal-Versionen benötigt, entfällt aber bei den neuen Versionen ab 6.0 (wo es nur noch PASCAL heißt).

Da es unter C verschiedene Modelle und somit Adressierungsarten gibt, müssen Assemblermodule dem Rechnung tragen. Dies zu managen kann man dem Assembler recht elegant und einfach übertragen. Hierzu gibt es die Anweisung *.MODEL* (beachten Sie bitte den führenden Punkt!), der Sie unter anderem das zu verwendende Modell übergeben: **.MODEL**

```
.MODEL SMALL
```

Dies hat für sich allein zunächst keine Auswirkung. Allerdings kann in Kombination mit verschiedenen anderen Anweisungen, die wir gleich anschließend besprechen werden, eine erhebliche Vereinfachung bei der Programmierung erzielt werden.

33.2 Vereinfachte Segmentanweisungen

Haben Sie mit *.MODEL* ein Modell gewählt, so können Sie auch die Anweisungen *.CODE, .DATA, .DATA?, .FARDATA, .FARDATA?, .CONST* und *.STACK* verwenden. Aber auch *nur* dann! Beachten Sie bitte auch hier die führenden Punkte.

.CODE ist die Vereinfachung für die in diesem Buch bisher angegebene Anweisung Code *SEGMENT BYTE PUBLIC 'Code'* sowie die Abschlußanweisung *ENDS*. *.CODE* erledigt dies in Abhängigkeit vom gewählten Modell selbst. *.CODE* vergibt einen Segmentnamen, stellt das Segment auf die korrekte Segmentgrenze ein (*Alignment*) und sorgt auch für die Verknüpfung mit anderen Segmenten gleichen Namens.

Es wird Sie nicht verwundern, zu erfahren, daß *.DATA* und alle anderen oben genannten Anweisungen ähnliches mit den unter *C* möglichen verschiedenen Datensegmenten und dem Stacksegment tun, weshalb ich hier nicht näher darauf einzugehen brauche.

Wie kann man sich nun das Leben mit den Anweisungen erleichtern? Ganz einfach:

```
.MODELL LARGE

.FARDATA
AnInitializedFarDate DD 0

.FARDATA?
AnUnInitializedFarDate DB ?

.DATA
AnInitializedNearDate DT 1.0E45

.DATA?
AnUnInitializedNearDate DW ?

.CONST
AReadOnlyDate DW 4711

.STACK 100

.CODE
:
:
END
```

Was Ihnen auffallen sollte, ist:

- Nirgendwo gibt es ein ASSUME. Vergessen Sie es ab jetzt, wenn Sie es nicht ausdrücklich im Code verwenden wollen, um andere Segmente einstellen zu können.
- Sie sehen kein einziges ENDS. Vergessen Sie auch diese Anweisung! Sie brauchen sie nur noch für die Deklaration von Strukturen und Makros. Der Assembler betrachtet alles bis zur nächsten .XXXX-Anweisung als zu dem aktuellen Segment gehörig.
- Es gibt keine langwierige, umständliche und fehlerträchtige SEGMENT-Anweisung mehr. Sie müssen sich um *Alignments*, *Combines* und ähnliches nur noch in Ausnahmefällen kümmern. In solchen Situationen können Sie dann allerdings bedenkenlos auf das bisher Erfahrene zurückgreifen.

Wie Sie sehen, erleichtert diese Art der Anweisungsbenutzung das Programmieren nicht nur erheblich, es trägt auch sehr zur besseren Lesbarkeit des Quelltextes bei.

33.3 Sprachenvereinbarung

Neben dem zu verwendenden Speichermodell kann man *.MODEL* auch ein Symbol übergeben, das angibt, welche Übergabe- und Namenskonvention der Assembler benutzen soll. Wie Sie ja wissen, verlangt C, daß alle extern realisierten, einzubindenden Routinen und/oder Daten mit einem Unterstrich beginnen müssen und daß zwischen Groß- und Kleinschreibung unterschieden wird. Ferner werden Parameter zur Übergabe an Routinen von rechts nach links auf den Stack gelegt. Andere Sprachen arbeiten hier anders.

Diesem Umstand trägt die .MODEL-Anweisung Rechnung. Sie können nämlich die Sprache angeben, zu der das Assemblermodul kompatibel sein soll. Mögliche Schlüsselwörter sind C, BASIC, FORTRAN, PASCAL, SYSCALL und STDCALL.

TASM

Turbo Assembler geht auch hier teilweise eigene Wege: So kennt dieser Assembler die Sprache STDCALL nicht, dafür aber NOLANGUAGE, das auch Voreinstellung ist, sowie PROLOG und CPP.

BASIC, FORTRAN und PASCAL sind identisch. Sie stellen die Übergabekonvention nach Pascal ein, also Parameter von links nach rechts auf dem Stack. Exportierte Routinen-, Label- und Datennamen werden in Großbuchstaben konvertiert, Unterstriche werden *nicht* vorangestellt. Die Verantwortung für das Entfernen der Parameter vom Stack trägt die *gerufene* Routine.

C und STDCALL dagegen stellen (automatisch) Unterstriche voran und unterscheiden, wie auch SYSCALL, das *keine* Unterstriche verwendet, zwischen Groß- und Kleinscheibung. Argumente werden von rechts nach links auf dem Stack erwartet (oder dort abgelegt). Sie müssen bei C von der *rufenden*, bei SYSCALL und STDCALL von der *gerufenen* Routine entsorgt werden.

Die Tabelle auf der folgenden Seite soll dies zusammenfassen.

Ein weiteres Beispiel:

```
.MODEL SMALL, C
```

wählt die Namens- und Übergabekonvention von C sowie das SMALL-Modell dieser Sprache.

	Unterstrich	Groß-/Klein-schreibung	Argumente	Stackbereini-gung
C, CPP	ja	ja	rechts→links	Rufer
SYSCALL	nein	ja	rechts→links	Gerufener
STDCALL	ja	ja	rechts→links	Gerufener*
BASIC	nein	groß	links→rechts	Gerufener
FORTRAN	nein	groß	links→rechts	Gerufener
PASCAL	nein	groß	links→rechts	Gerufener

* falls die Möglichkeit der variablen Argumentliste genutzt wird, muß die rufende Routine für die Stackbereinigung sorgen.

33.4 Argumentanweisungen

Wozu spezifiziert die Anweisung *.MODEL* auch die Reihenfolge der Argumente auf dem Stack? Eine Namenskonvention ist verständlich, weil der Assembler ja kompatible Namen generieren soll. Aber Übergabekonvention? An welcher Stelle konnten wir dies bisher nutzen? Antwort: Bisher nicht! Aber ab jetzt!

Wenn ich Sie bis zu diesem Punkt nur mit den zugegebenermaßen unbequemen, überholten, unhandlichen und schwer nachvollziehbaren, archaischen Möglichkeiten des Assemblers bekannt gemacht habe, so war dies aus didaktischen Gründen in Ihrem Interesse notwendig. Denn wenn Sie die schwierigen Zusammenhänge verstanden haben, fällt Ihnen der Umgang mit dem Assembler leichter. Sie wissen nun, wie auf Parameter usw. zurückgegriffen wird und welchen Code der Assembler dazu generiert.

Wenn Sie nun die wesentlichen Erleichterungen der modernen Assembler vorgestellt bekommen, können Sie beim Erstellen des Quelltextes die einfachen und hochsprachenähnlichen Möglichkeiten nutzen. Beim Betrachten des Disassemblats mit *Debuggern* zwecks Fehlersuche dagegen können Sie den generierten Code lesen und nachvollziehen. Denn eines sollten Sie im Auge behalten: Die Vereinfachungen, die wir in diesem Kapitel betrachten, werden vom Assembler wieder in die einzelnen Prozessorbefehle übersetzt. Die Prozessoren kennen eben keine symbolischen Namen, sondern nur Adressen!

Wie würden Sie in C eine Routine deklarieren, der zwei Parameter übergeben werden?

```
void TestProc(char Var1, int Var2);
```

Unter Turbo Pascal sieht dies ganz ähnlich aus:

```
procedure TestProc(Var1:char; Var2:Integer);
```

Und in Assembler? Ebenso einfach:

```
TestProc PROC FAR Var1:BYTE, Var2:WORD
```

Genau hier liegt die Notwendigkeit, mittels *.MODEL* die Übergabekonvention anzugeben. Denn die C-Routine legt, wie wir wissen, zuerst *Var2* auf den Stack und dann *Var1*, während Turbo Pascal dies genau andersherum tut. Nun muß aber die Assemblerroutine wissen, wo welcher Parameter steckt.

Wurde mittels *.MODEL* daher die Pascal-Konvention gewählt, so verfährt der Assembler genau so, als hätten Sie ein Symbol namens *Var1 EQU BYTE PTR [BP+8]* sowie eines über *Var2 EQU WORD PTR [BP+6]* (der Stack ist wortorientiert) erzeugt. Mit *.MODEL C* dagegen betrachtet der Assembler *Var1* als *EQU BYTE PTR [BP+6]* und *Var2* als *EQU WORD PTR [BP+8]*.

HINWEIS

Beachten Sie bitte, daß im Disassemblat tatsächlich der im Quelltext stehende Befehl

```
mov     ax,Var2
```

angegeben wird als

```
mov     ax,[bp+6]
```

wenn wir den Pascal-Fall betrachten. Es war also sinnvoll, erst die »schwierigere« Art der Programmierung kennenzulernen.

TASM

Der Turbo Assembler macht im Falle der Angabe *.MODEL PASCAL* noch mehr: Er modifiziert automatisch den RET-Befehl so, daß die angegebenen Parameter vom Stack genommen werden! Sie dürfen daher bei der Nutzung der vereinfachten Anweisungen keinesfalls mehr die RET-Version verwenden, bei der Sie die zu entfernenden Bytes angeben! Denn dies würde zu katastrophalen Säuberungsaktionen führen!

HINWEIS

Sobald Sie in den Routinenkopf eine Variablenliste einfügen, richtet der Assembler auch einen Stackrahmen ein. Tun Sie es also nicht auch! Wenn man nachdenkt, ist das auch logisch. Denn die Deklarationen in der Variablenliste sind ja eigentlich nichts anderes als *EQU xxx PTR [BP+yy]* Anweisungen. Dazu muß aber der Stackrahmen schon bestehen.

Also – entweder die vereinfachte Anweisung:

```
TestProc PROC FAR Var1:BYTE, Var2:WORD, Var3:DWORD
         mov   ax,Var2
         mov   bl,Var1
         div   bl
         ret
TestProc ENDP
```

oder die archaische, aber im Ergebnis identische *Hardliner*-Version:

```
TestProc PROC FAR
Var1 EQU BYTE PTR [BP+12]
Var2 EQU WORD PTR [BP+10]
Var3 EQU DWORD PTR [BP+6]
         push  bp
         mov   sp,bp
         mov   ax,Var2
         mov   bl,Var1
         div   bl
         mov   sp,bp
         pop   bp
         ret   8
TestProc ENDP
```

ACHTUNG Wenn Sie die Vereinfachungen nutzen wollen, bedeutet dies aber auch, daß Sie tatsächlich im Assemblermodul alle Parameter angeben müssen, die übergeben werden. Denn andernfalls blieben Bytes auf dem Stack zurück!

33.5 Lokale Variablen

Auch für lokale Variablen gibt es Erleichterungen: Das Schlüsselwort *LOCAL* dient dazu, solche zu definieren. Dies erfolgt unmittelbar nach dem Routinenkopf:

```
TestProc PROC FAR Var1:BYTE, Var2:WORD
LOCAL LocalVar:WORD
         mov   ax,Var2
         mov   LocalVar,ax
         :
```

Der Assembler macht hier auch nichts anderes, als was wir bisher von Hand programmiert hätten:

```
TestProc PROC FAR
Var1 EQU BYTE PTR [BP+8]
Var2 EQU WORD PTR [BP+6]
LocalVar EQU WORD PTR [BP-2]
         sub   sp,2
         mov   ax,Var2
         mov   LocalVar,ax
         :
```

HINWEIS Beachten Sie bitte, daß der Assembler die beiden fett gedruckten Sequenzen identisch behandelt. Sie dürfen also die Zeile `sub sp,2` im

zweiten Beispiel nicht selbst programmieren: Dies erledigt der Assembler über die LOCAL-Anweisung für Sie!

ACHTUNG

Die LOCAL-Anweisung muß tatsächlich unmittelbar hinter dem Routinenkopf stehen! Denn LOCAL hat noch einen Zwilling, den wir weiter unten noch kennenlernen werden. Wenn Sie LOCAL an beliebigen Stellen definieren würden, käme der Assembler durcheinander. Da er das nicht will, beschwert er sich lieber. Wenn Sie jedoch daran denken, daß LOCAL im wahrsten Sinne des Wortes lokal definiert ist, dann bleibt Ihnen eigentlich gar nichts anderes übrig, als es zwischen dem Prozedurkopf und dem dazugehörigen ENDP anzusiedeln! Denn mit den Befehlen der Routine tun Sie das ja auch.

HINWEIS

Auch bei dieser Anweisung wird ein Stackrahmen eingerichtet, wenn nicht schon durch eine Variablenliste einer besteht.

33.6 Register retten

Auch dies ist eine angenehme Möglichkeit! Häufig schleicht sich ein Fehler ein, wenn man am Ende einer Routine alle zu Beginn geretteten Register restaurieren will:

```
:
:
push    bx
push    ax
push    cx
push    dx
push    si
push    di
push    ds
push    es
:
:
:
:
pop     es
pop     di
pop     ds
pop     si
pop     dx
pop     bx
pop     ax
ret
```

USES

Wahrscheinlich ist das Poppen des gesicherten Inhalts von CX in BX noch der harmlosere der beiden Fehler! Wissen Sie, welchen anderen Fehler ich im Auge habe? Dieser ist fatal: Das Restaurieren des DS-Registers mit dem Inhalt von DI wird mit einiger Wahrscheinlichkeit ein falsches Datensegment setzen, was beim nächsten Zugriff zu Problemen führen wird. Ebenso wird der Inhalt des Datensegments in DI nicht das gewünschte Resultat ergeben. Und – sind Sie sicher, alles restauriert zu haben? Ab jetzt: ja! Denn es gibt *USES*.

```
TestProc PROC FAR USES bx, Var1:Word
         mov   ax,Var1
         mov   bl,10
         div   ax
         ret
TestProc ENDP
```

Diese Sequenz erzeugt das gleiche, als wenn Sie folgenden Text assembliert hätten:

```
TestProc PROC FAR
Var1 EQU WORD PTR [BP+6]
         push  bp
         mov   bp,sp
         push  bx
         mov   ax,Var1
         mov   bl,10
         div   ax
         pop   bx
         mov   sp,bp
         pop   bp
         ret   2
TestProc ENDP
```

Bequem können Sie nun auf die Eigenheiten der Hochsprachen eingehen, also z.B. die in C wichtigen Register SI und DI sichern mit *USES si di* oder in Pascal das ach so wichtige Datensegment DS mit *USES ds*. Sie müssen sich nicht mehr um Anzahl und Reihenfolge beim Pushen/ Poppen kümmern. Dazu hat man den Assembler.

TIP

Um das teilweise beharrliche Sich-Verweigern des Assemblers bei Verwendung von *USES* zu umgehen, achten Sie darauf, daß die Register, die gerettet werden sollen, nicht durch Kommata getrennt werden. Allerdings muß nach dem letzten Register sehr wohl ein Komma stehen, wenn weitere Anweisungen kommen, wie z.B. eine Variablenliste! Also:

```
TestProc PROC FAR USES ax bx cx si di, Var1:WORD, Var2:BYTE
```

33.7 Automatische Sprungzielanpassung

Bei der automatischen Sprungzielanpassung handelt es sich um eine Erleichterung, die ich sehr gern benutze. Ich weiß nicht, wie Ihr Programmierstil ist, aber ich setze mich einfach hin und baue einen Prozedurrahmen, den ich dann im Verlauf der Sitzung weiter und weiter ausfülle. So wächst meine Prozedur ständig. Regelmäßig erhalte ich dann ab einem bestimmten Stadium der Programmentwicklung eine unschöne Fehlermeldung vom Assembler, die mir sagt, daß das Sprungziel außer Reichweite ist!

Der Grund: Durch zusätzlich eingefügten Code zwischen dem Ort einer Entscheidung und dem Sprungziel ist dieses über die maximalen 128 Bytes Entfernung hinausgerutscht, die mit bedingten Sprungbefehlen überbrückt werden kann. Dieser Fehler ist sehr schnell behoben. So können Sie aus

```
          :
Start:    :
          :
          jbe   Start
          :
```

schnell die Konstruktion

```
          :
Start:    :
          :
          :
          :
          ja    Next
          jmp   Start
Next:     :
```

machen, die offensichtlich logisch identisch ist. Allerdings kann dies recht schnell recht müßig werden, wenn Sie größere Programmteile haben und durch das Einführen einer Sequenz das Sprungziel eines anderen bedingten Sprunges ebenfalls außer Reichweite rutscht. Dann kommt es statt der vermeintlich fehlerlosen Assemblierung zum nächsten Fehler und so weiter.

Sowohl MASM als auch TASM bieten jedoch die Möglichkeit, die Sprungziele automatisch anpassen zu lassen, d. h., daß die eben gezeigten Änderungen unbemerkt von Ihnen im Hintergrund bei der Assemblierung stattfinden. Wohl gemerkt: *bei der Assemblierung!* Am Quelltext wird nichts verändert.

Das einzige, was Sie tun müssen, um diese automatische Sprungweitenkorrektur durchführen zu lassen, ist, eine Assembleranweisung in den Quelltext aufzunehmen:

MASM OPTION LJMP bzw. OPTION NOLJMP

TASM JUMPS bzw. NOJUMPS

In beiden Fällen schaltet die erste Anweisung die Automatik ein, die zweite Anweisung schaltet sie wieder aus.

ACHTUNG Ein kleiner Schönheitsfehler muß allerdings noch erwähnt werden! Wie Sie ja wissen, kann der JMP-Befehl aus unterschiedlichen Byte-Sequenzen zusammengesetzt und daher unterschiedlich lang sein. Somit unterstellen beide Assembler bei der Assemblierung zunächst einmal den schlimmsten Fall und reservieren Bytes für die längstmögliche Konstruktion. Stellt sich dann heraus, daß der JMP-Befehl kürzer codiert werden kann, wird zwar der kürzere Befehl verwendet, da die Assemblierung aber (meistens) in einem Durchgang erfolgt (*one pass*), können die entstehenden Lücken nicht entfernt werden. Denn dies hieße ja, daß nachfolgende Adressen wiederum korrigiert werden müßten, was in einem Schritt nicht geht! Daher werden die Lücken mit NOPs gefüllt.

Das verursacht im Prinzip keinerlei Verlust an Ausführungsgeschwindigkeit, da die NOPs hinter dem JMP-Befehl stehen. Aber es sieht nicht schön aus und kostet Bytes. Dennoch ist es eine bequeme Art der Programmierung, weil der Assembler nie mehr Fehler aufgrund zu weit entfernter Sprungziele erzeugt.

33.8 Lokale Sprungziele

Ein Nachtrag zu Sprungzielen sei noch angemerkt. MASM und TASM verhalten sich hier etwas anders, wenn auch beide Assembler im Endeffekt das gleiche realisieren können.

MASM Unter MASM sind standardmäßig alle Labels, die durch die »:«-Anweisung deklariert werden, lokal definiert! Was bedeutet das?

Fall 1

```
Proc1 PROC NEAR
         mov   ax,00001h
         jmp   Ende
LabelA:  mov   ax,00002h
Ende:    ret
Proc1 ENDP

Proc2 PROC NEAR
         mov   ax,00003h
```

```
          jmp   Ende
LabelB:   mov   ax,00004h
Ende:     ret
Proc2 ENDP
```

Wenn Sie diesen Quelltext mit MASM übersetzen, so erhalten Sie keine Fehlermeldung! Denn das Label `Ende` hat nur *lokale* Gültigkeit, existiert also als Sprungziel nur *innerhalb* der jeweiligen Routine. So führt Zeile 2 in `Proc1` einen Sprung in Zeile 4 von `Proc1` aus, während Zeile 2 in `Proc2` analog zu Zeile 4 von `Proc2` verzweigt. Das geht so weit, daß `LabelA` aus `Proc2` und `LabelB` aus `Proc1` nicht anspringbar, weil nicht sichtbar ist.

Nun möchte man aber eventuell dennoch einmal Labels anspringen können, die außerhalb des »sichtbaren Bereichs« liegen. Dies kann man dadurch erreichen, daß man `LabelA` und `LabelB` mit einem doppelten Doppelpunkt versieht:

Fall 2

```
Proc1 PROC NEAR
          mov   ax,00001h
          jmp   Ende
LabelA:: mov    ax,00002h
Ende:     ret
Proc1 ENDP

Proc2 PROC NEAR
          mov   ax,00003h
          jmp   Ende
LabelB:: mov    ax,00004h
          jmp   LabelA
Ende:     ret
Proc2 ENDP
```

Solchermaßen deklarierte Labels sind global verfügbar. Allerdings funktioniert das ganze nur, wenn mittels *.MODEL* eine Sprache angegeben wurde. Unterbleibt dies, gibt es auch keine globalen Labels mit dem *zweifachen Doppelpunkt*! In diesem Fall werden solchermaßen deklarierte Labels als lokale Labels betrachtet.

ACHTUNG

Das jedoch bedeutet nicht etwa, daß globale Labels aus anderen Modulen öffentlich zugänglich sind! Sie sind lediglich *innerhalb des Quelltextes* für den Assembler als *nicht lokal* definiert, was nicht das gleiche wie *public* ist!

Es gibt aber auch noch eine andere Möglichkeit: Sie können mit Hilfe von

```
OPTION SCOPED bwz. OPTION NOSCOPED
```

die allgemeine Betrachtung des Assemblers über lokale Gültigkeit ins rechte Licht rücken. Denn der oben genannte Mechanismus gilt für die Standardeinstellung *scoped*. Wird dagegen die Assembleranweisung *OPTION NOSCOPED* eingegeben, so erhalten Sie im Fall 1 sehr wohl eine Fehlermeldung, da alle Labels eine globale (*noscoped*) Gültigkeit haben (nur innerhalb dieses Quelltextes) und somit das Label `Ende` verbotenerweise zweimal definiert wird. Und das mag der Assembler nicht.

Leider gibt es in diesem Betrachtungsmodus keine Möglichkeit, einzelne Labels lokal zu interpretieren! Einen »umgekehrten zweifachen Doppelpunkt« gibt es nicht, was bedeutet, daß in diesem Modus *immer alle Labels global* sichtbar sind.

TASM

TASM geht standardmäßig genau vom Gegenteil aus: *alle Labels sind global sichtbar*, weshalb Fall 1 zu einer Fehlermeldung führt, da `Ende` zweimal deklariert wird! Kein Problem dagegen ergibt sich aus diesem Grund jedoch beim Anspringen von `LabelA` aus `Proc2`. Nun würde dies aber bedeuten, daß unter TASM keine *lokalen* Labels deklarierbar wären. Und das geht nun beileibe nicht!

`LOCALS bzw. NOLOCALS`

Mit der Anweisung *LOCALS* teilen Sie dem Assembler mit, daß Sie einen bestimmten Typ von *lokal* zu betrachtenden Labels nutzen wollen. Solche Labels müssen immer mit @@ beginnen:

```
LOCALS

Proc1 PROC NEAR
         mov   ax,00001h
         jmp   @@Ende
LabelA:  mov   ax,00002h
@@Ende:  ret
Proc1 ENDP

Proc2 PROC NEAR
         mov   ax,00003h
         jmp   @@Ende
LabelB:  mov   ax,00004h
         jmp   LabelA
@@Ende:  ret
Proc2 ENDP
```

`@@Ende` sind in diesem Fall lokal sichtbare Labels, während mit »:« deklarierte Labels im gesamten Modul verfügbar sind.

Zusammenfassend kann also festgestellt werden:

- Der *SCOPED*-Modus bei MASM entspricht dem *LOCALS*-Modus von TASM. Beide Assembler benutzen die »:«-Deklarierung von Labels hier unterschiedlich. Während MASM solche Labels als lokal betrachtet, sind sie unter TASM global.
- Soll bei MASM in diesem Modus ein global verfügbares Label erzeugt werden, so muß dies durch die »::«-Deklaration erfolgen. Soll dagegen unter TASM ein lokales Label zur Verwendung kommen, wird dies zwar mit »:« deklariert, die ersten beiden Zeichen müssen dann aber @@ sein (übrigens kann man auch zwei eigene Zeichen definieren – siehe Anhang).
- Der gegenteilige Modus heißt bei MASM *NOSCOPED*, bei TASM *NOLOCALS*. In diesem Modus sind in beiden Fällen alle Labels global verfügbar. Lokale Labels können nicht deklariert werden.

33.9 IDEAL, INVOKE usw.

Leider kann ich im Rahmen dieses Buches nicht auf die weiteren Möglichkeiten der Vereinfachungen mit MASM und/oder TASM eingehen. So gibt es z.B. den *IDEAL*-Modus von TASM oder die *INVOKE*-Anweisung in MASM, die die Einbindung von Hochsprachenroutinen in Assembler erleichtert. Auch lassen sich z.B. einzelne Routinen »anderssprachig« definieren, als es das *.MODEL* vorgibt. Schließlich kennen manche Sprachen variable Argumentlisten und unterstützen deren Verwendung, etc.

Aber es fällt nun immer schwerer, ein allgemein gültiges Assembler-Buch zu schreiben, denn wie Sie eben schon gesehen haben, entfernen wir uns immer mehr vom ursprünglichen Assembler und lernen die sehr potenten Möglichkeiten der modernen Vertreter dieser totgesagten Gattung kennen. Diese unterscheiden sich z.T. erheblich. Was auch nicht verwunderlich ist – denn sowohl Microsoft als auch Borland verfügen ja über eigene Hochsprachencompiler mit eigenen Macken, Schwächen, Fehlern, Fähigkeiten und Glanzlichtern. Und die jeweiligen Assembler sind darauf ab- und eingestimmt.

34 Tips, Tricks und Anmerkungen

34.1 CMPXCHG, und wie die Probleme gelöst werden können

Wie im Referenzteil erwähnt, ist Intel bei der Dokumentation des CMPXCHG-Befehls für den 80486 ein kleiner Fehler unterlaufen, den alle anderen naturgemäß übernommen haben. Daher kann es vorkommen, daß Sie den Befehl CMPXCHG »von Hand« programmieren müssen, falls Sie ihn verwenden wollen. Ob dies sinnvoll ist oder ob man nicht besser den Befehl emulieren sollte, überlasse ich Ihnen (falls Sie keine Programme erstellen wollen, die nur auf 80486ern und Pentium-Rechnern lauffähig sind).

Falls Sie CMPXCHG tatsächlich verwenden möchten, gibt es drei Möglichkeiten, dies zu realisieren:

1. Verwenden Sie einen Assembler, der nachweislich CMPXCHG in die korrekte Opcode-Sequenz übersetzt. MASM 6.0 und TASM 3.2 erledigen dies.

2. Lassen Sie einen beliebigen Assembler, der jedoch 80486-Befehle kennen muß, CMPXCHG im Quellcode falsch übersetzen. Durchsuchen Sie dann mit einem Debugger das Assemblat nach der Sequenz *0F – A6* oder *0F – A7*. Notieren Sie sich dann alle Bytes, die vom Debugger unter diesem Befehl angezeigt werden. Die auf *A6 / A7* folgenden Bytes codieren die Parameter von CMPXCHG und sind extrem wichtig. Nun ersetzen Sie im Quellcode die Zeile mit CMPXCHG gegen die eben notierten Bytes. Ein Beispiel:

   ```
   :
   mov   ax,00000h
   mov   bx,00001h
   mov   cx,00002h
   cmpxchg bx,cx
   :
   ```

   Der Assembler erzeugt nun mit einiger Wahrscheinlichkeit das folgende *falsche* Assemblat:

   ```
   :
   B80000      MOV      AX,0000
   BB0100      MOV      BX,0001
   B90200      MOV      CX,0002
   0FA7CB      CMPXCHG  BX,CX
   :
   ```

Ersetzen Sie nun im Quelltext den Befehl `cmpxchg bx, cx` durch die Opcodes, die der Assembler erzeugt hat, wobei Sie *A6 / A7* durch *B0 / B1* ersetzen, alles andere aber genau beibehalten:

```
:
mov   ax,00000h
mov   cx,00001h
mov   bx,00002h
DB 00Fh, 0B1h, 0CBh
:
```

Sie sehen, nur das falsche Byte $A7 wurde gegen das richtige $B1 ausgetauscht. Nun sollte nach erneuter Assemblierung alles in Ordnung sein.

3. Nehmen Sie sich den Referenzteil vor, und »erzeugen« Sie die Codesequenz selbst. Beachten Sie hierbei, daß
   - jede Sequenz mit *0F* beginnt. Ausnahme: bei Verwendung der erweiterten Register ab 80386 muß zusätzlich das Präfix *66* verwendet werden.
   - nur bei Byte-Operanden als nächster Wert *B0* folgt. Bei Wort- oder Doppelwortoperanden steht hier *B1*.
   - das sich anschließende *mod-reg-r/m*-Byte die korrekten Sachverhalte widerspiegelt!
   - die folgenden Angaben zu eventuell verwendeten Speicheroperanden, insbesondere die Größe des Adreßfeldes (16/32 Bit), stimmen.

Falls Sie mich fragen, dürfte Methode 1 die beste sein, wenn sie ein Update z.B. auf MASM 6.0 oder TASM 3.2 durchführen können und wollen. Ansonsten ist Methode 2 die vielversprechendste. Methode 3 sollte nur von absoluten Profis unter den Assemblerprogrammierern verwendet werden, die wissen, was sie tun (was selten der Fall ist!).

Falls Sie sich dagegen entschließen sollten, CMPXCHG einfach zu emulieren, so hätte ich hier ein Beispiel für die Realisierung. Wozu gibt es Makros? Hier ist eine hervorragende Anwendung dafür! Wir definieren einfach ein Makro mit dem Namen des zu emulierenden Befehls und sehen auch zwei Parameter vor, ganz so wie beim Original (siehe Referenzteil).

```
CmpXchg MACRO P1,P2
   cmp   al,P1
   jz    L1
   mov   al,P1
   jmp   L2
```

```
L1:mov   P1,P2
L2:
ENDM
```

Doch dieses Makro hat einen Haken! Wir können *CmpXchg* nur mit Byte-Operanden betreiben, da wir die Befehle *cmp al, P1* und *mov al, P1* verwendet haben.

Also brauchen wir zwei weitere Makros, die für Wort- und Doppelwortoperanden gedacht sind. Dann aber wird es unübersichtlich und außerdem unbequem. Denn dann können wir gleich auf Makros verzichten und von Hand programmieren.

Doch es gibt einen Weg. Wir müssen jedoch lediglich im Makro drei Fallunterscheidungen treffen. Übergeben wir

- Byte-Operanden, dann müssen *cmp al, P1* und *mov al, P1* benutzt werden,
- Wortoperanden, dann müssen *cmp ax, P1* und *mov ax, P1* benutzt werden und
- ab 80386-Prozessoren die erweiterten Doppelwortoperanden, dann brauchen wir *cmp eax, P1* und *mov eax, P1*.

Alle anderen Befehle sind gleich. Das führt uns zu folgendem Ansatz:

```
CmpXchg MACRO P1,P2
   {wenn Bytes, dann cmp al,P1}
   {wenn Words, dann cmp ax,P1}
   {wenn DWords, dann cmp eax,P1}
   jz    L1
   {wenn Bytes, dann mov al,P1}
   {wenn Words, dann mov ax,P1}
   {wenn DWords, dann mov eax,P1}
   jmp   L2
L1:mov   P1,P2
L2:
ENDM
```

Doch wie realisieren wir dies? Das ist nicht ganz so einfach, wie zunächst erwartet! Zwar könnten wir hier mit bedingter Kompilierung die Fallunterscheidung programmieren, aber woher bekommen wir die Information, ob *P1* nun für ein Byte, Wort oder Doppelwort steht? *P1* ist ja ein Pseudoparameter, der nur als Platzhalter während der Assemblierung dient und nicht etwa einem bestimmten Typ angehört! In ihm steht ja nur Text – und zwar der, den Sie im Quelltext benutzen, um das zu verwendende Register anzugeben. Also können wir nicht prüfen, welchen Datentyp er repräsentiert.

Aber es gibt einen Umweg, der zwar schrecklich kompliziert aussieht, es aber gar nicht ist:

IRP

IPR ist eine Anweisung, die anaolg zu MACRO ein Makro eröffnet, allerdings mit folgender Besonderheit: Das Makro wird in Abhängigkeit von zwei Parametern repetiert. Allgemein besitzt IRP folgende Struktur:

IRP P,<A1,A2,A3,...An>

Es fällt auf, daß IRP keinen Makronamen hat, also nur innerhalb von Makros verwendet werden kann. *P* ist ein Pseudoparameter, der nun der Reihe nach die »Werte« erhält, die in der folgenden spitzen Klammer aufgelistet sind. Dies können wir nutzen:

```
IRP Dummy,<AL,AH,BL,BH,CL,CH,DL,DH>
```

eröffnet also ein Makro, das nacheinander mit den Parametern AL, AH, BL etc. in `Dummy` aufgerufen wird. In Verbindung mit bedingter Assemblierung können wir also ein »Makro im Makro« erzeugen, das prüft, ob dem Makro *CmpXchg* z.B. eines der Byte-Register übergeben wurde:

```
IRP Dummy,<AL,AH,BL,BH,CL,CH,DL,DH>
   IFIDNI <Dummy>,<P2>
      cmp al,P1
   ENDIF
ENDM
```

Das mit IRP definierte Makro muß wie jedes Makro mit ENDM abgeschlossen werden. Innerhalb dieses Makros nun wird mit der Anweisung IFIDNI geprüft, ob die Pseudoparameter *Dummy* und *P2* den gleichen Inhalt haben. Wenn ja, wird der Befehl *cmp al, P1* assembliert, wenn nein, dann nicht.

IRP beginnt seine Tätigkeit, indem es in *Dummy* AL lädt. Die IFIDNI-Anweisung prüft nun, ob *P2* AL enthält. Anschließend repetiert sich IRP mit AH in *Dummy*, dann mit BL usw. Das Ergebnis dieses Makros ist, daß dann und nur dann, wenn *P2* eines der Byte-Register als Operanden enthält, die Zeile *cmp al, P1* assembliert wird. Genau das, was wir wollen!

Doch Ihnen wird als aufmerksamer Leser nicht entgangen sein, daß wir *P2* prüfen, obwohl im Befehl AL mit *P1* verglichen wird. Warum? Ganz einfach! Während *P1* im Originalbefehl sowohl Registeroperanden wie Speicheroperanden annehmen kann, kann *P2* nur ein Register bezeichnen. Prüften wir mit dem Makro *P1*, so müßten wir noch Byte-Speicheroperanden berücksichtigen, was die Sache komplizieren würde. Da ja alle Operanden in ihrer Größe festgelegt sind, wenn nur

einer definiert ist (ein *cmp WordVar, al* ist ja nicht möglich – der Assembler erzeugt dann eine Fehlermeldung), reicht also die Prüfung von *P2* aus.

Noch etwas: was passiert, wenn wir statt `AL` z.B. `al` eingeben, wie wir das üblicherweise tun? Gar nichts! `al` wird wie `AL` behandelt, da es der IFIDNI-Anweisung vollkommen egal ist, ob Groß- oder Kleinbuchstaben verwendet werden. Sollte hier unterschieden werden, so müßte IFIDN benutzt werden.

Wenn wir nun analoge »Makros im Makro« für die anderen Bedingungen konstruieren, so kommen wir zu folgendem Supermakro:

```
CmpXchg MACRO P1,P2
         LOCAL L1,L2
         .386
         IRP Dummy,<AL,AH,BL,BH,CL,CH,DL,DH>
            IFIDNI <Dummy>,<P2>
               cmp  al,P1
            ENDIF
         ENDM
         IRP Dummy,<AX,BX,CX,DX,SI,DI,SP,BP>
            IFIDNI <Dummy>,<P2>
               cmp  ax,P1
            ENDIF
         ENDM
         IRP Dummy,<EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP>
            IFIDNI <Dummy>,<P2>
               cmp  eax,P1
            ENDIF
         ENDM
               jz  L1
         IRP Dummy,<AL,AH,BL,BH,CL,CH,DL,DH>
            IFIDNI <Dummy>,<P2>
               mov  al,P1
            ENDIF
         ENDM
         IRP Dummy,<AX,BX,CX,DX,SI,DI,SP,BP>
            IFIDNI <Dummy>,<P2>
               mov  ax,P1
            ENDIF
         ENDM
         IRP Dummy,<EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP>
            IFIDNI <Dummy>,<P2>
               mov  eax,P1
            ENDIF
```

```
        ENDM
              jmp  L2
        L1:   mov  P1,P2
        L2:
        .8086
ENDM
```

Dieses Makro hat nur noch zwei Erweiterungen erfahren. Mittels *.386* wird der Assembler angewiesen, auch 386er-Befehle zu akzeptieren. Dies ist wichtig, da wir uns ja die Möglichkeit offen lassen wollen, auch die 32-Bit-Register der Prozessoren ab 80386 verwenden zu können. Am Ende des Makros wird wieder auf 8086-Kompatibilität zurückgeschaltet.

Fällt nur noch die Anweisung LOCAL auf. Was hat es damit auf sich? Wir deklarieren im Makro die Label *L1* und *L2*, die wir als Sprungziele bei der Verzweigung verwenden. Da Makros ja keine Unterprogramme sind, sondern bei jedem Aufruf an der betreffenden Stelle eingesetzt werden, würden zwei Aufrufe von *CmpXchg* zweimal die Labels *L1* und *L2* im Quelltext definieren – jedoch an verschiedenen Stellen! Das mag der Assembler überhaupt nicht – mit Recht! Daher bietet er die Möglichkeit, Labels als *lokal* zu definieren. Das ist eine sehr bequeme Methode, die wir, wie wir noch sehen werden, durch einen kleinen Schönheitsfehler im Code erkaufen.

Doch schauen wir uns zunächst einmal an, wie das Makro reagiert, wenn es mit unterschiedlichen Parametern aufgerufen wird:

```
:
CmpXchg bl,cl
CmpXchg ax,si
CmpXchg ebp,esp
:
```

Der Assembler erzeugt hieraus das folgende Assemblat:

```
cs:0000 3AC3          CMP     AL,BL
   0002 7407          JZ      000B
   0004 90            NOP
   0005 90            NOP
   0006 8AC3          MOV     AL,BL
   0008 EB03          JMP     000D
   000A 90            NOP
   000B 8AD9          MOV     BL,CL
   000D 3BC0          CMP     AX,AX
   000F 7407          JZ      0018
   0011 90            NOP
   0012 90            NOP
   0013 8BC0          MOV     AX,AX
```

```
0015 EB03          JMP     001A
0017 90            NOP
0018 8BC6          MOV     AX,SI
001A 66            DB      66
001B 3BC5          CMP     AX,BP
001D 7408          JZ      0027
001F 90            NOP
0020 90            NOP
0021 66            DB      66
0022 8BC5          MOV     AX,BP
0024 EB04          JMP     002A
0026 90            NOP
0027 66            DB      66
0028 8BEC          MOV     BP,SP
```

An den Offsets 0000 bis 000B steht die Folge, die das Makro mit den Parametern BL und CL erzeugt hat. Bis auf die eingestreuten NOPs ist diese Sequenz mit der identisch, die man für die Emulation von CMPXCHG »von Hand« programmieren würde. Zwischen 000D und 0018 folgt der Code für Wortoperanden, danach für DWords. Beachten Sie hierbei, daß DB 66 das Präfix ist, das die (16-Bit-) Registerangaben auf 32 Bit umsetzt. DB 66 – MOV BP,SP ist also als MOV EBP, ESP zu lesen.

Die NOPs sind der oben angesprochene kleine Schönheitsfehler! Sie werden von Assembler wegen und im Rahmen der Assemblierung lokaler Labels eingestreut – vergleichen Sie hierzu die Anmerkungen bei der automatischen Sprungzielanpasssung im letzten Kapitel. Das können Sie leider nicht verhindern – es stört aber auch nicht! Denn NOPs tun ja im wahrsten Sinne des Wortes gar nichts! Es liegt also an Ihnen, ob Sie sich die Sache so einfach wie möglich machen und das Makro benutzen. Dann müssen Sie wohl oder übel die NOPs in Kauf nehmen. Oder sie stören Sie aus irgendwelchen Gründen (Ästhetik, Byte-Geizerei, Taktfetischismus etc.). Dann seien Sie auf die Umschaltung auf manuellen Betrieb verwiesen.

Das Makro befindet sich in der Datei CMPXCHG.ASM auf der CD-ROM.

34.2 CPUID

Ein schönes Feature, das der Pentium da bekommen hat! Schade nur, daß man es nicht auch rückwirkend in die einzelnen Prozessoren einbauen kann! Aber man könnte ja eine Routine bauen, die diesen Befehl emuliert.

In der Datei CPUID.INC wird ein Makro definiert, das ähnlich wie GET_87.ASM prüft, welcher Prozessortyp gerade seinen Job tut. Ist es ein Pentium, so wird der Befehl CPUID ausgeführt. Bei allen anderen Prozessortypen wird ein Verhalten von CPUID simuliert.

Wie Sie das Makro CPUID am sinnvollsten einsetzen können, zeigt Ihnen GET_86.ASM. Dieses Assemblermodul, das ein eigenständiges Assemblerprogramm ohne Hochsprachenteil ist und somit sehr einfache Ausgabemöglichkeiten nutzt, bindet das *Inclusion-File* CPUID.INC ein: ein Beispiel dafür, wie man sich die Arbeit mit *Inclusion-Files* erleichtern kann.

34.3 Optimierung für Fortgeschrittene

Ich werde Ihnen hier anhand der Routinen aus *Mathe* zwei Optimierungsmöglichkeiten aufzeigen. Bei der ersten handelt es sich um das Problem der Übergabe von Funktionswerten aus lokalen Routinen via Routine an den rufenden Teil. Das zweite Beispiel zeigt die Nutzung (und Problematik der Nutzung) des Stackrahmens einer Routine durch ein lokales Unterprogramm.

Beachten Sie bitte, daß dies zunächst nur aufgrund der Eigenheiten von Turbo Pascal und daher nur für Turbo Pascal erfolgt. Vergleichen Sie daher die Kommentare in den unterschiedlichen MATHE*.ASM-Versionen!

Funktionswerte

Stellen Sie sich folgendes Pascal-Programm vor. Lokale Funktionen sind unter Pascal nicht selten. C-Programmierer können hier getrost weghören: Unter C gibt es keine lokalen Routinen!

```
function TwoOrThree:extended;
  function Number:extended;
  begin
    if GottGegeben then Number := 2.0
    else Number := 3.0
  end;
begin
  {hier passiert irgendwas}
  TwoOrThree := Number;
end;
```

Wenn Sie dann im Hauptprogramm mit

```
:
X := TwoOrThree;
:
```

einer Variablen das Ergebnis der Funktion zuweisen wollen, passiert folgendes. Zunächst das Listing von der Funktion *Number*:

```
                        :
CS : 0008 55            PUSH    BP
     0009 89E5          MOV     BP,SP
     000B 83EC0A        SUB     SP,+0A
     000E 803E740000    CMP     BYTE PTR [0074],00
     0013 740E          JZ      0023
     0015 CD3C          INT     3C
     0017 99            CWD
     0018 06            PUSH    ES
     0019 0000          ADD     [BX+SI],AL
     001B CD37          INT     37
     001D 7EF6          JLE     0015
     001F CD3D          INT     3D
     0021 EB0C          JMP     002F
     0023 9B            WAIT
     0024 2E            CS:
     0025 D9060400      FLD     DWORD PTR [0004]
     0029 9B            WAIT
     002A DB7EF6        FSTP    TBYTE PTR [BP-0A]
     002D 9B            WAIT
     002E DB6EF6        FLD     TBYTE PTR [BP-0A]
     0031 9B            WAIT
     0032 89EC          MOV     SP,BP
     0034 5D            POP     BP
     0035 C20200        RET     0002
```

Die Funktion richtet einen Stackrahmen ein und schafft Platz für eine lokale Variable vom Typ *extended*. Hier wird das Funktionsergebnis temporär abgelegt. Lassen Sie sich durch die INT-Befehle nicht irritieren: Turbo Pascal benutzt auch im Coprozessormodus für gewisse Zwecke (hier für das Laden von Konstanten) eine Schmalspurversion eines Emulators! Mit `fld dword ptr [0004]` wird die eigentlich interessante Sequenz eingeleitet.

Wie Sie sehen, wird die Konstante zunächst in den TOS geholt und dann in der temporären Variablen abgelegt, um anschließend sofort von dort wieder ausgelesen zu werden, da sie ja der Funktion *TwoOrThree*, die *Number* aufruft, im TOS als Ergebnis übergeben wird:

```
0045 55             PUSH    BP
0046 89E5           MOV     BP,SP
0048 83EC0A         SUB     SP,+0A
004B BF8A01         MOV     DI,018A
004E                :
```

```
  :                :
0068 E89DFF        CALL    0008
006B 9B            WAIT
006C DB7EF6        FSTP    TBYTE PTR [BP-0A]
006F 9B            WAIT
0070 DB6EF6        FLD     TBYTE PTR [BP-0A]
0073 9B            WAIT
0074 89EC          MOV     SP,BP
0076 5D            POP     BP
0077 C3            RET
```

Auch diese Funktion richtet ihren Stackrahmen ein und reserviert 10 Bytes lokal für das temporäre Speichern eines Funktionsergebnisses. Gegen Ende der Routine wird *Number* aufgerufen. Der im TOS übergebene Wert wird lokal und temporär zwischengespeichert, um sofort wieder ausgelesen zu werden, da ja *TwoOrThree* das Funktionsergebnis von *Number* als Ergebnis zurückgibt. Dann wird *TwoOrThree* beendet. Das Hauptprogramm, das *TwoOrThree* nun seinerseits aufruft, speichert den Wert im TOS schließlich an der Stelle, wo er hin soll.

Zeichnen wir kurz den Weg der Konstanten, welchen Wert sie auch immer hat, nach:

- Erzeugung in der Routine *Number* auf dem Coprozessor*stack*
- lokale Zwischenspeicherung in *Number* auf dem Stack des Prozessors
- Sofortiges Rücklesen auf den Coprozessorstack zur Übergabe
- Unmittelbare lokale Zwischenspeicherung in *TwoOrThree* auf dem Stack
- Sofortiges Zurückladen in den TOS zwecks Wertübergabe
- Endgültiges Abspeichern im Hauptprogramm

Hier sind offensichtlich zumindest die Befehle überflüssig, mit denen die Konstante in *TwoOrThree* zwischengespeichert wird. Wie an anderer Stelle gesagt: Die Reaktion des Compilers ist nicht erstaunlich. Mit den relativ schnellen Prozessorbefehlen wäre das auch nicht so tragisch. Doch Speicheroperationen des Coprozessors dauern relativ lange. An dieser Stelle muß dann ggf. von Hand optimiert werden.

TIP

Vermeiden Sie Funktionen mit Realzahlen, wenn abzusehen ist, daß solche Funktionen lokal in anderen Funktionen verwendet werden. Auch hier ist die Verwendung von Prozeduren, denen eine Variable *by reference* übergeben wird, wesentlich sinnvoller. Vor allem Routinen, die im Rahmen von Schleifen eingesetzt werden können, reduzieren hier z.T. die Ausführungszeit erheblich.

Die andere Alternative ist das, was ich Ihnen eigentlich zeigen wollte. Sie ist in der Unit *Mathe* mit den periodischen Funktionen realisiert und soll nun am Beispiel der Funktion *Sin* demonstriert werden. Sie wird folgendermaßen implementiert:

```
function Sin(X:extended):extended;
begin
  Trans($F0,$FF,$99,$00);
end;
```

Letztendlich dient sie nur als Ladefunktion der Assemblerroutine *Trans*[12]. Wenn wir einfach davon ausgehen, daß das stimmt, was ich im Kapitel über die Optimierungen zu den periodischen Funktionen erläutert habe, haben wir hier genau das oben geschilderte Problem! *Trans* liefert ein Funktionsergebnis auf dem TOS zurück, das *Sin* in einer temporären, lokalen Variable zwischenspeichert, um es sofort wieder in den TOS zurückzuladen ...

... wenn nicht *Trans* mit einer ganz speziellen Codesequenz beendet würde:

```
         :
         :
Stop1:   pop   bx
         pop   cx
         call  Restore
         mov   sp,bp
         pop   bp
         ret   10
Trans ENDP
```

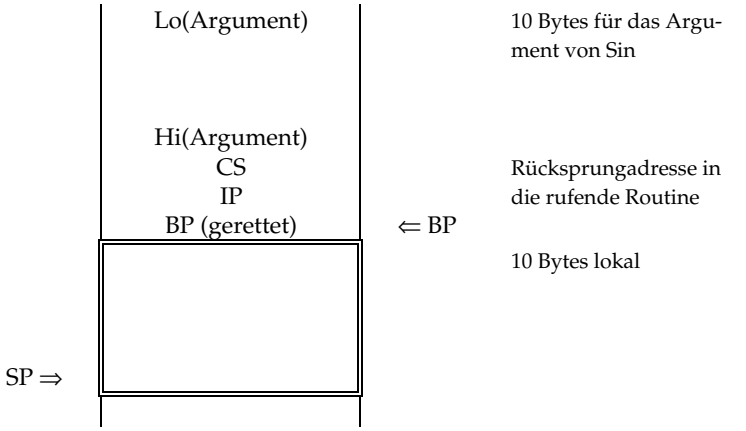
Trans erhält laut Deklaration nur vier Bytes (die über den Stack als Worte übergeben werden!) als Argumente. Warum wird sie dann mit `ret 10` beendet, das ja zehn Bytes vom Stack entfernt?

Beginnen wir mit *Sin*. Diese Funktion legt einen Stackrahmen an und reserviert zehn Bytes für die lokale Variable, in der das Funktionsergebnis gespeichert wird. Der Stack sieht dann in etwa wie in der Abbildung auf der folgenden Seite gezeigt aus.

Wenn die Funktion nun *Trans* aufruft, so legt sie zuvor noch die vier Worte des Steuercodes für *Trans* auf den Stack.

---

[12] Sie sollten sich hier zumindest kurz wundern, wie *Trans* seine Funktion erfüllen kann, wenn es den Winkel gar nicht übergeben bekommt! Aber das sehen wir gleich.



ACHTUNG

Obwohl *Trans* im Assemblermodul als *far* deklariert ist, springt *Sin* diese Routine nur *near* an! Der Grund hierfür liegt darin, daß der Compiler beide Module, also das Hochsprachenmodul und das Assemblermodul, im gleichen Segment unterbringt! Dann sind defaultmäßig alle Sprünge *near*!

HINWEIS

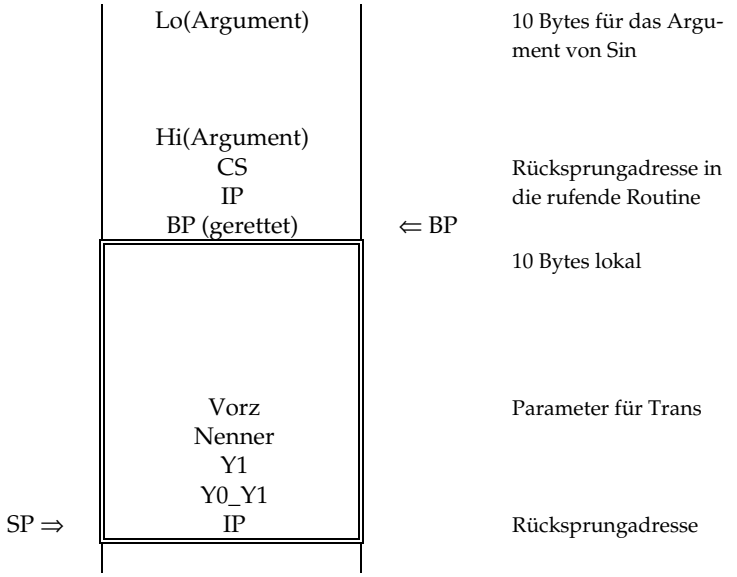
Lernen wir daraus, daß Deklarationen meistens nicht über das entsprechende Modul hinausgehen: *near/far*-Deklarationen im Assemblermodul werden nur hier richtig verarbeitet, *near/far*-Deklarationen im Hochsprachenmodul nur dort. Für die korrekte Verbindung müssen *Sie* sorgen!

Dieses Manko ist uns in diesem Fall, wie wir noch sehen werden, sehr recht – und deshalb belassen wir die unterschiedlichen Definitionen der gleichen Routine. In den allermeisten Fällen führt das aber zu Problemen, die sehr schwer zu entdecken sind!

TIP

Achten Sie beim Einbinden von Assemblermodulen in Hochsprachenmodule grundsätzlich darauf, daß die Deklaration im Hochsprachenmodul die gleiche ist wie die Definition im Assemblermodul. Eine als *far* deklarierte Routine muß auch als *far* eingebunden werden. In Pascal kann dies entweder mit dem Compilerschalter {$F+} oder mit der Anweisung *far* hinter dem Prozedurrumpf erfolgen. Zurück zum Listing! *Trans* sieht sich nun mit dem auf der nächsten Seite abgebildeten Zustand konfrontiert.

Weil *Trans near* deklariert ist, wird nur der Offset als Rücksprungadresse auf dem Stack abgelegt. *Trans* legt nun selbst keinen eigenen Stackrahmen an! Das bedeutet, daß auch *Trans* den Stackrahmen der aufrufenden Routine benutzt. Das mag zwar sehr unüblich sein und den allgemeinen Gepflogenheiten widersprechen, aber es ist erlaubt, führt zu keinen Problemen (wenn man aufpaßt) und ist auch sicher!



Was aber heißt das für *Trans*? Die Parameter, die von der aufrufenden Routine (also z.B. *Sin*) übergeben werden, können nicht, wie sonst üblich, über *[bp+nn]* aufgerufen werden! Denn schließlich wurde ja kein »untergeordneter« Stackrahmen eingerichtet. Das heißt also, daß diese Parameter, wie für die aufrufende Routine auch, lokale Parameter sind und über *[bp-mm]* aufgerufen werden müssen. Weil die rufende Routine eine lokale Variable eingerichtet hat, bevor die Parameter auf den Stack gelegt wurden, liegen diese »unter« der lokalen Variablen. Dagegen haben wir, wie *Sin* auch, auf das Argument, das *Sin* übergeben wurde, ganz normal Zugriff: über *[bp+6]*.

Auch für die Rücksprungsequenz hat das Folgen. Denn eigentlich müßte *Trans* gar keinen Stackrahmen entfernen – es hat ja keinen angelegt. Das aber führt uns zu einem Trick: Warum sollte nicht *Trans* den Stackrahmen von der rufenden Routine, hier *Sin*, entfernen? Da *Trans* ja ausschließlich von Routinen wie *Sin* aufgerufen werden kann und der einzige Befehl in diesen Routinen ist (also kein Zwang zum Rücksprung in *Sin* besteht), können wir praktisch die Ende-Sequenz der rufenden Routine gleich hier ausführen! Der Erfolg: Es wird nicht in *Sin* zurückgesprungen, und die dort durch den Hochsprachencompiler erzeugten (überflüssigen und störenden) Funktionswertverschiebungen unterbleiben.

Beenden wir also *Trans* einfach mit der Endesequenz, die auch *Sin* hat: Stackrahmen entfernen und zehn Bytes vom Stack nehmen, die der Teil dort angelegt hat, der *Sin* aufgerufen hat und in dem das Argu-

ment für *Sin* steht. Daß der RET-Befehl hier ein *retf* ist, weiß der Assembler aufgrund der *far*-Deklaration im Assemblermodul!

Stack

Noch einmal zur Verdeutlichung: Falls eine Routine keinen eigenen Stackrahmen definiert, so greift sie auf den Stackrahmen des »übergeordneten« Programmteils zurück[13]! Falls sie sich dann nicht absolut diszipliniert verhält, entstehen Probleme. Dies soll an folgendem Beispiel demonstriert werden. In der Unit *Mathe* ist die Funktion *Ln* implementiert:

```
PUBLIC Ln
Ln PROC FAR
         fldln2
         jmp   Common1
Ln ENDP
```

Diese Funktion lädt lediglich die Konstante *ln(2)* in den TOS und springt dann an das Label *Common1*.

HINWEIS

Ln hat, wie Sie sehen, keinen Endcode! Sie ruft *Common1* auch nicht via CALL-Befehl, sie springt einfach an ein Label in dieser Routine. Dies geht nur dann problemlos, wenn *Ln* und *Common*, die Routine, in der das Label *Common1* definiert wurde, absolut identische Annahmen über die übergebenen Argumente sowie lokale Parameter machen. In diesem Fall braucht das nicht sonderlich überprüft zu werden, da *Ln* weder Übergabewerte noch lokale Parameter benutzt.

ACHTUNG

Mit dem Laden der Konstanten und dem absoluten Sprung ohne Rückkehr ist *Ln* beendet. Diese Funktion hat vollständig die Kontrolle an das andere Programm, *Common*, abgegeben. Das bedeutet einerseits, daß *Common* die Aufräumarbeiten für *Ln* übernehmen muß, andererseits, daß *Ln* auch beim Assembler als beendet angemeldet werden muß. Daher muß *Ln* nach dem Sprung mit der Assembleranweisung ENDP abgemeldet werden.

```
Common_Log PROC FAR
         X EQU TBYTE PTR [bp+6]
Common1:push   bp
         mov   bp,sp
         sub   sp,4
         call  Save
         fld   X
         fyl2x
```

---

13 Übrigens: Dies ist auch das Problem von Interrupt-Routinen! Diese richten in den seltensten Fällen selbst einen Stackrahmen ein, sondern benutzen den des unterbrochenen Programmteils. Das kann in bestimmten Fällen zu Problemen führen!

```
          call  Check
          jz    C1_1
          jc    C1_1
          mov   [M_Error_],-4
C1_1:     call  Restore
          mov   sp,bp
          pop   bp
          ret   10
Common_Log ENDP
```

Common, dessen Label *Common1* angesprungen wurde, tut nun die eigentliche Arbeit. Hier benötigen wir das Argument der Funktion, weshalb wir hier das Symbol X deklarieren. Da *Common* und *Ln* den gleichen Stackrahmen benutzen, findet *Common* das Argument, das *Ln* übergeben wurde, an der gleichen Stelle, an der es auch *Ln* finden würde. Ferner richtet *Common* auch einen Stackrahmen ein und schafft Platz für lokale Variablen. Abgesehen von den drei CALL-Befehlen, mit denen Unterprogramme aufgerufen werden, fällt uns eigentlich nichts Gravierendes auf. Es wird lediglich der *Logarithmus naturalis* des übergebenen Arguments gebildet, wobei man sich die Beziehung Ln(x) = Ln(2) · Ld(x) zunutze macht. Ld(x) ist über den Befehl FYL2X realisierbar, der außerdem noch die Multiplikation mit der (durch die Funktion *Ln* auf den Stack gelegten) Konstanten Ln(2) übernimmt.

Ein anderes Beispiel. Schauen wir uns einmal die erste der drei aufgerufenen Routinen an:

```
PUBLIC Save
Save PROC NEAR
          Control EQU WORD PTR [bp-4]
          push  ax
          fstcw [Control]
          fwait
          mov   ax,[Control]
          or    ax,0003Fh
          xchg  ax,[Control]
          fldcw [Control]
          fwait
          mov   [Control],ax
          pop   ax
          ret
Save ENDP
```

Save dient dazu, das Kontrollwort des Prozessors so zu verändern, daß Fehler, die bei Rechenoperationen auftreten, abgefangen werden. Es soll jedoch kein Interrupt ausgelöst werden, der z.B. bei einer Divi-

sion durch 0 die allseits beliebte Systemmeldung »Division durch 0« mit anschließendem Abbruch des Programms bewirkt. Vielmehr soll die Fehlerbereinigung innerhalb einer darauf spezialisierten Routine erfolgen. Der vielerorts schon gepriesene und angemahnte »gute Programmierstil« gebietet es, das zu verändernde *Kontrollwort* zwischenzuspeichern und später wieder zu restaurieren. Genau das tut die Routinenkombination *Save – Restore*. *Save* übernimmt die Speicherung und Veränderung des *Kontrollworts* und wird zu Beginn jeder mathematischen Routine aufgerufen, während *Restore* am Ende jeder Routine für die Wiederherstellung sorgt.

Save benötigt also einen Platz, an dem es das *Kontrollwort* sichern kann. Wenn es dazu eine lokale Variable benutzt, muß *Save* einen eigenen Stackrahmen einrichten, und der wird beim Beenden von *Save* wieder entfernt – und somit auch das *Kontrollwort*, das *Restore* noch benötigt. Den kostbaren und knappen Speicher im Datensegment wollen wir mit dem vergleichsweise »unwichtigen« und vor allem temporären Zwischenspeichern eines *Kontrollworts* in einer globalen Variablen auch nicht belasten.

TIP

Es gibt doch einen anderen Weg: *Save* und *Restore* werden ausschließlich im Rahmen von mathematischen Routinen benötigt. Sie werden niemals einzeln für sich aus dem Programm aufgerufen. Warum also sollte man nicht in der rufenden Routine lokalen Platz auf dem Stack reservieren, den dann die lokal aufgerufenen Routinen nutzen können? Dieser wird ja erst entfernt, wenn die rufende Routine beendet wird und somit nach dem Aufruf von *Restore*!

Genau das wurde realisiert. *Common* reserviert vier Bytes lokale Variablen, von denen nun zwei Bytes von *Save – Restore* benutzt werden. Nachdem aber beide Routinen keinen eigenen Stackrahmen einrichten, gilt für sie der gleiche Stackrahmen wie für *Common*. Mit allen Konsequenzen!

Da *Common* nun aber nur den Platz für die lokalen Variablen reserviert, nicht aber selbst darauf zurückgreift, deklarieren wir die Symbole für den Zugriff erst dort, wo sie benötigt werden: in *Save* und *Restore*. Alles andere würde verwirren!

ACHTUNG

Wenn Sie diese Art der Programmierung nutzen, so achten Sie unbedingt darauf, daß sich die lokalen Routinen mit den lokalen Variablen nicht gegenseitig stören. Beispielsweise benutzt *Check*, die dritte lokale Routine, die die Prüfung auf die korrekte Durchführung des Befehls übernimmt, ebenfalls eine lokale Variable, in der sie den Inhalt des *Statusworts* des Coprozessors speichert (Sie wissen ja, daß wir nur über den Umweg einer Speicherstelle Zugriff auf dieses Wort haben). Achten Sie immer darauf, daß es hier keine Konflikte gibt. Die Benut-

zung von Symbolen für den Zugriff erleichert Ihnen hier die Arbeit sehr. Sie müssen dann lediglich bei der Definition der Symbole darauf achten, daß es zu keinen Überschneidungen kommt. Der Rest ist dann einfach:

```
PUBLIC Check
Check PROC NEAR
         Status EQU WORD PTR [bp-2]
         clc
         fstsw [Status]
         mov   [M_Error_],0
         fwait
         test  [Status],0004h
         jz    Check1
         mov   [M_Error_],0FFFFh
         jmp   C_NaN
Check1:  test  [Status],0008h
         jz    Check2
         mov   [M_Error_],0FFFEh
         jmp   C_NaN
Check2:  test  [Status],0010h
         jz    Check3
         mov   [M_Error_],0FFFDh
         jmp   C_NaN
Check3:  test  [Status],0001h
         jz    C_End
C_NaN:   ffree st
         fincstp
         fldz
         fdiv  st,st
         stc
C_End:   fclex ret
Check ENDP
```

Auch die periodischen Funktionen, die, wie wir gesehen haben, schon sehr trickreich arbeiten, benutzen diese Art der Kontrollwortmanipulation. Das aber bedeutet, daß auch diese Routinen Platz für lokale Variablen bereitstellen müssen. *Und zwar dort, wo Save, Check und Restore es erwarten.* Da diese Routinen (wie wir es festgelegt haben) die ersten vier Bytes lokalen Bereichs nutzen (*[bp-2]* und *[bp-4]*), tun sie es auch bei den periodischen Funktionen.

Sin, *Cos* etc. legen hier aber eine lokale Variable mit zehn Bytes Umfang für das Funktionsergebnis an. Dies können wir nicht verhindern, da die »Laderoutinen«, wie ich sie genannt habe, in der Hochsprache programmiert wurden. Wir können nun nicht einfach bewirken, daß

Sin, Cos & Co. nur unserer schrägen Programmierung wegen ihre Richtlinien übergehen! Falls wir also Platz für lokale Variablen benötigen, so müssen wir im Hochsprachenteil lokale Variablen definieren. Diese jedoch werden immer »unterhalb« der lokalen Variablen für das Funktionsergebnis angesiedelt. An einer Stelle also, an der *Save, Check* und *Restore* sie nicht erwarten.

Dieses Dilemma können wir auf verschiedene Weise lösen:

- Wir definieren tatsächlich im Hochsprachenmodul lokale Variablen für das *Kontrollwort* und das *Statuswort*. Diese liegen dann an *[bp-12]* und *[bp-14]*. Dann müssen wir in *Save, Check* und *Restore* auf die lokalen Plätze mit diesen Adressen zugreifen und entsprechende Symboldefinitionen vornehmen. Nachteil: wir müßten *Ln* etc abändern, indem wir ebenfalls zehn Bytes lokalen Speichers »vorschalten«, der dann nicht genutzt wird. Falls wir dann noch weitere Routinen programmieren, die *Save, Check* und *Restore* benutzen, müssen wir aufpassen, daß diese nicht mehr lokalen Speicher voranstellen. In diesem Fall müßten wir alles von vorn neu deklarieren!
- Wir definieren andere Zwillinge von *Save, Check* und *Restore*, die auf unterschiedliche Speicherstellen zugreifen. Nicht sonderlich erstrebenswert, wenn wir dies im Kapitel »Optimierungen für Fortgeschrittene« behandeln!
- Wir greifen auf solche lokalen Variablen indirekt zu. Dann brauchen wir lediglich in den unterschiedlichen Routinen Zeiger auf die entsprechenden lokalen Variablen zu erstellen. Damit haben wir das Problem aber nur verschoben. Denn auch die Zeiger müßten irgendwo gespeichert werden: in lokalen Variablen.
- Wir benutzen doch globale Variablen. Der Datenspeicher ist jedoch zu wertvoll! Es muß auch anders gehen!
- Wir freuen uns, daß wir *Sin & Co.* so programmiert haben, daß niemals auf die lokale Variable für das Funktionsergebnis zurückgegriffen wird. Damit wird sie weder von *Sin & Co.* noch von *Trans* benutzt. Der reservierte Speicherplatz kann bedenkenlos von *Save, Check* und *Restore* benutzt werden. Da die lokale Variable von *Sin* etc. freundlicherweise beginnend mit *[bp-2]* angelegt wurde, brauchen wir nichts zu ändern.

Um solche Programme zu entwickeln, die dann auch problemlos laufen, benötigen Sie Hilfe: einen guten Debugger, möglichst viel Erfahrung, Lust und Zeit! Aber es lohnt sich.

34.4 Daten im Assemblermodul

Wie an anderer Stelle schon erwähnt, mag es Pascal nicht, wenn andere Daten in seinem Datensegment definieren. Mehr noch: Daten, die nicht durch Pascal definiert werden, kennt es überhaupt nicht. Egal, in welchem Segment! Nicht, daß der Compiler Fehlermeldungen ausgibt oder die Mitarbeit verweigert. Nein, viel schlimmer: Er straft solche Daten mit Nichtbeachtung. Dies äußert sich darin, daß Sie in Assemblermodulen angesiedelte Daten nicht im Pascal-Teil verwenden können. Ein Beispiel! Zunächst der Quelltext des Assemblerteils, gleich anschließend dann der Pascal-Teil:

```
Data SEGMENT WORD PUBLIC 'Data'
PUBLIC PrivateWord
PrivateWord DW ?
Data ENDS

Code SEGMENT BYTE PUBLIC 'Code'
ASSUME CS:Code, DS:Data
PUBLIC PublicFunction
PublicFunction PROC FAR
         mov   PrivateWord, 04711h
         :
         :
         mov   ax, PrivateWord
         ret
PublicFunction ENDP
Code ENDS
END
```

```
Program Test;

Function PublicFunction:Word; Far; External;
{$L MyAsm}

Var PublicWord : Word;

begin
  PublicWord := $0815;
  if PublicWord <> PrivateWord then exit
  else PublicWord := PublicFunction;
end.
```

In diesem Fall werden Sie eine Mitteilung vom Compiler bekommen, daß es keine Variable namens *PrivateWord* gibt, obwohl sie korrekt im Datensegment von Pascal liegt und einen korrekten Wert enthält! *Pu-*

blicFunction liefert auch den richtigen Wert zurück! Dennoch läßt der Compiler einen Zugriff auf dieses Datum nicht zu!

Ausweg: definieren Sie eine Prozedur *PrivateWord* im Pascal-Teil sowie eine Variable vom Typ ^Word. Dieser weisen Sie dann die Adresse der Pseudoprozedur zu:

```
Program Test;

Function  PublicFunction:Word; Far; External;
Procedure PrivateWord; Far; External;
{$L MyAsm}

Var PublicWord : Word;
    AsmWord    : ^Word;

begin
  PublicWord := $0815;
  AsmWord := @PrivatWord;
  if PublicWord <> AsmWord^ then exit
  else PublicWord := PublicFunction;
end.
```

Übrigens wurde im Assemblerteil das DS-Register nicht belegt, weil Pascal dies schon durch den Eintrittscode erledigt hat. Wenn die Routinen aufgerufen werden, so steht die korrekte Adresse schon in DS.

Sie werden fragen, wozu das gut sein soll, wenn man doch einfacher *PrivateWord* im Pascal-Teil definieren kann und in das Assemblermodul exportiert. Nun – in diesem Fall ist das sicherlich einfacher. Aber wenn Sie Daten im Code- oder einem anderen Segment unterbringen wollen und aus dem Pascal-Teil darauf zugreifen wollen? Dann geht es eben nur über diesen Umweg! Aber denken Sie daran: Dieser Trick funktioniert nur im Real-Mode des Prozessors, weil im Protected-Mode ein schreibender Zugriff auf das Codesegment verboten ist!

Lieber Leser, wir sind am Ende des Teils dieses Buches angekommen, der Ihnen das Arbeiten mit dem Assembler näherbringen sollte. Was nun noch folgt, ist eine ausführlich kommentierte Referenz der Assemblerbefehle und -anweisungen.

Ich hoffe, daß ich Ihnen zeigen konnte, daß der Assembler alles andere als ein schwer handhabbares Ungetüm aus grauer Vorzeit ist. Zugegebenermaßen unterscheidet sich die Art der Programmierung in Assembler erheblich von der in Hochsprachen: Man muß zu jedem Zeitpunkt genau wissen, welchen Inhalt welches Register des Prozessors und/oder Coprozessors hat. Es ist auch ungewohnt, Zahlen nicht als Ganzes zu betrachten, sondern sie in Bits zerlegt zu manipulieren.

Schließlich muß man sich auch daran gewöhnen, einfache Programmierbefehle erst aus Maschinencode zusammenzusetzen, bevor man sie sinnvoll in Programme einbinden kann. Aber auf der Kehrseite der Medaille steht hocheffizienter, kompakter und schneller Code – eine Voraussetzung für modernes Programmieren!

Es würde mich freuen, wenn ich Ihr Interesse für diese Art der Programmierung geweckt habe. In diesem Sinne wünsche ich Ihnen viel Spaß bei der Vertiefung Ihrer Assemblerkenntnisse und der Entwicklung eigener Assemblermodule.

Teil 3 Referenz

35 Prozessorbefehle

In der folgenden Referenz werden auch die Prozessorbefehle aufgelistet, die nur im Protected-Mode oder im Virtual-8086-Mode der Prozessoren 80286ff angewendet werden können. Allerdings wird auf eine detaillierte Besprechung verzichtet, da ihre Verwendung nicht Gegenstand dieses Buches ist. Bei Bedarf sei daher auf weiterführende Literatur verwiesen.

Bei der Beschreibung der Funktionen werden in den Tabellen folgende Abkürzungen benutzt:

O	*Overflow-Flag*
D	*Direction-Flag*
I	*Interrupt-Enable-Flag*
T	*Trap-Flag*
S	*Sign-Flag*
Z	*Zero-Flag*
A	*Auxiliary-Flag*
P	*Parity-Flag*
C	*Carry-Flag*

Bei den Flagangaben bezeichnet ein *?* ein nach der Operation nicht definiertes Flag, was bedeutet, daß aus der Flagstellung keine Rückschlüsse erlaubt sind. Ein * steht für ein durch den Befehl manipuliertes Flag, bei dem die Stellung vom Funktionsergebnis abhängig ist. *1* und *0* repräsentieren dementsprechend ein explizit gelöschtes oder gesetztes Flag. Wird ein Flag von der Operation nicht verändert, so wird dies durch einen Leer-Eintrag gekennzeichnet.

Für die Operanden der Befehle werden folgende Symbole verwendet:

r8	8-Bit-Register des Prozessors
r16	16-Bit-Register des Prozessors
r32	32-Bit-Register des Prozessors
r	allgemeines Register
sr	Segmentregister CS, DS, ES, FS, GS oder SS
m8	8-Bit-Speicherstelle
m16	16-Bit-Speicherstelle
m32	32-Bit-Speicherstelle
m48	48-Bit-Speicherstelle
i8	8-Bit-Konstante, direkt angegeben
i16	16-Bit-Konstante, direkt angegeben
i32	32-Bit-Konstante, direkt angegeben
a16	16-Bit-Adresse (Offset)

a16:a16 Adresse mit 16-Bit-Offset und 16-Bit-Segment
a32 32-Bit-Adresse (Offset)
a32:a16 Adresse mit 32-Bit-Offset und 16-Bit -Selektor

BVar Byte-Variable; 8 Bits
WVar Word-Variable; 16 Bits
DVar DoubleWord-Variable; 32 Bits.
QVar QuadWord-Variable; 64 Bits.

Darüber hinaus gibt es folgende Abkürzungen:

EA *Effective Address*, effektive Adresse. Dieser Begriff tritt nur bei den Taktangaben in Verbindung mit Prozessoren vom Typ 8086/8088 auf. Bei diesen muß, falls auf eine Speicherstelle zurückgegriffen werden soll, Zeit in Form von zusätzlichen Takten für die Berechnung der *effektiven Adresse* berücksichtigt werden. Je nach Art der Adressierung kommt es hierbei zu unterschiedlichen Werten. Bei 80286- bis Pentium-Prozessoren erfolgt die Adreßberechnung anders, weshalb bei diesen Prozessortypen keine zusätzliche Berechnungszeit anfällt.

Ein Zugriff auf eine Speicherstelle kann (unabhängig vom Prozessortyp) auf mehrere Arten erfolgen:

- durch direkte Angabe einer Adresse (meist nur des Offsetanteils). Dies wird als Adressierung mittels *Displacement* bezeichnet;
  Beispiele:
  mov ax, [WVar]
  mov bx, [WVar+4]

- durch indirekte Angabe einer Adresse über ein Basis- (BX, BP; *base*) oder ein Indexregister (DI, SI; *index*);
  Beispiele:
  mov ax, [BP]
  mov bx, [SI]

- durch gemischte indirekte und direkte Angabe einer Adresse über ein Basis- (BX, BP; *base*) oder Indexregister (DI, SI; *index*) und eines *Displacements*;
  Beispiele:
  mov ax, [BX+4]
  mov bx, Array[SI] ; ≡ OFFSET Array + [SI]

- durch indirekte Angabe einer Adresse über ein Basis- (BX, BP; *base*) und ein Indexregister (DI, SI; *index*);
  Beispiele:
  mov ax, [BX+DI]
  mov bx, [BP+SI]

- durch gemischte direkte und indirekte Adressierung mittels Basis- und Indexregister und eines Offsetanteils;

Beispiele:
mov ax, Array[BX+SI]
mov bx, [BP+DI+6]

Für die 8086/8088-Prozessoren können die zusätzlichen Takte für EA folgender Tabelle entnommen werden:

disp	*base/index*	*base/index +disp*	BP+DI / BX+SI	BP+SI / BX+DI	BP+DI+disp / BX+SI+disp	BP+SI+disp / BX+DI+disp
+6	+5	+9	+7	+8	+11	+12

m bei Taktangaben gibt die Anzahl der Komponenten des folgenden Befehls an. Hierzu zählen neben allen Bytes des Opcodes dieses folgenden Befehls auch eventuell vorhandene Konstanten (byteweise Zählung) und Adreßverschiebungen.

n bei Taktangaben steht für die Anzahl der Wiederholungen des Befehls.

Bei den Taktangaben sind weiterhin folgende Ergänzungen zu berücksichtigen:

- Segmentpräfixe (CS:, DS: etc.) schlagen mit zusätzlichen +2 Takten zu Buche.
- Bei 8086/8088-Prozessoren ist die effektive Adresse zu berücksichtigen (s.o.).
- Bei 8088-Prozessoren muß ein 16-Bit-Datenzugriff mittels zweier 8-Bit-Buszugriffe erfolgen. Daher sind die Taktraten für diesen Prozessor länger als beim 8086, jedoch in der Regel niemals doppelt so lang. Auf die detaillierte Angabe der Taktraten für 8088-Prozessoren wird jedoch aus Platzgründen verzichtet.
- Auch bei den 80286- bis Pentium-Prozessoren muß eine EA berücksichtigt werden, wenn alle drei möglichen Adressierungsarten (*base* + *index* + *displacement*) zusammen verwendet werden. In diesem Fall muß ein Takt zur Taktrate des Befehls addiert werden (Bei Pentium und 80486 wird dieser zusätzliche Takt *nicht immer* benötigt!).

Es werden auch die zum jeweiligen Befehl gehörenden Opcodes angegeben, die sich aus einem oder zwei Bytes zusammensetzen (von Befehlspräfixen oder Ausnahmen einmal abgesehen). Soweit Speicheradressen Teil des Befehls sind, werden diese in den sich anschließenden Bytes codiert. In diesem Fall besteht der Opcode aus zwei Bytes, wobei das zweite Byte wie folgt zusammengesetzt wird:

Bits	7	6	5	4	3	2	1	0
Byte2	mod		reg			r/m		

Hierbei bedeuten:

mod *Modus*; zwei Bits (Bit 7 und 6 des Bytes), die die Art der folgenden Adresse angeben:

00 Falls r/m = 110, wird ein direkter Speicheroperand (16-Bit-Adresse) verwendet. Andernfalls ist das *Displacement* 0 und indirekte Adressierung mittels *base, index* oder beidem wird benutzt.

01 Es folgt eine 8-Bit-Adresse, die vorzeichenbehaftet auf 16 Bit ausgedehnt wird. Es werden *zwei* Bytes Adresse angegeben.

10 Es folgt eine vollständige 16-Bit-Adresse mit der üblichen Intel-Konvention: zunächst das *lo-byte*, dann das *hi-byte*.

11 Beide Operanden sind Register. In diesem Fall bezeichnet *reg* das Ziel (Operand 1) und *r/m* die Quelle (Operand 2).

reg *Register*; drei Bits (Bit 5, 4 und 3 des Bytes), die in Abhängigkeit von der Operandengröße das verwendete Register angeben:

000 AL/AX/EAX
001 CL/CX/ECX
010 DL/DX/EDX
011 BL/BX/EBX
100 AH/SP/ESP
101 CH/BP/EBP
110 DH/SI/ESI
111 BH/DI/EDI

r/m *Register/Memory*; drei Bits (Bit 2, 1 und 0 des Bytes), die Aufschluß über die Art der Adressierung ermöglichen:

000 EA = DS:[BX + SI + 16-Bit-*disp*]
001 EA = DS:[BX + DI + 16-Bit-*disp*]
010 EA = SS:[BP + SI + 16-Bit-*disp*]
011 EA = SS:[BP + DI + 16-Bit-*disp*]
100 EA = DS:[SI + 16-Bit-*disp*]
101 EA = DS:[DI + 16-Bit-*disp*]
110 EA = SS:[BP + 16-Bit-*disp*]. *Ausnahme*: *mod* = 00 → EA = 16-Bit-Adresse
111 EA = DS:[BX + 16-Bit-*disp*]

Falls *mod* = 11, ist *r/m* wie *reg* zu interpretieren, wobei *r/m* das Quell- und *reg* das Zielregister codiert.

In Verbindung mit der Adressierung von Operanden durch das *mod reg r/m*-Byte sind noch folgende Abkürzungen zu erwähnen:

/r Dieser Eintrag im *mod reg r/m*-Byte eines Opcodes steht für den Hinweis, daß das *mod reg r/m*-Byte nur ein Register adressieren kann.

/m Analog zu /r kann bei diesem Befehl nur ein Speicheroperand im *mod reg r/m*-Byte Verwendung finden.

/rm Bei dieser Angabe kann das *mod reg r/m*-Beyte entweder einen Speicheroperanden oder ein Register aufnehmen.

/0 .. /7 Angabe über die Bit-Stellung im *reg*-Feld des *mod reg r/m*-Bytes eines Prozessorbefehls. Dies bedeutet, daß das *reg*-Feld nicht zur Adressierung verwendet werden kann. Ob der in *r/m* spezifizierte Wert ein Register oder einen Speicher-

operanden codiert, entscheidet in diesem Fall *mod*: Bei *mod* = 11 spezifiziert *r/m* ein Register gemäß den *reg*-Daten, andernfalls ist ein Speicheroperand gemeint.

Die Bits des *reg*-Feldes codieren in diesen Fällen vielmehr zusätzlich zu dem eigentlichen, vorangehenden Code-Byte den Prozessorbefehl. So ist z.B. das erste Byte im Opcode für die Befehle ADC und ADD gleich: $80, wenn eine 8-Bit-Konstante verwendet wird. Durch unterschiedlich gesetzte Bits im *reg*-Feld des *mod reg r/m*-Bytes, was ja zusätzlich noch für die Adressierung zuständig ist, können die Befehle unterschieden werden. Bei ADC ist *reg* = 2, bei ADD 0. Dies wird im Opcode durch die Symbole /0 und /2 repräsentiert. Achtung: /0 heißt nicht, daß das *mod reg r/m*-Byte = 0 ist, sondern lediglich, daß in diesem Byte das *reg*-Feld = 0 ist.

Die Prozessoren ab dem 80386 besitzen erweiterte Fähigkeiten der Adressierung von Speicherstellen. Benutzt man die 32-Bit-Adressierung, so kommen neben den eben besprochenen Möglichkeiten der direkten, indirekten und gemischten Adressierung noch einige hinzu, die mit der »normalen« *mod-reg-r/m*-Kodierung nicht möglich sind. Aus diesem Grunde kann bei der Verwendung von 32-Bit-Registern und/oder 32-Bit-Adressierung ein zusätzliches Byte im Opcode notwendig werden, das die erweiterten Adressierungsarten ermöglicht.

EA der Zugriff auf eine Speicherstelle kann analog zur 16-Bit-Adressenberechnung auf verschiedene Weise erfolgen:

- durch direkte Angabe einer vollständigen (32-Bit-)Adresse. Dies wird wie bei der 16-Bit-Adressierung als Adressierung mittels *Displacement* bezeichnet (beachten Sie bitte, daß die Adresse der Variablen 32 Bit umfaßt, nicht etwa ihr Inhalt, wie man an WVar im folgenden Beispiel sehen kann);
  Beispiele:
  mov eax, [Wvar]
  mov ebx, [WVar+4]

- durch indirekte Angabe einer Adresse über ein Basis- (base) oder ein Indexregister (index). Bitte beachten Sie, daß bis auf ESP jedes der 32-Bit-Register hierzu verwendet werden kann. Beachten Sie bitte auch, daß mit den Faktoren 2, 4 oder 8 eine Skalierung des Index, nicht aber der Basis möglich ist;
  Beispiele:
  mov eax, [EAX]
  *mov ebx, [EDI*4]*

- durch gemischte indirekte und direkte Angabe einer Adresse über ein Basis- (base) oder Indexregister (index) und eines (8- oder 32-Bit-) Displacements. Auch in diesem Fall ist eine Skalierung möglich;
  Beispiele:
  mov eax, [EBX+4]
  *mov ebx, Array[ESI*2]* ; ≡ OFFSET Array + [ESI*2]

- durch indirekte Angabe einer Adresse über ein Basis- (base) und ein Indexregister (index);
  Beispiele:
  mov eax, [EAX+EBX] ; EAX fungiert als Basis, EBX als Index!
  *mov ebx, [EBP+ESI*8]* ; Indizes sind skalierbar!

- durch gemischte direkte und indirekte Adressierung mittels Basis- und Indexregister und eines Offsetanteils (= 8- oder 32-Bit-Displacement);
  Beispiele:
  *mov eax, Array[EBP+ESI*2]* ; Array =*Offset*, EBP = *base*, ESI = *index*
  *mov ebx, [EDX+EDX*4+4]*

Es ist klar, daß diese Erweiterungen nicht möglich sind, ohne an der »alten« *mod-reg-r/m*-Philosophie etwas zu ändern. Konsequenz: nötigenfalls Einführung eines *sib*-Feldes im Opcode, d.h. eines weiteren Bytes nach dem *mod-reg-r/m*-Byte. Dieses Byte wird immer dann notwendig, wenn ein Index mit zur Adressenberechnung herangezogen wird. Andernfalls entfällt es, es bleibt dann alles beim alten, wobei sich jedoch die Bedeutung des *mod-reg-r/m*-Feldes geändert hat:

Bits	7	6	5	4	3	2	1	0
Byte2	**mod**		**reg**			**r/m**		
Byte3	s		i			b		

Hierbei bedeuten wie bisher, nun aber mit unterschiedlicher Bedeutung:

mod Modus; zwei Bits (Bit 7 und 6 des Bytes), die die Art der folgenden Adresse angeben:

- 00 Die Adresse steht in einem der 32-Bit-Register oder wird als direkter 32-Bit-Wert hinter den Opcode geschrieben.
- 01 Die Adresse berechnet sich aus einer Basis, die in einem 32-Bit-Register steht, sowie einem 8-Bit-Displacement.
- 10 Die Adresse wird aus einer Basis in einem Register und einem 32-Bit-Displacement berechnet.
- 11 Beide Operanden sind Register. In diesem Fall bezeichnet *reg* das Ziel (Operand 1) und *r/m* die Quelle (Operand 2).

Diese Definition gilt für alle r/m-Werte mit einer Ausnahme: r/m = 100b. In diesem Fall wird ein zusätzliches sib-Byte nötig, das dem mod-reg-r/m-Byte folgt und einen Index angibt.

reg Register; drei Bits (Bit 5, 4 und 3 des Bytes), die in Abhängigkeit von der Operandengröße das verwendete Register angeben:

- 000 AL/AX/EAX
- 001 CL/CX/ECX
- 010 DL/DX/EDX
- 011 BL/BX/EBX

100 AH/SP/ESP
101 CH/BP/EBP
110 DH/SI/ESI
111 BH/DI/EDI

r/m Register/Memory; drei Bits (Bit 2, 1 und 0 des Bytes), die Aufschluß über die verwendete Basis geben:

000 [EAX]
001 [ECX]
010 [EDX]
011 [EBX]
100 sib-Byte notwendig wegen zusätzlichem Index.
101 [EBP]. Ausnahme: mod = 00 → EA = 32-Bit-Adresse
110 [ESI]
111 [EDI]

Falls *mod* = 11, ist *r/m* wie *reg* zu interpretieren, wobei *r/m* das Quell- und *reg* das Zielregister codiert.

s scale; zwei Bits (Bit 7 und 6 des Bytes), die einen Skalierungsfaktor angeben:

00 Faktor 0 → Skalierung mit 1
01 Faktor 1 → Skalierung mit 2
10 Faktor 2 → Skalierung mit 4
11 Faktor 3 → Skalierung mit 8

i index; drei Bits (Bit 5, 4 und 3 des Bytes), die das 32-Bit-Register angeben, das den Index enthält:

000 EAX
001 ECX
010 EDX
011 EBX
100 kein Index
101 EBP
110 ESI
111 EDI

b base, drei Bits (Bit 2, 1 und 0 des Bytes), die Aufschluß über die verwendete Basis geben:

000 [EAX]
001 [ECX]
010 [EDX]
011 [EBX]
100 Sonderfall
101 [EBP]
110 [ESI]
111 [EDI]

Sonderfall: Falls *b* = 100b ist, hängt die verwendete Basis vom *mod*-Feld ab. Ist *mod* = 00b, so wird ein 32-Bit-Displacement ohne Basis verwendet, bei allen anderen Werten von *mod* ist [EBP] die Basis. Dann entscheidet *mod* = 01b, daß dem *sib*-Byte ein 8-Bit-Displacement folgt, das intern auf 32-Bit erweitert wird, bzw. bei *mod* = 10b ein »echtes« 32-Bit-Displacement. *Mod* = 11b definiert, daß beide Operanden 32-Bit-Register sind.

Das klingt alles sehr kompliziert, ist es aber nicht. Untersuchen Sie, falls Sie aus einer Adresse tatsächlich »von Hand« die Informationen extrahieren wollen, zunächst das *mod*-Feld aus dem zweiten Byte des Opcodes (dem *mod-reg-r/m*-Byte). Hat dies den Inhalt 11b, so ist der Fall klar: Beide Operanden sind 32-Bit-Register, es kann kein *sib*-Byte folgen! Welche Register nun verwendet werden, codieren in identischer Weise die Felder *reg* und *r/m*. Konsultieren Sie dann die unter der Erläuterung von *reg* gezeigte Tabelle.

Bei allen anderen Werten für *mod* muß eine EA berechnet werden. Einfachster Fall, weil Ausnahme: *mod* = 00b und *r/m* = 101b. Dann wird als Adresse eine vollständige, echte 32-Bit-Adresse ohne Basis und Index verwendet. Andernfalls gibt es eine Basis, eventuell einen Index, und es muß gegebenenfalls ein *Displacement* berücksichtigt werden. Betrachten Sie nun das *r/m*-Feld. Hat dieses Feld *nicht* den Inhalt 100b, so ist die Adresse *nicht* indiziert (hat also keinen Index neben einer Basis) und es gibt kein *sib*-Byte. In diesem Falle enthält ein Register, das durch *r/m* genauer spezifiziert wird (vgl. Tabelle unter der Erläuterung von *r/m*), eine Basis. Zu dieser muß entweder kein *Displacement* (*mod* = 00b) addiert werden oder aber ein 8-Bit- (*mod* = 01b) oder 32-Bit-Displacement (*mod* = 10b). Diese *Displacements* folgen unmittelbar auf das zweite Byte des Opcodes und bestehen aus einem Byte (*mod* = 01b; es wird intern auf 32 Bit erweitert) oder aus vier Bytes (*mod* = 10b).

Hat dagegen das *r/m*-Feld den Inhalt 100b, so ist die Adresse indiziert und es folgt ein *sib*-Byte (Byte 3 des Opcodes). Dieses *sib*-Byte beinhaltet dann die Basis, den Index und den Skalierungsfaktor. Nun kann *mod* untersucht werden: *mod* = 00b zeigt, daß kein *Displacement* berücksichtigt werden muß, *mod* = 01b verweist auf ein 8-Bit-Displacement und *mod* = 10b auf ein 32-Bit-Displacement, analog zum obigen Fall ohne Index. Auch hier folgt dann das *Displacement*, falls vorhanden, auf das *sib*-Byte, und zwar ganz analog zu oben. *Mod* = 11b war ja die schon besprochene Ausnahme, daß beide Operanden Register sind. Das *sib*-Byte enthält nun in *s* den Skalierungsfaktor, genauer die Potenz zur Basis 2, die den Skalierungsfaktor bildet; *i* codiert das Indexregister und *b* das Basisregister. Einzige Ausnahme hier: wenn *b* den Wert 101b enthält und *mod* = 00b ist, liegt kein Basisregister vor, sondern eine echte 32-Bit-Adresse. Ist *mod* nicht 00b, so zeigt dies EBP als Basis an.

In der folgenden Referenz werden zu den einzelnen Befehlen unter anderem Hinweise zur Verwendung, zu Ausführungsgeschwindigkeiten und zu den korrespondierenden Opcodes gegeben. Hierbei können in Abhängigkeit von den notwendigen Parametern des Befehls unter *Verwendung* unterschiedliche Angaben erfolgen. Sie werden tabellarisch aufgeführt und in der Spalte # mit Nummern versehen. Die korrespondierenden Ausführungszeiten und Opcodes werden dann unter *Takte* und *Opcodes* ebenfalls tabellarisch unter den entsprechenden Nummern angegeben.

In der Referenz werden Sie keine Taktangaben für den Pentium Pro finden! Der Grund hierfür ist, daß keine exakten Angaben gemacht werden können. Aufgrund der Architektur des Pentium Pro (»*three-way superscalar, pipelined architecture*«) kommen bei der Ausführung der Befehle Techniken der parallelen Bearbeitung zum Tragen, mit denen der Prozessor im Durchschnitt drei Instruktionen pro Takt decodieren, *dispatchen* und ausführen kann.

Bei einigen Befehlen können durch falsche Operanden oder andere Bedingungen Ausnahmesituationen auftreten, sogenannte Exception*s*. Bei der Besprechung der einzelnen Befehle sind die bei desem Befehl möglichen Exceptions in Abhängigkeit vom Betriebssmodus aufgelistet. So werden neben der Art der Exception auch deren mögliche Ursachen in Form einer Codezahl angegeben, die Sie anhand folgender Auflistung decodieren können:

#AC

1 Der *Alignment-Check* ist aktiviert, der CPL ist 3, und es wird ein nicht ausgerichtetes Datum vorgefunden.

#BR

1 Das Testergebnis liegt außerhalb der im Operanden vorgegebenen Grenzen.

#DB

1 Das GD-Flag in Debugregister DR7 ist gesetzt, und es wird auf ein Debugregister zugegriffen.

#DE

1 Der Operand hat den Wert $00.

2 Der Divisor (Quelloperand) hat den Wert $00.

3 Das Ergebnis ist im Format des Zieloperanden nicht darstellbar.

#GP

Allgemein:

1 Der Protected-Mode ist nicht aktiv.

2 Der Befehl kann im Virtual-8086-Mode nicht ausgeführt werden.

Nullselektoren

3 Auf das DS-, ES-, FS- oder GS-Register wird zugegriffen, der enthaltene Selektor ist aber ein Nullselektor.

4 Der im Zieloperanden, Interrupt-Gate, Trap-Gate, Call-Gate, Task-Gate oder Task-State-Segment verzeichnete Selektor ist ein Nullselektor.

5 Der im Gate verzeichnete Selektor für das Codesegment ist ein Nullselektor.

6 Der in das SS-Register zu ladende Selektor ist ein Nullselektor.

7 Die Ziel-/Rücksprungadresse ist Null, oder der Selektor für das Stacksegment ist ein Nullselektor.

Grenzüberschreitungen

8 Die verwendete Speicheradresse liegt außerhalb des durch den Selektor in CS, DS, ES, FS oder GS ausgewählten Segments.

9 Die verwendete Speicheradresse liegt außerhalb des adressierbaren Bereichs von $0000 bis $FFFF.

10 Die Rücksprungadresse bzw. die Zieladresse im Operanden, Call-Gate oder Task-Switch-State liegt nicht im anzuspringenden Codesegment.

11 Die Zieladresse im Codesegment liegt außerhalb des adressierbaren Bereichs von $0000 bis $FFFF für das Codesegment, weil ein ADRSIZE-Präfix verwendet wurde.

12 Der Selektor für das Codesegment, das Gate oder den Task-Switch-State zeigt nicht in eine Deskriptortabelle.

13 Der Selektor für das Codesegment oder Stacksegment für den Rücksprung zeigt nicht in eine Deskriptortabelle.

14 Der Selektor zeigt auf einen Eintrag außerhalb der verwendeten Tabelle.

15 Der Selektor zeigt auf eine LDT oder eine Stelle außerhalb der GDT.

16 Eine Interruptnummer zeigt auf einen Vektor außerhalb der IDT.

17 Die Einsprungadresse in der IDT, im Interrupt-Gate, Trap-Gate oder Task-Gate liegt nicht in einem Codesegment.

18 Das EBP-Register zeigt auf eine Adresse, die außerhalb des Stack-Segments liegt.

19 Das EBP-Register zeigt auf eine Adresse, die außerhalb des adressierbaren Bereichs von $0000 bis $FFFF liegt.

Falsche Segmente

20 Das Ziel der Operation liegt in einem als non-writable markierten Segment.

21 Der Selektor, der in das SS-Register geladen werden soll, zeigt auf ein als non-writable markiertes Segment.

22 Der Selektor, der in das DS-, ES-, FS- oder GS-Register geladen werden soll, zeigt nicht auf ein als readable markiertes Code- oder Datensegment.

23 Der Selektor im Operanden zeigt nicht auf ein conforming oder non-conforming Codesegment, auf ein Call-Gate, Task-Gate oder ein Task-Switch segment.

24 Der Selektor eines Call-Gates, Interrupt-Gates oder Trap-Gates zeigt nicht auf ein Codesegment.

25 Der Selektor im Operanden zeigt nicht auf einen GDT-Eintrag, oder der selektierte GDT-Eintrag zeigt nicht auf eine LDT.

26 Der Selektor in einem Task-Gate zeigt auf ein Task-Switch segment, das als not present markiert ist.

27 Das Stacksegment ist kein als writable markiertes Datensegment.

28 Der Deskriptor eines Codesegments enthält nicht die Typinformation für ein Codesegment.

29 Der Deskriptor, auf den der Selektor der Rücksprungadresse zeigt, beschreibt kein Codesegment.

30 Der Deskriptor in der IDT beschreibt keine Interrupt-Gate, Trap-Gate oder Task-Gate.

Privilegverletzungen

31 Der IOPL ist kleiner als 3, oder der DPL des Interrupt-, Trap- oder Task-Gate-Deskriptors ist nicht gleich 3.

32 Der CPL ist größer als 0.

33 Der CPL ist größer als der IOPL (I/O-Privileg-Level) des aktuellen Programms oder der aktuellen Routine

34 Der CPL ist größer als der IOPL (I/O-Privileg-Level), und irgendeines der Permission-Bits im TSS für den angesprochenen Port ist gesetzt.

35 Der DPL im anzuspringenden non-conforming Codesegment ist nicht gleich dem CPL, oder der RPL des Selektors in der Ziel- oder Rücksprungadresse ist größer als der CPL.

36 Der DPL im anzuspringenden conforming Codesegment ist größer als der CPL.

37 Der DPL im anzuspringenden conforming Codesegment ist größer als der RPL im Selektor der Rücksprungadresse.

38 Der DPL des Codesegments, auf das ein Call-Gate zeigt, ist größer als der CPL.

39 Der DPL des Stacksegments ist nicht gleich dem RPL im Selektor der Rücksprungadresse.

40 Der DPL eines Call-Gates, Task-Gates oder eines Task-Switch-State ist kleiner als der CPL oder als der RPL des betreffenden Selektors.

41 Der DPL ist kleiner als der CPL und der RPL des Selektors, der in das DS-, ES- FS- oder GS-Register geladen wird. Dies ist eine Schutzverletzung, obwohl der Selektor auf ein Daten- oder non-conforming Codesegment zeigt.

42 Der DPL des Interrupt-Gates, Trap-Gates oder Task-Gates, das durch Auslösen des INT n-Befehls (auch INT3 und INTO) verwendet werden soll, ist kleiner als der CPL.

43 Der RPL des Selektors in der Ziel- oder Rücksprungadresse ist größer als der CPL.

44 Der RPL des Selektors der Rücksprungadresse ist kleiner als das CPL.

45 Der RPL im Selektor des Stacksegments und der Rücksprungadresse sind verschieden.

46 Das SS-Register wird mit einem Selektor geladen, dessen RPL nicht gleich dem DPL des Segments und dem CPL ist.

Falsche Bitstellungen

47 Das Global/local-Flag im Selektor eines Task-Switch-States steht auf »local«.

48 Ein Permission-Bit im Task-Switch-State für den angesprochenen Port ist gesetzt.

49 Versuch, eine ungültige Bitkombination in CR0 zu schreiben (z.B. PG = 1 und PE = 0 oder CD = 0 und NE = 1).

50 Versuch, ein Bit im Kontrollregister CR4 zu setzen.

51 Versuch, in der Pointer-Tabelle eines Page-Directory-Eintrages reservierte Bits für die Nutzung der Physical-Address-Extension zu setzen, wenn PAE in Kontrollregister CR4 und PG in Kontrollregister CR0 gesetzt sind.

52 Der CPL ist größer als 0, und das PCE-Flag in Kontrollregister CR4 ist gelöscht.

53 Das PCE-Flag in Kontrollregister CR4 ist gelöscht.

54 Der CPL ist größer als 0, und das TSD-Flag in Kontrollregister CR4 ist gesetzt.

55 Das TDS-Flag in Kontrollregister CR4 ist gesetzt.

Ungültige Operanden/Befehle

56 Auf die Debugregister kann in dieser Betriebsart nicht zugegriffen werden.

57 POPF/POPFD wurde mit einem OPSIZE-Präfix versehen.

58 Das SP-bzw. ESP-Register enthält den Wert 7, 9, 11, 13, oder 15.

59 Der Inhalt von ECX codiert ein reserviertes oder nicht implementiertes modellspezifisches Register (MSR).

60 Der Wert in ECX ist nicht 0 oder 1.

61 Das modellspezifische Register (MSR) SYSENTER_CS_MSR enthält den Wert 0.

Inkonsistenzen

62 Beim Laden des SS-Registers wird eine Fehlerursache vorgefunden: Der Selektor zeigt nicht in die GDT, der RPL des Selektors ist nicht gleich dem CPL, das als Stack verwendete Datensegment ist als non-writable markiert, oder der DPL des Segments ist nicht gleich dem CPL.

63 Beim Laden des DS-, ES-, FS- oder GS-Registers wird folgende Fehlerursache vorgefunden: Der Selektor ist ein Nullselektor oder zeigt nicht in die GDT, das Segment ist kein Daten- oder readable Codesegment, oder das Segment ist zwar ein Daten- oder non-conforming Codesegment, aber sowohl der RPL als auch der CPL sind größer als der DPL.

64 Der Deskriptor weist seinen Task-Switch-State als busy oder nicht verfügbar aus.

65 Der Selektor im Quelloperanden zeigt auf ein Segment, das kein Task-Switch-State (TSS) ist oder aber auf ein TSS, das zu einem gerade aktiven Task gehört.

66 Das Codesegment der Rücksprungadresse ist non-conforming, und sein DPL ist nicht gleich dem RPL des Selektors für das Codesegment.

67 Das Codesegment der Rücksprungadresse ist conforming, und sein DPL ist größer als der RPL des Selektors für das Codesegment.

#MF

1 Eine FPU-Exception wartet auf die Behandlung.

#NM

1 Das TS-Flag in Kontrollregister CR0 ist gesetzt.

2 Das EM-Flag oder das TS-Flag in Kontrollregister CR0 ist gesetzt.

3 Das MP-Flag und das TS-Flag in Kontrollregister CR0 sind gesetzt.

#NP

1 Ein Codesegment, Datensegment, Stacksegment oder ein Call-Gate, Task-Gate oder Task-State-Segment (TSS) sind als not present markiert.

2 Der LDT-Deskriptor in der GDT ist als not present markiert.

3 Das Task-State-Segment ist als not present markiert.

4 Der nach DS, ES, FS oder GS zu ladende Selektor zeigt auf ein Segment, das als not present markiert ist.

#PF

1 Beim Versuch, auf eine Page zuzugreifen, wurde ein Fehler entdeckt.

#SS

1 Die verwendete Speicheradresse liegt außerhalb des durch den Selektor in SS ausgewählten Segments.

2 Der neu geladene Wert im SS- oder ESP-Register weist auf Adressen, die außerhalb der Stacksegmentgrenzen liegen.

3 Das Ablegen von Daten auf dem Stack sprengt die Grenzen des Stacksegments.

4 Das Ablegen einer Rücksprungadresse und/oder Flags und/oder eines Fehlercodes sprengt die Grenzen des Stacksegments.

5 Das Ablegen einer Rücksprungadresse und/oder Parameter und/oder Stackpointer auf den Stack, ohne daß ein Task-Switch stattfindet, führt zum Überschreiten der Grenzen des Stacksegments.

6 Das Ablegen einer Rücksprungadresse und/oder Parameter und/oder Stackpointer auf den Stack bei einem Task-Switch führt zum Überschreiten der Grenzen des Stacksegments.

7 Das SS-Register wird mit einem Selektor geladen, dessen dazugehöriges Segment als not present markiert ist.

8 Das SS-Register wird im Rahmen eines Task-Switchs mit einem Selektor geladen, dessen dazugehöriges Segment als not present markiert ist.

9 Die Stackspitze liegt außerhalb der Segmentgrenzen des Stacksegments.

10 Stackspitze und StackByteasis liegen außerhalb der Segmentgrenzen des Stacksegments.

#TS

1 Der Selektor für das Stacksegment ist ein Nullselektor.

2 SS:ESP liegt außerhalb des TSS.

3 Der RPL des im TSS verzeichneten Selektors für das Stacksegment ist nicht gleich dem DPL des Codesegments, auf das umgeschaltet wird.

4 Der DPL des neuen Stacksegments, auf das der Selektor im TSS zeigt, ist nicht gleich dem DPL des neuen Codesegments.

5 Das neue Stacksegment nicht kein als writable markiertes Datensegment.

6 Der Selektor für ein Stacksegment zeigt nicht in eine Deskriptortabelle.

#DU

1 Der Befehl ist in diesem Modus nicht verfügbar.

2 Der Quelloperand ist keine Speicherstelle.

3 Der Zieloperand ist kein Register.

4 Der Zieloperand ist ein Register.

5 Der Zieloperand ist keine Speicherstelle.

6 Das EM-Flag in Kontrollregister CR0 ist gesetzt.

7 Das Präfix kann nicht in Verbindung mit dem sich anschließenden Befehl eingesetzt werden.

8 Das CS-Register soll geladen werden.

9 Das DE-Flag in Kontrollregister CR4 ist gesetzt, und ein Operand bezeichnet die Debugregister DR4 oder DR5.

10 Der Prozessor ist bei Ausführung des Befehls nicht im System-Management-Mode (SMM).

11 Die Exception wird mit Sicherheit ausgeführt.

AAA 8086

Funktion Dieser Befehl dient zur Korrektur des Ergebnisses einer Addition mittels ADD, falls die beiden addierten Werte gültige ungepackte BCDs waren.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				?	?		*		?		*

Nach dem Befehl sind Overflow-, Sign-, Zero- und Parity-Flag nicht definiert. Das Auxiliary- und Carry-Flag werden bei einem Überlauf mittels BCD-Arithmetik gesetzt, also falls die Binärzahl korrigiert werden mußte, um eine gültige BCD zu erhalten.

Verwendung

#	Parameter	Beispiel
	keine	AAA

Arbeitsweise AAA prüft zunächst, ob das Auxiliary-Flag (durch die vorhergehende Addition) gesetzt wurde. Ist dies der Fall oder ist das untere Nibble (die Bits 3 bis 0) des Wertes in AL größer als 9, so wird zum Wert in AL 6 addiert und das obere Nibblein AL (Bits 7 bis 4) gelöscht. Der Wert in AH wird um eins erhöht, und das Carry- und Auxiliary-Flag werden gesetzt. Andernfalls werden nur die beiden Flags gelöscht, Korrekturmaßnahmen unterbleiben!

Takte

#	8086	80286	80386	80486	Pentium
	8	3	4	3	3

Opcodes

#	B1
	37

Exceptions Keine

Bemerkungen Mit diesem Befehl können nur ungepackte BCDs bearbeitet werden. Für gepackte BCDs steht der Befehl DAA zur Verfügung. AAA arbeitet nur *nach* einer Addition korrekt, da er das Auxiliary-Flag auswertet, mit dem ein Dezimalüberlauf signalisiert wird. Für Subtraktionen existiert der Befehl AAS, für Multiplikationen AAM und für Divisionen AAD.

Beschreibung Seite 73

AAD *8086*

Funktion Dieser Befehl dient zur Vorbereitung einer Division mittels DIV, falls die beiden zu dividierenden Werte gültige BCDs sind.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				*	*		?		*		?

Nach dem Befehl sind das Overflow-, Auxiliary- und Carry-Flag nicht definiert. Das Sign-, Zero- und Parity-Flag werden in Abhängigkeit vom Inhalt von AX gesetzt.

Verwendung

#	Parameter	Beispiel
	keine	AAD

Arbeitsweise AAD führt eine Operation nach der Formel AL = AL + 10 · AH durch. Nach AAD wird AH auf 0 gesetzt.

Takte

#	8086	80286	80386	80486	Pentium
	60	14	19	14	10

Opcodes

#	B1	B2
	D5	0A

Exceptions Keine

Bemerkungen Mit diesem Befehl können nur ungepackte BCDs bearbeitet werden. AAD arbeitet nur *vor* einer Division korrekt. Für Subtraktionen existiert der Befehl AAS, für Multiplikationen AAM und für Additionen AAA. AAD kann dazu verwendet werden, zwei ungepackte BCDs zu packen.

Tip Das zweite Byte im Opcode ($0A) repräsentiert den Wert, den der Befehl AAD bei der Berechnung verwendet (10; siehe oben). Zwar kann der Assembler nicht angewiesen werden, AAD anders als mit dem Opcode D5-0A zu übersetzen, da AAD keinen Operanden hat. Dennoch kann die Eingabe über die Angabe von »data bytes« erzwungen werden. Setzt man z.B im Quellcode an die Stelle, an der ein AAD 8 erwünscht ist, die Folge DB 0D5h, 008h, so assembliert der Assembler diese Bytes in die gewünschte Befehlssequenz.

Achtung: Der Prozessor arbeitet dann tatsächlich in der gewünschten Wiese, d.h. er bildet in diesem Falle AL = AL + 8 · AH. Jedoch kann es bei der Betrachtung des Codes oder bei der schrittweisen Ausführung mit Debuggern zu seltsamen Anzeigen kommen, da die Debugger nicht auf diesen legitimen, jedoch nicht beabsichtigten Mißbrauch des Befehls AAM vorbe

reitet sind. Dies führt zu keinerlei Fehlfunktionen des Prozessors; Sie können solche befremdlichen Codesequenzen gefahrlos ignorieren! Lediglich der Debuggerteil, der die Codes disassembliert und auf dem Bildschirm in lesbarer Weise darstellt, ist hierbei überfordert.

Beschreibung Seite 76

AAM *8086*

Funktion Dieser Befehl dient zur Korrektur einer Multiplikation zweier ungepackter BCDs mittels MUL.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				*	*		?		*		?

Nach dem Befehl sind das Overflow-, Auxiliary- und Carry-Flag nicht definiert. Das Sign-, Zero- und Parity-Flag werden in Abhängigkeit vom Inhalt von AX gesetzt.

Verwendung

#	Parameter	Beispiel
	keine	AAM

Arbeitsweise AAM führt eine Operation nach der Formel AH = AL DIV 10 und AL = AL MOD 10 durch.

Takte

#	8086	80286	80386	80486	Pentium
	83	16	17	15	18

Opcodes

#	B1	B2
	D4	0A

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#DE	-	15, 23, 24, 26, 35	-	1	1

Bemerkungen Mit diesem Befehl können nur ungepackte BCDs bearbeitet werden. AAM arbeitet nur *nach* einer Multiplikation korrekt. Für Subtraktionen existiert der Befehl AAS, für Divisionen AAD und für Additionen AAA.

AAM kann dazu verwendet werden, eine gepackte BCD zu entpacken.

Hinweis Mit AAM läßt sich sehr einfach eine hexadezimal codierte Zahl in ASCII-Ziffern zerlegen. Befindet sich nämlich in AX eine Zahl, deren Wert $63 (=99) nicht überschreitet, so führt die Anwendung von AAM zu einer

Trennung der beiden Hexadezimalziffern in AH und AL, das Ergebnis ist dezimal codiert. Die anschließende ODER-Verknüpfung mit $3030 stellt dann die ASCII-Repräsentation der beiden Ziffern in AH und AL dar:

```
mov ax, Zahl                ; Zahl ist hexadezimal
aam
or  ax,03030h               ; zwei ASCII-Ziffern
```

Tip

Das zweite Byte im Opcode ($0A) repräsentiert den Wert, den der Befehl AAM bei der Berechnung verwendet (10; siehe oben). Zwar kann der Assembler nicht angewiesen werden, AAM anders als mit dem Opcode D4-0A zu übersetzen, da AAM keinen Operanden hat. Dennoch kann die Eingabe über die Angabe von »data bytes« erzwungen werden. Setzt man z.B. im Quellcode an die Stelle, an der ein AAM 16 erwünscht ist, die Folge DB 0D4h, 010h, so assembliert der Assembler diese Bytes in die gewünschte Befehlssequenz.

Achtung: Der Prozessor arbeitet dann tatsächlich in der gewünschten Wiese, d.h. er bildet in diesem Falle AH = AL DIV 16 und AL = AL MOD 16. Jedoch kann es bei der Betrachtung des Codes oder bei der schrittweisen Ausführung mit Debuggern zu seltsamen Anzeigen kommen, da die Debugger nicht auf diesen legitimen, jedoch nicht beabsichtigten Mißbrauch des Befehls AAM vorbereitet sind. Dies führt zu keinerlei Fehlfunktionen des Prozessors; Sie können solche befremdlichen Codesequenzen gefahrlos ignorieren! Lediglich der Debuggerteil, der die Codes disassembliert und auf dem Bildschirm in lesbarer Weise darstellt, ist hierbei überfordert.

Beschreibung Seite 75

AAS *8086*

Funktion

Dieser Befehl dient zur Korrektur einer Subtraktion zweier ungepackter BCDs mittels SUB.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				?	?		*		?		*

Nach dem Befehl sind das Overflow-, Sign-, Zero- und Parity-Flag nicht definiert. Das Auxiliary- und Carry-Flag werden bei einem Überlauf mittels BCD-Arithmetik gesetzt, also falls die Binärzahl korrigiert werden mußte, um eine gültige BCD zu erhalten.

Verwendung

#	Parameter	Beispiel
	keine	AAS

Arbeitsweise AAS prüft zunächst, ob das Auxiliary-Flag (durch die vorhergehende Subtraktion) gesetzt wurde. Ist dies der Fall oder ist das untere Nibble (die Bits 3 bis 0) des Wertes in AL größer als 9, so wird vom Wert in AL 6 subtrahiert und das obere Nibble in AL (Bits 7 bis 4) gelöscht. Der Wert in AH wird um eins verringert und das Carry- und Auxiliary-Flag werden gesetzt. Andernfalls werden nur die beiden Flags gelöscht, Korrekturmaßnahmen unterbleiben.

Takte

#	8086	80286	80386	80486	Pentium
	8	3	4	3	3

Opcodes

#	B1
	3F

Exceptions Keine

Bemerkungen Mit diesem Befehl können nur ungepackte BCDs bearbeitet werden. Für gepackte BCDs steht der Befehl DAS zur Verfügung. AAS arbeitet nur *nach* einer Subtraktion korrekt. Für Divisionen existiert der Befehl AAD, für Multiplikationen AAM und für Additionen AAA.

AAS kann dazu benutzt werden, Hexadezimalziffern als Dezimalziffern darzustellen. Falls AL *eine* hexadezimale Ziffer enthält, findet sich nach AAS in AH/AL deren dezimale Repräsentation.

Durch Verküpfung der so erhaltenen Dezimalziffern mittels OR AX,$3030 lassen sich die ASCII-Zeichen der Ziffern ermitteln.

Beschreibung Seite 75

ADC *8086*

Funktion ADC (**ad**d with **c**arry) führt eine Addition durch, wobei der Zustand des Carry-Flags berücksichtigt wird.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Addition gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	ADC AH, AL	
2	r8, m8	ADC CL, Bvar	
3	m8, r8	ADC Bvar, DH	
4	r8, i8	ADC AH, 012h	

5	m8, i8	ADC Bvar, 034h	
6	r16, r16	ADC AX, BX	
7	r16, m16	ADC CX, Wvar	
8	m16, r16	ADC Wvar, DX	
9	r16, i16	ADC AX, 04711h	
10	m16, i16	ADC Wvar, 00815h	
11	r16, i8	ADC DX, 012h	
12	m16, i8	ADC Wvar, 012h	
13	r32, r32	ADC EAX, EBX	ab 80386
14	r32, m32	ADC ECX, Dvar	ab 80386
15	m32, r32	ADC Dvar, EDX	ab 80386
16	r32, i32	ADC EAX, 012345678h	ab 80386
17	m32, i32	ADC Dvar, 098765432h	ab 80386
18	r32, i8	ADC ESI, 012h	ab 80386
19	m32, i8	ADC Dvar, 034h	ab 80386
20	AL,i8	ADC AL, 012h	nur AL!
21	AX, i16	ADC AX, 01234h	nur AX!
22	EAX, i32	ADC EAX, 012345678h	nur EAX!

Arbeitsweise

ADC addiert den Wert des zweiten Operanden zum Wert des ersten Operanden. Anschließend wird, falls das Carry-Flag gesetzt ist, zum Ergebnis 1 addiert, andernfalls bleibt die Summe unverändert. Das Ergebnis dieser Berechnung wird im ersten Operanden abgelegt.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	9+EA	7	6	2	2
3	16+EA	7	7	3	3
4	4	3	2	1	1
5	17+EA	7	7	3	3
6	3	2	2	1	1
7	9+EA	7	6	2	2
8	16+EA	7	7	3	3
9	4	3	2	1	1
10	17+EA	7	7	3	3
11	4	3	2	1	1
12	17+EA	7	7	3	3
13	-	-	2	1	1
14	-	-	6	2	2
15	-	-	7	3	3
16	-	-	2	1	1
17	-	-	7	3	3
18	-	-	2	1	1
19	-	-	7	3	3
20	4	3	2	1	1
21	4	3	2	1	1
22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	12	/r							
2	12	/m	a16						
3	10	/m	a16						
4	80	/2	i8						
5	80	/2	a16		i8				
6	13	/r							
7	13	/m	a16						
8	11	/m	a16						
9	81	/2	i16						
10	81	/2	a16		i16				
11	83	/2	i8						
12	83	/2	a16		i8				
13	13	/r							mit Präfix OPSIZE
14	13	/m	a16						mit Präfix OPSIZE
15	11	/m	a16						mit Präfix OPSIZE
16	81	/2	i32						mit Präfix OPSIZE
17	81	/2	a16		i32				mit Präfix OPSIZE
18	83	/2	i8						mit Präfix OPSIZE
19	83	/2	a16		i8				mit Präfix OPSIZE
20	14								
21	15								
22	15								

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen

ADC kann wirksam zur Berechnung von Summen benutzt werden, die die Registergröße des Prozessors überschreiten (z.B. zur Addition von Dwords bei 8086-Prozessoren oder für Qwords bei 80386ern). Hierzu wird die erste (»untere«) Hälfte der beiden Summanden mit ADD addiert. Falls ein Überlauf stattfindet, so wird das Carry-Flag gesetzt. Dieser Überlauf kann dann bei der Addition der zweiten (»oberen«) Hälfte der Operanden mittels ADC berücksichtigt werden, ohne Fallunterscheidungen programmieren zu müssen!

Beschreibung

Seite 51

ADD *8086*

Funktion ADD führt eine Addition durch.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Addition gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	ADD AH, AL	
2	r8, m8	ADD CL, Bvar	
3	m8, r8	ADD Bvar, DH	
4	r8, i8	ADD AH, 012h	
5	m8, i8	ADD Bvar, 034h	
6	r16, r16	ADD AX, BX	
7	r16, m16	ADD CX, Wvar	
8	m16, r16	ADD Wvar, DX	
9	r16, i16	ADD AX, 04711h	
10	m16, i16	ADD Wvar, 00815h	
11	r16, i8	ADD DX, 012h	
12	m16, i8	ADD Wvar, 012h	
13	r32, r32	ADD EAX, EBX	ab 80386
14	r32, m32	ADD ECX, Dvar	ab 80386
15	m32, r32	ADD Dvar, EDX	ab 80386
16	r32, i32	ADD EAX, 012345678h	ab 80386
17	m32, i32	ADD Dvar, 098765432h	ab 80386
18	r32, i8	ADD ESI, 012h	ab 80386
19	m32, i8	ADD Dvar, 034h	ab 80386
20	AL,i8	ADD AL, 012h	nur AL!
21	AX, i16	ADD AX, 01234h	nur AX!
22	EAX, i32	ADD EAX, 012345678h	nur EAX!

Arbeitsweise ADD addiert den Wert des zweiten Operanden zum Wert des ersten Operanden und legt das Ergebnis im ersten Operanden ab.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	9+EA	7	6	2	2
3	16+EA	7	7	3	3
4	4	3	2	1	1
5	17+EA	7	7	3	3
6	3	2	2	1	1
7	9+EA	7	6	2	2
8	16+EA	7	7	3	3
9	4	3	2	1	1
10	17+EA	7	7	3	3

11	4	3	2	1	1
12	17+EA	7	7	3	3
13	-	-	2	1	1
14	-	-	6	2	2
15	-	-	7	3	3
16	-	-	2	1	1
17	-	-	7	3	3
18	-	-	2	1	1
19	-	-	7	3	3
20	4	3	2	1	1
21	4	3	2	1	1
22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	02	/r							
2	02	/m	a16						
3	00	/m	a16						
4	80	/0	i8						
5	80	/0	a16		i8				
6	03	/r							
7	03	/m	a16						
8	01	/m	a16						
9	81	/0	i16						
10	81	/0	a16		i16				
11	83	/0	i8						
12	83	/0	a16		i8				
13	03	/r							mit Präfix OPSIZE
14	03	/m	a16						mit Präfix OPSIZE
15	01	/m	a16						mit Präfix OPSIZE
16	81	/0	i32						mit Präfix OPSIZE
17	81	/0	a16		i32				mit Präfix OPSIZE
18	83	/0	i8						mit Präfix OPSIZE
19	83	/0	a16		i8				mit Präfix OPSIZE
20	04								
21	05								
22	05								

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Im Unterschied zu ADC wird bei der Addition das Carry-Flag nicht berücksichtigt, wohl aber anhand des Ergebnisses der Addition verändert!

Beschreibung Seite 51

AND *8086*

Funktion AND verknüpft die beiden Operanden durch eine logische UND-Verknüpfung.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				0				*	*		?		*		0

Die Flags *sign*, *zero* und *parity* werden in Abhängigkeit vom Ergebnis der Verknüpfung gesetzt. Das Overflow- und Carry-Flag werden explizit gelöscht, das *Auxiliary-Flag* ist nicht definiert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	AND AH, BL	
2	r8, m8	AND CL, Bvar	
3	m8, r8	AND Bvar, CH	
4	r8, i8	AND AL, 012h	
5	m8, i8	AND Bvar, 034h	
6	r16, r16	AND AX, BX	
7	r16, m16	AND CX, Wvar	
8	m16, r16	AND Wvar, DX	
9	r16, i16	AND AX, 04711h	
10	m16, i16	AND Wvar, 00815h	
11	r16, i8	AND DX, 012h	ab 80386
12	m16, i8	AND Wvar, 012h	ab 80386
13	r32, r32	AND EAX, EBX	ab 80386
14	r32, m32	AND ECX, Dvar	ab 80386
15	m32, r32	AND Dvar, EDX	ab 80386
16	r32, i32	AND EAX, 012345678h	ab 80386
17	m32, i32	AND Dvar, 098765432h	ab 80386
18	r32, i8	AND ESI, 012h	ab 80386
19	m32, i8	AND Dvar, 034h	ab 80386
20	AL,i8	AND AL, 012h	nur AL!
21	AX, i16	AND AX, 01234h	nur AX!
22	EAX, i32	AND EAX, 012345678h	nur EAX!

Arbeitsweise

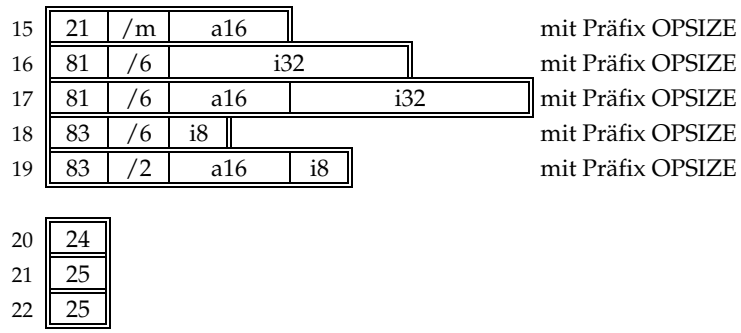
AND führt eine logische UND-Verknüpfung durch. Hierbei wird bitweise der erste Operand mit dem zweiten Operand UND-verknüpft, das Resultat wird dann in den ersten Operanden eingetragen. Der zweite Operand bleibt unverändert.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	9+EA	7	6	2	2
3	16+EA	7	7	3	3
4	4	3	2	1	1
5	17+EA	7	7	3	3
6	3	2	2	1	1
7	9+EA	7	6	2	2
8	16+EA	7	7	3	3
9	4	3	2	1	1
10	17+EA	7	7	3	3
11	-	-	2	1	1
12	-	-	7	3	3
13	-	-	2	1	1
14	-	-	6	2	2
15	-	-	7	3	3
16	-	-	2	1	1
17	-	-	7	3	3
18	-	-	2	1	1
19	-	-	7	3	3
20	4	3	2	1	1
21	4	3	2	1	1
22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	22	/r							
2	22	/m	a16						
3	20	/m	a16						
4	80	/6	i8						
5	80	/6	a16					i8	
6	23	/r							
7	23	/m	a16						
8	21	/m	a16						
9	81	/6	i16						
10	81	/6	a16					i16	
11	83	/6	i8						
12	83	/6	a16					i8	
13	23	/r							mit Präfix OPSIZE
14	23	/m	a16						mit Präfix OPSIZE



Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 46

ARPL *80286*

Funktion Anpassung des sogenannten RPL-Feldes des Selektors an den CPL.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									?						

Das Zero-Flag ist gesetzt, wenn das RPL-Feld im ersten Operanden (*dest*) kleiner als das im zweiten (*source*) ist. Alle weiteren Flags bleiben unberührt.

Verwendung

#	Parameter	Beispiel
1	r16, r16	ARPL AX, BX
2	m16, r16	ARPL WVar, DX

Arbeitsweise Beide Operanden müssen Selektoren beinhalten. Das RPL-Feld des Selektors wird durch die Bits 0 und 1 des Selektors codiert. ARPL vergleicht nun die RPL-Felder der beiden Operanden. Ist das RPL-Feld des ersten Operanden kleiner als das des zweiten, so wird das *Zero-Flag* gesetzt und der Wert aus dem zweiten Operanden in den ersten kopiert. Andernfalls wird das *Zero-Flag* gelöscht und weitere Aktionen unterbleiben.

Takte

#	8086	80286	80386	80486	Pentium
1	-	10	20	9	7
2	-	11	21	9	7

Opcodes

#	B1	B2	
1	63	/r	
2	63	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	virtual 8086 mode code	virtual 8086 mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	3, 8, 20	8
#PF	?	1	?	1	./
#SS	0	1	0	1	1
#UD	-	-	-	1	1

Bemerkungen

ARPL stellt sicher, daß ein zu überprüfender Selektor, der als erster Operand angegeben wird, niemals ein niedrigeres RPL-Feld besitzt als ein Vergleichsselektor. Auf diese Weise wird verhindert, daß das Modul/Programm, das durch den zu überprüfenden Selektor repräsentiert wird, mehr Privilegien (= niedrigeres RPL, *privilege level*) erhält, als ihm durch den zweiten beschrieben zustehen, es also auf Segmente zugreifen kann, die ihm verboten sind. Daher ist der Befehl ARPL auch nur in Prüfroutinen des Betriebssystems sinnvoll, auch wenn er von Anwendungsprogrammen genutzt werden kann.

Beschreibung

Seite 237.

BOUND *80186*

Funktion

Prüfung eines Index anhand zweier vorgegebener Grenzen.

Flags

X X X X O D I T S Z X A X P X C

Es werden keine Flags verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16, adr	BOUND AX, WVar	
2	r32, adr	BOUND ECX, DVar	ab 80386

Arbeitsweise

BOUND vergleicht den Wert im ersten Operanden zunächst mit dem Wert, der durch die Adressenangabe des zweiten Operanden spezifiziert wird. Ist das Ergebnis des Vergleichs negativ, der zu prüfende Wert also kleiner als der Grenzwert an der durch den zweiten Operanden übergebenen Adresse, so wird ein Interrupt $05 ausgelöst. Der Rücksprung aus der dadurch aufgerufenen Interrupt-Routine führt dann auf den BOUND-Befehl zurück, so daß die Aktionen in der Interrupt-Routine nochmals überprüft werden.

Im anderen Fall erfolgt eine weitere Prüfung mit der spezifizierten Obergrenze, die unmittelbar auf den Wert der Untergrenze folgen muß, da ihre Adresse nicht explizit übergeben wird. Ist das Ergebnis des Vergleichs hier positiv, so liegt der zu prüfende Wert oberhalb der Obergrenze, und es wird ebenfalls ein Interrupt $05 ausgelöst. Auch in diesem Fall führt der Rücksprung aus der Interrupt-Routine wieder auf den BOUND-Befehl.

Takte

#	8086	80286	80386	80486	Pentium
1	-	13	10	7	8
2	-	-	10	7	8

Die Taktraten gelten natürlich nur für den Fall, daß der Wert innerhalb der Grenzen liegt. Andernfalls kommen noch weitere Taktraten für den Interrupt hinzu.

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	62	/m	a16		
2	62	/m	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#BR	-	1	-	1	1
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
#UD	-	2	-	2	2

Bemerkungen

Adr ist ein Zeiger auf eine Speicherstelle, an der zwei Grenzwerte liegen. Der erste gibt die Unter-, der zweite die Obergrenze an, gegen die geprüft werden soll. ACHTUNG: Die Größe dieser beiden Werte richtet sich nach der Wahl des *regs*. Ist *reg* ein 16-Bit-Register wie AX, DX oder SI, so interpretiert BOUND die Struktur an der Stelle *Adr* auch als zwei hintereinander stehende 16-Bit-Werte. Im Falle der Verwendung eines 32-Bit-Registers als erstem Operanden, wie EBX oder EDI, verwendet BOUND die nächsten zweimal 32 Bits an der Stelle *Adr* als Grenzwerte.

Beschreibung

Seite 121

BSF 80386

Funktion: Bitweises Durchsuchen eines Operanden; die Suchrichtung ist »vorwärts« (*B*it *S*earch *F*orward).

Flags:

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									*						

BSF verändert nur das Zero-Flag, alle anderen Flags bleiben unverändert.

Verwendung:

#	Parameter	Beispiel
1	r16, r16	BSF AX, BX
2	r16, m16	BSF AX, WVar
3	r32, r32	BSF EAX, EBX
4	r32, m32	BSF EAX, DVar

Arbeitsweise: BSF untersucht den als zweiten Operanden angegebenen Wert, beginnend mit dem niedrigstwertigen Bit (Bit 0) in Richtung des höchstwertigen Bits (Bit 15 bzw. 31). Die Nummer des ersten gesetzten Bits wird dann in das im ersten Operanden genannte Register abgelegt, und das Zero-Flag wird gelöscht. So resultiert aus *BSF AX, WordVar* ein AX = $0002, falls in *WordVar* z.B. der Wert $1234 gestanden hat, da Bit 2 in $1234 gesetzt ist und dieses, bei Bit 0 beginnend, das erste gesetzte Bit ist. Ist kein Bit im zweiten Operanden gesetzt (der Wert dieses Operanden ist dann 0), so wird im ersten Operanden ebenfalls 0 übergeben und das Zero-Flag gesetzt.

Takte:

#	8086	80286	80386	80486	Pentium
1	-	-	10+3·n	6-42	6-34
2	-	-	10+3·n	6-42	6-35
3	-	-	10+3·n	6-42	6-42
4	-	-	10+3·n	6-42	6-43

Opcodes:

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	0F	BC	/r						
2	0F	BC	/m	a16					
3	0F	BC	/r						mit Präfix OPSIZE
4	0F	BC	/m	a16					mit Präfix OPSIZE

Exceptions:

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Der erste und der zweite Operand müssen immer die gleiche Größe haben. *BSF EAX, WordVar* oder *BSF BX, DWordVar* sind nicht erlaubt. Die Angabe »n« in der Rubrik »80386« unter »Takte« bezeichnet die Anzahl der Bitprüfungen, die notwendig sind, bis BSF zu einem Ergebnis gekommen ist. So ist n=0, falls das erste untersuchte Bit schon gesetzt ist, während n=15 ist, falls alle Bits gelöscht sind.

Beschreibung Seite 134

BSR *80386*

Funktion Bitweises Durchsuchen des Operanden mit umgekehrter Suchrichtung (*B*it *S*earch *R*everse).

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									*						

BSR verändert nur das Zero-Flag, alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
1	r16, r16	BSR AX, BX
2	r16, m16	BSR AX, WVar
3	r32, r32	BSR EAX, EBX
4	r32, m32	BSR EAX, DVar

Arbeitsweise BSR arbeitet wie BSF, jedoch beginnend mit dem höchstwertigen Bit (Bit 15 bzw. 31) und in Richtung Bit 0 suchend. Die Nummer des ersten gesetzten Bits wird dann in das im ersten Operanden genannte Register abgelegt, und das Zero-Flag wird gelöscht. So resultiert aus *BSR AX, WordVar* ein AX = $000C, falls in *WordVar* z.B. der Wert $1234 gestanden hat, da Bit 12 in $1234 gesetzt ist und dieses, bei Bit 15 beginnend, das erste gesetzte Bit ist. Ist kein Bit im zweiten Operanden gesetzt (der Wert dieses Operanden ist dann 0), so wird im ersten Operanden ebenfalls 0 übergeben und das Zero-Flag gesetzt.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	10+3·n	6-103	7-39
2	-	-	10+3·n	6-103	7-40
3	-	-	10+3·n	6-103	7-71
4	-	-	10+3·n	6-103	7-72

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	0F	BD	/r						
2	0F	BD	/m	a16					
3	0F	BD	/r						mit Präfix OPSIZE
4	0F	BD	/m	a16					mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Siehe auch BSF. Die Angabe »n« in der Rubrik »80386« unter »Takte« bezeichnet die Anzahl der Bitprüfungen, die notwendig sind, bis BSF zu einem Ergebnis gekommen ist. So ist n=0, falls das erste untersuchte Bit schon gesetzt ist, während n=15 ist, falls alle Bits gelöscht sind.

Beschreibung Seite 135

BSWAP *80486*

Funktion Vertauschen der Byte-Reihenfolge in einem Doppelwort.

Flags X X X X O D I T S Z X A X P X C

Es werden keine Flags verändert.

Verwendung

#	Parameter	Beispiel
	r32	BSWAP EAX

Arbeitsweise BSWAP legt zunächst eine temporäre Kopie des Operanden an. Dann wird der Inhalt der Kopie wie folgt byteweise in das Register kopiert:

Temp		Operand
Byte 3	⇒	Byte 0
Byte 2	⇒	Byte 1
Byte 1	⇒	Byte 2
Byte 0	⇒	Byte 3

Takte	#	8086	80286	80386	80486	Pentium
		-	-	-	1	1

Opcodes

#	B1	B2	Bemerkungen
	0F	C8	BSWAP EAX
	0F	C9	BSWAP ECX
	0F	CA	BSWAP EDX
	0F	CB	BSWAP EBX
	0F	CC	BSWAP ESP
	0F	CD	BSWAP EBP
	0F	CE	BSWAP ESI
	0F	CF	BSWAP EDI

Exceptions Keine

Beschreibung Seite 148

BT *80386*

Funktion Prüfung auf gesetztes Bit (*B*it *T*est).

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															*

Der Wert des zu prüfenden Bits wird im Carry-Flag gesichert, alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
1	r16, r16	BT AX, BX
2	m16, r16	BT WVar, AX
3	r16, i8	BT AX, 002h
4	m16, i8	BT WVar, 00Ch
5	r32, r32	BT EAX, EBX
6	m32, r32	BT DVar, EDX
7	r32, i8	BT ESI, 01Fh
8	m32, i8	BT DVar, 010h

Arbeitsweise BT kopiert den Zustand des Bits im ersten Operanden (oft als Basis bezeichnet), das durch den Wert des zweiten Operanden bezeichnet wird, in das Carry-Flag. So stellt BT im Beispiel #3 oben den Zustand des Bits #2 des Wertes in AX fest. Ist dieses Bit gesetzt, so wird das Carry-Flag ebenfalls gesetzt, andernfalls gelöscht.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	3	3	4
2	-	-	12	8	9
3	-	-	3	3	4
4	-	-	6	3	4
5	-	-	3	3	4
6	-	-	12	8	9
7	-	-	3	3	4
8	-	-	6	3	4

Opcodes

#	B1	B2	B3	B4	B5	B6	Bemerkungen
1	0F	A3	/r				
2	0F	A3	/m	a16			
3	0F	BA	/4	i8			
4	0F	BA	/4	a16		i8	
5	0F	A3	/r				mit Präfix OPSIZE
6	0F	A3	/m	a16			mit Präfix OPSIZE
7	0F	BA	/4	i8			mit Präfix OPSIZE
8	0F	BA	/4	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Siehe auch BTC, BTR, BTS.

Beschreibung Seite 135

BTC *80386*

Funktion Prüfung auf gesetztes Bit mit anschließender Bildung des Komplementärwertes des Bits (*B*it *T*est and *C*omplement).

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															*

Der komplementäre Wert des zu prüfenden Bits wird im Carry-Flag gesichert, alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
1	r16, r16	BTC AX, BX
2	m16, r16	BTC WVar, AX
3	r16, i8	BTC AX, 002h
4	m16, i8	BTC WVar, 00Ch
5	r32, r32	BTC EAX, EBX
6	m32, r32	BTC DVar, EDX
7	r32, i8	BTC ESI, 01Fh
8	m32, i8	BTC DVar, 010h

Arbeitsweise

BTC arbeitet analog zu BT. Im Unterschied zu diesem Befehl setzt aber BTC nach der Bitprüfung und -kopie das spezifizierte Bit im ersten Operanden zurück, falls es gesetzt ist, oder setzt es, falls es gelöscht ist (Komplementbildung).

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	6	6	7
2	-	-	13	13	13
3	-	-	6	6	7
4	-	-	8	8	8
5	-	-	6	6	7
6	-	-	13	13	13
7	-	-	6	6	7
8	-	-	8	8	8

Opcodes

#	B1	B2	B3	B4	B5	B6	Bemerkungen
1	0F	BB	/r				
2	0F	BB	/m	a16			
3	0F	BA	/7	i8			
4	0F	BA	/7	a16		i8	
5	0F	BB	/r				mit Präfix OPSIZE
6	0F	BB	/m	a16			mit Präfix OPSIZE
7	0F	BA	/7	i8			mit Präfix OPSIZE
8	0F	BA	/7	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen

Siehe auch BT, BTR, BTS.

Beschreibung

Seite 135

BTR 80386

Funktion Prüfung auf gesetzes Bit mit anschließendem Löschen (*B*it *T*est and *R*eset).

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															*

Der Wert des zu prüfenden Bits wird im Carry-Flag gesichert, alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
1	r16, r16	BTR AX, BX
2	m16, r16	BTR WVar, AX
3	r16, i8	BTR AX, 002h
4	m16, i8	BTR WVar, 00Ch
5	r32, r32	BTR EAX, EBX
6	m32, r32	BTR DVar, EDX
7	r32, i8	BTR ESI, 01Fh
8	m32, i8	BTR DVar, 010h

Arbeitsweise BTR arbeitet analog zu BT. Im Unterschied zu diesem Befehl löscht aber BTR nach der Bitprüfung und -kopie das spezifizierte Bit im ersten Operanden explizit.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	6	6	7
2	-	-	13	13	13
3	-	-	6	6	7
4	-	-	8	8	8
5	-	-	6	6	7
6	-	-	13	13	13
7	-	-	6	6	7
8	-	-	8	8	8

Opcodes

#	B1	B2	B3	B4	B5	B6	Bemerkungen
1	0F	B3	/r				
2	0F	B3	/m	a16			
3	0F	BA	/6	i8			
4	0F	BA	/6	a16		i8	
5	0F	B3	/r				mit Präfix OPSIZE
6	0F	B3	/m	a16			mit Präfix OPSIZE
7	0F	BA	/6	i8			mit Präfix OPSIZE
8	0F	BA	/6	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch BT, BTC, BTS.

Beschreibung Seite 135

BTS *80386*

Funktion Prüfung auf gesetzes Bit mit anschließendem Setzen (*B*it *T*est and *S*et).

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															*

Der Wert des zu prüfenden Bits wird im Carry-Flag gesichert, alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
1	r16, r16	BTS AX, BX
2	m16, r16	BTS WVar, AX
3	r16, i8	BTS AX, 002h
4	m16, i8	BTS WVar, 00Ch
5	r32, r32	BTS EAX, EBX
6	m32, r32	BTS DVar, EDX
7	r32, i8	BTS ESI, 01Fh
8	m32, i8	BTS DVar, 010h

Arbeitsweise BTS arbeitet analog zu BT. Im Unterschied zu diesem Befehl setzt aber BTS nach der Bitprüfung und -kopie das spezifizierte Bit im ersten Operanden explizit.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	6	6	7
2	-	-	13	13	13
3	-	-	6	6	7
4	-	-	8	8	8
5	-	-	6	6	7
6	-	-	13	13	13
7	-	-	6	6	7
8	-	-	8	8	8

Opcodes

#	B1	B2	B3	B4	B5	B6	Bemerkungen
1	0F	AB	/r				
2	0F	AB	/m	a16			
3	0F	BA	/5	i8			
4	0F	BA	/5	a16		i8	
5	0F	AB	/r				mit Präfix OPSIZE
6	0F	AB	/m	a16			mit Präfix OPSIZE
7	0F	BA	/5	i8			mit Präfix OPSIZE
8	0F	BA	/5	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch BT, BTC, BTR.

Beschreibung Seite 135

CALL *8086*

Funktion Aufruf eines Unterprogramms (einer sogenannten Prozedur).

Flags X X X X O D I T S Z X A X P X C

Call verändert den Zustand der Flags nicht.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	i16	CALL NearLabel	near; relativ!
2	m16	CALL [WVar]	near; absolut!
3	r16	CALL [AX]	near; absolut!
4	i16:16	CALL FarLabel	far; absolut
5	m16:16	CALL [DVar]	far; absolut
6	i32	CALL FarLabel	ab 80386
7	m32	CALL [DVar]	ab 80386
8	r32	CALL [EAX]	ab 80386
9	i16:32	CALL FlatLabel	ab 80386
10	m32:16	CALL [VAR]	ab 80386

Arbeitsweise

CALL legt die auf den CALL-Befehl folgende Adresse auf den Stack und lädt die neue Zieladresse in die Register CS und/oder IP. Dies kann zum einen nur das IP-Register betreffen, wenn es sich um Sprünge innerhalb des Segments handelt, da hierbei die Segmentadresse (in CS) gleich bleibt. Solche Sprünge nennt man *Near Jumps* (siehe Verwendung #1). Bei ihnen wird der Operand vorzeichenbehaftet interpretiert und zum Inhalt des IP-Registers addiert. Das heißt, die direkten *Near Jumps* haben als Operanden eine Sprungweite, die die Differenz von aktueller Adresse und Zieladresse repräsentiert.

Auch im sogenannten *Flat-Modell* der 80386er und Nachfolger, in dem eine lineare 32-Bit-Adresse Verwendung finden kann, spricht man von *Near Jumps* (siehe Verwendung #6). Hier wird die 32-Bit-Sprungweite zum Inhalt des EIP-Registers vorzeichenbehaftet addiert. Allerdings kann auch ein Intersegmentsprung (*Far Jump*) durchgeführt werden (siehe Verwendung #4). Hierbei muß eine vollständige Adresse, bestehend aus Segment- und Offsetanteil, angegeben werden. Die ersten 16 Bits der Adresse werden dabei als Zieloffset interpretiert und in das IP-Register geladen, die folgenden 16 Bits als Segmentanteil in CS.

Auch im Falle des *Flat-Modells* ist ein »Intersegmentsprung« möglich (siehe Verwendung #9). Hierbei wird eine lineare 32-Bit-Adresse sowie ein 16-Bit-«Selektor« angegeben. Eine detailliertere Besprechung würde hier allerdings zu weit führen. CALL läßt auch indirekte Sprünge zu, bei denen das Ziel nicht selbst als Adresse übergeben wird. Bei solchen Sprüngen (*indirect jumps*) wird die Zieladresse vielmehr aus einer Adresse ausgelesen. Bei *Near Jumps* kann das entweder eine 16-Bit-Speicherstelle (siehe Verwendung #2) oder ein Register (siehe Verwendung #3) sein. Der an der spezifizierten Stelle stehende Wert wird dann in das IP-Register kopiert. Es handelt sich hierbei also um *absolute* Sprünge, da das Ziel als absolute Adresse und nicht wie im Falle von *Near Jumps* als Relativadresse angegeben wird. Selbstverständlich gibt es *indirekte Near Jumps* auch im *Flat-Modell* (siehe Verwendung #7, #8).

Far Jumps sind ebenfalls indirekt möglich, allerdings kann hier die Adresse nur über eine Speicherstelle übergeben werden (siehe Verwendung #5). Üblicherweise erfolgt dies über zwei 16-Bit-Werte an der als Operand übergebenen Adresse. Die dort stehenden Werte werden dann in IP und CS kopiert – die Offsetanteile stehen immer vor den Segmentanteilen. Beim 80386 ist auch im *Flat-Modell* die Angabe eines 32-Bit-Offsets, gefolgt von einem 16-Bit-»Segment« an der Adresse möglich (siehe Verwendung #10).

Takte

#	8086	80286	80386	80486	Pentium
1	19	7	7+m	3	1
2	21	11	10+m	5	2
3	16	7	7+m	5	2
4	28	13	17+m	18	4
5	37+EA	16	22+m	17	5
6	-	-	7+m	3	1
7	-	-	10+m	5	2
8	-	-	7+m	5	2
9	-	-	17+m	18	4
10	-	-	22+m	17	5

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	E8	i16							
2	FF	/2	a16						
3	FF	/2							
4	9A	a16		a16					Offset : Segment!
5	FF	/3	a16						
6	E8	a32							mit Präfix OPSIZE
7	FF	/2	a32						mit Präfix OPSIZE
8	FF	/2							mit Präfix OPSIZE
9	9A	a32				a16			mit Präfix OPSIZE
10	FF	/3	a32				a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 4, 5, 8, 10	0	8, 10	8, 10
	?	12, 23, 24, 35, 36, 38, 40, 47, 64	?	-	-
#NP	?	1	?	-	./.
#PF	?	1	?	1	./.
#SS	0	1, 5	0	-	-
	?	5, 6, 8	?	-	-
#TS	?	1, 2, 3	?	-	./.

Bemerkungen

Der CALL-Befehl hat im Protected-Mode des 80286 und im Virtual-8086-Mode der 80386-Prozessoren sowie deren Nachfolgern eine Erweiterung erfahren, die die Privilegstufen und Taskwechsel berücksichtigt. Dies soll jedoch in diesem Zusammenhang nicht beschrieben werden.

CBW *8086*

Funktion

Erweiterung eines Bytes zu einem Wort.

Flags

X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	CBW

Arbeitsweise CBW konvertiert ein Byte in AL in ein Wort in AX. Dies erfolgt, indem das Bit 7 des Bytes in die Bits 8 bis 15 kopiert wird. Auf diese Weise ist sichergestellt, daß ein vorzeichenbehaftetes Byte ebenfalls korrekt in ein vorzeichenbehaftetes Wort expandiert wird.

Takte	#	8086	80286	80386	80486	Pentium
		2	2	3	3	3

Opcodes	#	B1
		98

Exceptions Keine

Beschreibung Seite 78

CDQ *80386*

Funktion Erweiterung eines Doppelworts zu einem Quadwort.

Flags	X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung	#	Parameter	Beispiel
		keine	CDQ

Arbeitsweise CDQ konvertiert ein Doppelwort in EAX in ein Quadwort in EDX:EAX. Dies erfolgt, indem das Bit 31 des DWords in EAX in die Bits 0 bis 31 des DWords in EDX kopiert wird. Auf diese Weise ist sichergestellt, daß ein vorzeichenbehaftetes DWord ebenfalls korrekt in ein vorzeichenbehaftetes QWord expandiert wird.

Takte	#	8086	80286	80386	80486	Pentium
		-	-	2	3	2

Opcodes	#	B1
		99

Exceptions Keine

Beschreibung Seite 138

CLC 8086

Funktion Carry-Flag löschen.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															0

Das Carry-Flag wird gelöscht. Alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
	keine	CLC

Takte

#	8086	80286	80386	80486	Pentium
	2	2	2	2	2

Opcodes

#	B1
	F8

Exceptions Keine

Beschreibung Seite 40

CLD 8086

Funktion Direction-Flag löschen.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
					0										

Das Direction-Flag wird gelöscht. Alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
	keine	CLD

Takte

#	8086	80286	80386	80486	Pentium
	2	2	2	2	2

Opcodes

#	B1
	FC

Exceptions Keine

Beschreibung Seite 41

CLI 8086

Funktion Interrupts verbieten.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
						0									

Das Interrupt-Enable-Flag wird gelöscht. Alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
	keine	CLI

Takte

#	8086	80286	80386	80486	Pentium
	2	3	3	5	7

Opcodes

#	B1
	FA

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	33	0	33	-

Beschreibung Seite 41

CLTS 80286

Funktion Task-Switch- Flag löschen

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	CLTS

Arbeitsweise Löscht das Task-Switch-Flag in Register CR0.

Takte

#	8086	80286	80386	80486	Pentium
	-	2	5	7	10

Opcodes

#	B1	B2
	0F	06

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#GP	0	32	0	32	-

Bemerkungen CLTS kann nur im Rahmen der Betriebssystemaktivitäten benutzt werden. Es ist eine sogenannte privilegierte Funktion, die nur ausgeführt werden kann, wenn der aktuelle *privilege level* CPL = 0 ist (= höchste Privilegstufe, nur in Betriebssystemmodulen!). CLTS darf auch vom Real-Mode aus ausgeführt werden, um den Protected-Mode zu initialisieren.

Der Prozessor setzt das *Task-Switch-Flag* TS im Kontrollregister CR0 bei jedem Taskwechsel, um in Multitasking-Umgebungen eine Synchronisation mit der FPU (*floating point unit*) und das Sichern von deren Umgebung zu erreichen.

Beschreibung Seite 947

CMC *8086*

Funktion Dieser Befehl kehrt das Carry-Flag um.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															*

Der Zustand des Carry-Flags wird invertiert. Alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel
	keine	CMC

Takte

#	8086	80286	80386	80486	Pentium
	2	2	2	2	2

Opcodes

#	B1
	F5

Exceptions Keine

Beschreibung Seite 40

CMOVcc Pentium

Funktion Bedingtes Kopieren eines Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16, r16	CMOVA AX, DI
2	r16, m16	CMOVNBE DX, WVar
3	r32, r32	CMOVL ECX, EDX
4	r32, m32	CMOVNG EDI, DVar

Arbeitsweise CMOVcc arbeitet wie der MOV-Befehl, nur daß das Kopieren unter bestimmten Bedingungen erfolgt. Die Bedingungen sind die gleichen, die auch von den bedingten Sprüngen Jcc her bekannt sind. Ist die jeweilige Bedingung erfüllt, so wird der MOV-Befehl ausgeführt, andernfalls nicht.

CMOV wird also immer nach einem Befehl durchgeführt, bei dem die Flags im EFlagregister verändert werden, also zum Beispiel nach einem Vergleich zweier Zahlen mittels CMP, nach arithmetischen Befehlen wie ADD etc. Es gibt folgende Bedingungen:

- CMOVA, CMOVAE, CMOVB, CMOVBE, CMOVNA, CMOVNAE, CMOVNB, CMOVNBE; Bedingungen, die bei einem Vergleich vorzeichenloser Zahlen bestehen. Ausgewertet werden das Carry- und/oder Zero-Flag.
- CMOVG, CMOVGE, CMOVL, CMOVLE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE; Bedingungen, die bei einem Vergleich vorzeichenbehafteter Zahlen bestehen. Ausgewertet werden das Sign-, Zero- und/oder Overflow-Flag.
- CMOVE, CMOVNE, CMOVNZ, CMOVZ; Bedingungen, die bei einem Vergleich von Zahlen allgemein bestehen. Ausgewertet wird das Zero-Flag.
- CMOVC, CMOVNC, CMOVNO, CMOVNP, CMOVNS, CMOVO, CMOVP, CMOVPE, CMOVPO, CMOVS; Bedingungen, die einzelne Flags betreffen und von verschiedenen Befehlen erzeugt werden.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	Bemerkungen
	66	0F	47	i16		*i32*		CMOVA
	66	0F	43	i16		*i32*		CMOVAE
	66	0F	42	i16		*i32*		CMOVB
	66	0F	46	i16		*i32*		CMOVBE
	66	0F	42	i16		*i32*		CMOVC
	66	0F	44	i16		*i32*		CMOVE
	66	0F	4F	i16		*i32*		CMOVG
	66	0F	4D	i16		*i32*		CMOVGE
	66	0F	4C	i16		*i32*		CMOVL
	66	0F	4E	i16		*i32*		CMOVLE
	66	0F	46	i16		*i32*		CMOVNA
	66	0F	42	i16		*i32*		CMOVNAE
	66	0F	43	i16		*i32*		CMOVNB
	66	0F	47	i16		*i32*		CMOVNBE
	66	0F	43	i16		*i32*		CMOVNC
	66	0F	45	i16		*i32*		CMOVNE
	66	0F	4E	i16		*i32*		CMOVNG
	66	0F	4C	i16		*i32*		CMOVNGE
	66	0F	4D	i16		*i32*		CMOVNL
	66	0F	4F	i16		*i32*		CMOVNLE
	66	0F	41	i16		*i32*		CMOVNO
	66	0F	4B	i16		*i32*		CMOVNP
	66	0F	49	i16		*i32*		CMOVNS
	66	0F	45	i16		*i32*		CMOVNZ
	66	0F	40	i16		*i32*		CMOVO
	66	0F	4A	i16		*i32*		CMOVP
	66	0F	4A	i16		*i32*		CMOVPE
	66	0F	4B	i16		*i32*		CMOVPO
	66	0F	48	i16		*i32*		CMOVS
	66	0F	44	i16		*i32*		CMOVZ

* Die kursiv gedruckten Bytefolgen beziehen sich auf die 32-Bit-Versionen des entsprechenden Befehls. Hier dient wie üblich das Präfix $66 als Unterscheidungsmerkmal, das der folgenden 2-Byte-Opcode-Sequenz statt einer 16-Bit-Konstanten eine 32-Bit-Konstante folgt!

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Nicht alle Prozessoren der Pentium-Pro-Familie unterstützen diesen Befehl. Ob er bei einem bestimmten Prozessor unterstützt wird oder nicht, kann mit der CPUID-Instruktion festgestellt werden. Falls sowohl das CMOV- Flag gesetzt ist, wird FCMOVcc unterstützt.

Beschreibung Seite 149

CMP *8086*

Funktion Vergleich zweier Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Vergleich gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	CMP AH, BL	
2	r8, m8	CMP CL, BVar	
3	m8, r8	CMP BVar, CH	
4	r8, i8	CMP AL, 012h	
5	m8, i8	CMP BVar, 034h	
6	r16, r16	CMP AX, BX	
7	r16, m16	CMP CX, WVar	
8	m16, r16	CMP WVar, DX	
9	r16, i16	CMP AX, 04711h	
10	m16, i16	CMP WVar, 0815h	
11	r16, i8	CMP DX, 012h	
12	m16, i8	CMP WVar, 012h	
13	r32, r32	CMP EAX, EBX	ab 80386
14	r32, m32	CMP ECX, DVar	ab 80386
15	m32, r32	CMP DVar, EDX	ab 80386
16	r32, i32	CMP EAX, 123456h	ab 80386
17	m32, i32	CMP DVar, 987654h	ab 80386
18	r32, i8	CMP ESI, 012h	ab 80386
19	m32, i8	CMP DVar, 034h	ab 80386
20	AL,i8	CMP AL, 012h	nur AL!
21	AX, i16	CMP AX, 01234h	nur AX!
22	EAX, i32	CMP EAX, 123456h	nur EAX!

Arbeitsweise CMP führt einen Vergleich durch. Hierbei wird formal, d.h. nicht realiter, der zweite Operand vom ersten abgezogen, und anhand des Resultats werden die Bits gesetzt. Beide Operanden bleiben unverändert!

Takte	#	8086	80286	80386	80486	Pentium
	1	3	2	2	1	1
	2	9+EA	6	6	2	2
	3	9+EA	7	5	2	2
	4	4	3	2	1	1
	5	10+EA	6	5	2	2
	6	3	2	2	1	1
	7	9+EA	6	6	2	2
	8	9+EA	7	5	2	2
	9	4	3	2	1	1
	10	10+EA	7	7	3	2
	11	4	3	2	1	1
	12	10+EA	6	5	3	2
	13	-	-	2	1	1
	14	-	-	6	2	2
	15	-	-	5	2	2
	16	-	-	2	1	1
	17	-	-	5	2	2
	18	-	-	2	1	1
	19	-	-	5	2	2
	20	4	3	2	1	1
	21	4	3	2	1	1
	22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	3A	/r							
2	3A	/m	a16						
3	38	/m	a16						
4	80	/7	i8						
5	80	/7	a16					i8	
6	3B	/r							
7	3B	/m	a16						
8	39	/m	a16						
9	81	/7	i16						
10	81	/7	a16				i16		
11	83	/7	i8						
12	83	/7	a16					i8	
13	3B	/r							mit Präfix OPSIZE
14	3B	/m	a16						mit Präfix OPSIZE
15	39	/m	a16						mit Präfix OPSIZE
16	81	/7	i32						mit Präfix OPSIZE
17	81	/7	a16				i32		mit Präfix OPSIZE
18	83	/7	i8						mit Präfix OPSIZE
19	83	/2	a16					i8	mit Präfix OPSIZE

20	3C
21	3D
22	3D

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Beschreibung Seite 25

CMPS *8086*
CMPSB *8086*
CMPSW *8086*
CMPSD *80386*

Funktion Vergleich zweier Strings.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Vergleich gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	keine	CMPSB	
2a	keine	CMPSW	
3a	keine	CMPSD	ab 80386
1b	m8, m8	CMPS BString1, BString2	*
2b	m16, m16	CMPS WString1, WString2	*
3b	m32, m32	CMPS DString1, DString2	* ab 80386

*ACHTUNG! Bei der Verwendung der allgemeinen Form müssen mehrere Dinge beachtet werden:

- Durch die bloße Angabe der beiden Pseudooperanden werden die Registerkombinationen DS:SI (DS:EDI) und ES:DI (ES:EDI) noch nicht korrekt besetzt! Dies hat unbedingt gesondert vor der Verwendung des Stringbefehls zu erfolgen!

- Der Assembler achtet auf die korrekte Angabe der Operanden. So muß als erster Operand der String angegeben werden, dessen Segmentanteil in DS steht. Als zweiter Operand wird dann der String erwartet, dessen Segmentanteil in ES steht! Stimmen diese Segmentanteile der Adressen nicht mit den in DS/ES stehenden überein, so erfolgt eine Fehlermeldung!
- Die verwendeten Strings müssen korrekt definiert worden sein, da nur über ihre Definition der Assembler die benötigte Information über die zu verwendende Datumsgröße erhält! In Verwendung #1b muß daher eine Definition der Art

  ```
  BString1 DB 128 DUP (?)
  ```

  erfolgen. Durch die Anweisung DB kann der Assembler den Befehl *CMPS BString1, BString2* in den Befehl *CMPSB* übersetzen. Analoges gilt für die Verwendungen #2b und #3b!

Arbeitsweise

Mit CMPS ist ein Vergleich zweier Strings möglich. Als String bezeichnet man beim Assembler eine Folge von Variablen gleicher Größe. So gibt es *Byte-Strings*, *Wort-Strings* und *Doppelwort-Strings*. Die Größe der Strings ist keinen Beschränkungen unterworfen, wird aber durch das gewählte Speichermodell limitiert. So können im Real-Mode die Strings maximal 64 kByte, entsprechend der Größe eines Segments, sein.

CMPS (*CoMPare Strings*) ist der Überbegriff für die Befehle CMPSB, CMPSW und CMPSD und wird durch den Assembler in einen der drei Befehle übersetzt. CMPS hat deshalb zwei nur zur Assemblierung benötigte Pseudooperanden, die jedoch nur angeben, ob byteweise (CMPSB), wortweise (CMPSW) oder doppelwortweise (CMPSD) verglichen werden soll.

Die eigentlichen Operanden des Befehls CMPS werden nicht explizit angegeben. Vielmehr vergleicht CMPS in allen drei Variationen einen String, dessen Adresse in DS:SI (DS:ESI bei CMPSD) verzeichnet ist, mit einem String, dessen Adresse in ES:DI (ES:EDI) steht. Der Vergleich erfolgt wie bei CMP durch eine Subtraktion der Bytes, Worte oder Doppelworte aus DS:DI (DS:EDI) von denen in ES:SI (ES:ESI). Das Ergebnis der Subtraktionen wird auch hier verworfen – wie bei CMP werden nur die Flags anhand des Ergebnisses gesetzt.

CMPS vergleicht pro Durchgang nur jeweils ein Datum. Stringweises Vergleichen wird daher erst in Verbindung mit einem der Präfixe REPc möglich. Diese Präfixe bewirken, daß CMPS so lange ausgeführt wird, bis ein Abbruchkriterium erfüllt ist (siehe dort).

Damit diese Präfixe korrekt arbeiten können, verändert CMPS auch den Inhalt der Register SI und DI. So wird nach dem Vergleich in Abhängigkeit vom Status des Direction-Flags sowie von der Art des Vergleichs der Inhalt dieser Register um ein Byte (CMPSB), zwei Bytes (CMPSW) oder vier Bytes (CMPSD) erhöht (DF = 0) oder verringert (DF = 1). Auf diese Weise zeigen die Indexregister SI und DI nach dem Vergleich auf das nächste Datum!

Takte

#	8086	80286	80386	80486	Pentium
1	22	8	10	8	5
2	22	8	10	8	5
3	-	-	10	8	5

Opcodes

#	B1	Bemerkungen
1	A6	
2	A7	
3	A7	mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Vor der Verwendung von CMPS müssen die Register DS, SI (ESI), ES und DI (EDI) mit den Adressen der Strings geladen werden.

Beschreibung Seite 70

CMPXCHG *80486*

Funktion Vergleich zweier Operanden mit anschließendem Austausch.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Vergleich gesetzt.

Verwendung

#	Parameter	Beispiel
1	r8, r8	CMPXCHG CH, BL
2	m8, r8	CMPXCHG BVar, CH
3	r16, r16	CMPXCHG CX, BX
4	m16, r16	CMPXCHG WVar, DX
5	r32, r32	CMPXCHG EDX, EBX
6	m32, r32	CMPXCHG DVar, EDX

Arbeitsweise

CMPXCHG führt einen Vergleich wie CMP durch. Allerdings werden hierbei drei Operanden verwendet. Zwei von ihnen werden explizit übergeben (s.o.), der andere ist implizit vorgegeben. Bei ihm handelt es sich um den Akkumulator, also AL (8 Bits), AX (16 Bits) oder EAX (32 Bits).

CMPXCHG vergleicht nun zunächst den Akkumulator mit dem ersten Operanden. So würde z.B. durch CMPXCHG CH, BL der in AL stehende Wert mit dem in CH verglichen. Das weitere Verhalten hängt vom Ergebnis dieses Vergleichs ab:

- Sind die Werte im Akkumulator und im ersten Operanden (»Ziel«) gleich, so wird der Inhalt des zweiten Operanden (»Quelle«) in den ersten (»in das Ziel«) kopiert. Der Inhalt von der Quelle bleibt unverändert. Im obigen Beispiel würden sich dann in CH und BL identische Werte, nämlich der Inhalt von BL befinden.
- Unterscheiden sich die Werte im Akkumulator und im ersten Operanden, so wird der Inhalt vom Ziel, also vom zweiten Operanden, in den Akkumulator kopiert. In diesem Fall bleiben die Inhalte der explizit angegebenen Operanden unverändert, lediglich der Akkumulator wird aktualisiert. Im Beispiel oben fänden sich also identische Werte in AL und CH wieder.

Man könnte das Beispiel CMPXCHG CH, BL also wie folgt in Assembler emulieren:

```
    cmp   al, ch    ; Vergleich Akku/Ziel
    je    L1        ; Werte gleich → Sprung
    mov   al, ch    ; Kopie Ziel → Akku
    jmp   L2        ; fertig!
L1: mov   ch, bl    ; Kopie Quelle → Ziel
L2:                 ; weiter im Text!
```

Die Flags werden nach diesem Befehl wie bei CMP verändert. Ob AL, AX oder EAX als Akkumulator verwendet wird, entscheidet die Art der Operanden. Werden 8-Bit-Operanden verwendet, so ist AL der Akkumulator. Bei 16-Bit-Operatoren ist es AX, bei 32-Bit-Operatoren EAX.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	-	6/6*	6
2	-	-	-	7/10*	6
3	-	-	-	6/6*	6
4	-	-	-	7/10*	6
5	-	-	-	6/6*	6
6	-	-	-	7/10*	6

* Die ersten Werte gelten, falls der Vergleich erfolgreich war, andernfalls gelten die zweiten Werte.

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	0F	B0	/r		
2	0F	B0	/m	a16	
3	0F	B1	/r		
4	0F	B1	/m	a16	
5	0F	B1	/r		mit Präfix OPSIZE
6	0F	B1	/m	a16	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen

Leider hat sich in der Dokumentation von Intel zum 80486 ein kleiner Fehler eingeschlichen! So ist der Opcode von CMPXCHG nicht etwa, wie dort beschrieben, *0F – A6 / 0F – A7*, sondern, wie oben angegeben, *0F – B0 / 0F – B1*. Dieser Fehler wurde in alle mir bislang zur Verfügung stehenden Referenzen übernommen.

Selbst Microsoft und Borland haben diesen Fehler in ihren Programmen übernommen. So wird von den meisten Assemblern der Befehl CMPXCHG tatsächlich in den falschen Opcode übersetzt. Bislang konnte ich nur TASM Version 3.2 und MASM 6.0 ausmachen, die hier den korrekten Opcode erzeugen. Die integrierten Assembler unterstützen in der Regel die 80486-Befehle sowieso nicht, weshalb es hier nicht zu Problemen kommt.

Auch die Debugger, allen voran TD selbst in der Version 3.2, fangen mit der Befehlsfolge *0F – B0 – xx – xx* nichts an. Sie wird genau *so* disassembliert und angezeigt. Dagegen wird die *falsche* Folge *0F – A6 – xx – xx* mit *CMPXCHG xx, yy* disassembliert. Lassen Sie sich hierdurch nicht beeinflussen! Selbst wenn der Debugger *CMPXCHG xx, yy* anzeigt, führt ein Ausführen dieses Befehls zu unschönen Ergebnissen. Falls Sie unter DOS arbeiten und INT 06 nicht mit einem Interrupt-Handler belegt haben, so wird der Rechner abstürzen. Andernfalls werden Sie je nach Interrupt-Handler eine Fehlermeldung der Art »*invalid opcode*« erhalten, da der 80486 bei unbekannten Befehlen diesen Interrupt auslöst. Unter Windows 3.1 erhalten Sie die Fehlermeldung »*Diese Anwendung hat die Systemintegrität durch Ausführung eines ungültigen Befehls verletzt und wird abgebrochen ...*«.

Dagegen verkraftet der Prozessor nicht nur die korrekte Opcode-Sequenz *0F – B0 – xx – xx*, er führt auch den Befehl CMPXCHG aus. Selbst wenn sich Debugger hier im *Trace-Modus* und bei *Single Step* mit dem Disassemblieren etwas schwer tun!

Beschreibung

Seite 147

CMPXCHG8B *Pentium*

Funktion Vergleich zweier Operanden mit anschließendem Austausch.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									*						

Das Zero-Flag wird in Abhängigkeit vom Vergleich gesetzt.

Verwendung

#	Parameter	Beispiel
	m64	CMPXCHG8B QVar

Arbeitsweise CMPXCHG8B führt einen Vergleich wie CMP durch. Allerdings werden hierbei drei Operanden verwendet. Einer von ihnen wird explizit übergeben (s.o.), die beiden anderen sind implizit vorgegeben, EDX:EAX und ECX:EBX.

CMPXCHG8B vergleicht analog zu CMPXCHG beim 80486 zunächst EDX:EAX mit dem Operanden. Das weitere Verhalten hängt nun vom Ergebnis dieses Vergleichs ab:

- Sind die Werte in EDX:EAX und im Operanden (»Ziel«) gleich, so wird der Inhalt des zweiten impliziten Operanden (ECX:EBX) in den Operanden (»in das Ziel«) kopiert. Der Inhalt von EDX:EAX bleibt unverändert.
- Unterscheiden sich die Werte in EDX:EAX und im Operanden, so wird der Inhalt von »Ziel«, also vom Operanden, in EDX:EAX kopiert.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	10

Opcodes

#	B1	B2	B3	B4
	0F	C7	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	3, 8, 20	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
#UD	-	6	-	6	6

Beschreibung Seite 147

CPUID — Pentium

Funktion: Feststellung des Typs der CPU.

Flags:

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung:

#	Parameter	Beispiel
	keine	CPUID

Arbeitsweise: Dieser Befehl erwartet in EAX eine Steuerzahl, anhand derer er Informationen zur CPU zurückgibt.

Wird in EAX 0 übergeben, so antwortet der Prozessor mit folgender Registerbelegung:

EAX	EBX	ECX	EDX
max	vs #1	vs #2	vs #3

Hierbei bedeutet *max* eine Zahl > 0, die angibt, welchen maximalen Wert als Steuercode in EAX der Prozessor versteht. Beim Pentium ist *max* = 1 beim Pentium Pro ist *max* = 2, so daß dem Befehl CPUID die Steuercodes 0 und 1, im Falle des Pentium Pro auch 2 übergeben werden können. *vs* steht für *vendor string* und codiert einen herstellerspezifischen Identifikationsstring. Beim Pentium und Pentium Pro enthält EBX das Doppelwort 0756E6547h, EDX 049656E69h und ECX 06C65746Eh. Liest man nun die Registerinhalte in der Reihenfolge ECX:EDX:EBX, wobei wie bei Intel üblich bei Byte 0 von EBX begonnen und zu hohen Adressen gelesen werden muß (also rückwärts lesen), so codiert diese Registerkombination den String »*GenuineIntel*« (G = ASCII 047h, e = ASCII 065h, etc.).

Wird dem Pentium/Pentium Pro der Steuercode 1 übergeben, so antwortet er mit folgender Registerbelegung:

EAX	EBX	ECX	EDX
id	0	0	ff

id stellt hierbei das wichtigste Byte dar. Der Wert codiert

- in Bits 3 bis 0 die *Stepping-ID*,
- in Bits 7 bis 4 die Modellnummer und
- in Bits 11 bis 8 die Familie.
- Bits 13 und 12 sind beim Pentium reserviert; beim Pentium Pro wird der Prozessortyp angegeben: 00b steht für *original OEM processor*, 01b für *Intel overdrive processor*, 10b für *dual processor* (nicht in Kombination mit 80386 und 80486), und 11b ist reserviert.
- Bits 31 bis 14 sind reserviert.

Die *Stepping-ID* ist eine Revisionsnummer des Herstellers. Als Modellnummer wird eine Codezahl für die CPU innerhalb der Familie übergeben. Sie beginnt mit 1 für die ersten ausgelieferten Prozessoren. Der Eintrag im Familienfeld ist beim Pentium 5, beim Pentium Pro 6.

Werden dem Befehl Werte größer als *max* übergeben, so sind die Registerinhalte nicht definiert bzw. reserviert!

Die Inhalte von EBX und ECX sind ebenfalls reserviert.

ff, die *feature flags*, geben im Falle des gesetzten Zustands Auskunft über:

- FPU (Floating Point Unit on chip): die Anwesenheit einer Floating-Point-Unit auf dem Chip: Bit 0;
- VME (Virtual 8086 Mode Enhancements): der Prozessor unterstützt verschiedene virtuelle 8086-Modi: Bit 1;
- DE (Debugging Extensions): der Prozessor unterstützt verschiedene Maßnahmen zum Debuggen: Bit 2;
- PSE (Page Size Extensions): der Prozessor unterstützt 4-MByte-Pages: Bit 3;
- TSC (Time Stamp Counter): der Prozessor unterstützt den Befehl RDTSC: Bit 4;
- MSR (Model Specific Registers): der Prozessor unterstützt den Befehl RDMSR; Bit 5;
- PAE (Physical Address Extension): der Prozessor unterstützt physikalische Adressen, die größer als 32 Bits sind: Bit 6;
- MCE (Machine Check Exception): die Möglichkeit zu einer *Machine Check Exception*: Bit 7;
- CX8: die Unterstützung des CMPXCHG8B-Befehls in Bit 8.
- APIC: der Prozessor enthält den *Advanced Programmable Interrupt-Controller* (APIC) auf dem Chip, er wurde aktiviert und kann genutzt werden; *nur ab Pentium Pro*; Bit 9;
- Bit 10 ist reserviert;
- SEP (Sys Enter Present): Der Prozessor unterstützt Fast System Call; *nur ab Pentium II*; Bit 11;
- MTRR (Memory Type Range Registers): der Prozessor unterstützt bestimmte Register (MTRRs); *nur ab Pentium Pro;* Bit 12
- PGE-PTE; der Prozessor unterstützt bestimmte Flags, die vom *Translation Lookaside Buffer* (TLB) benutzt werden; *nur ab Pentium Pro*; Bit 13;
- MCA (Machine Check Architecture); *nur bei Pentium Pro*; Bit 14;
- CMOV: der Prozessor unterstützt den Befehl CMOVcc; *nur ab Pentium Pro*; Bit 15;
- die Bits 22 bis 16 sind reserviert.
- MMX (MMX Technology on Chip): der Prozessor unterstützt die MMX-Technologie; *nur bei MMX-Rechnern*; Bit 23;
- Die Bits 31 bis 24 sind reserviert.

Wird ab dem Pentium Pro der Steuercode 2 in EAX beim Aufruf von CPUID übergeben, so übergibt er in EAX, EBX, ECX und EDX Informationen zum Cache. Bitte beachten Sie folgende Ausnahmen:

- Bit 31 jedes Registers gibt an, ob die zurückgegebenen Daten gültig (Bit 31 = 0) oder reserviert (Bit 31 = 1) sind. Alle weiteren Daten sind Byte-Werte (Deskriptoren), sofern sie gültig sind. Das heißt, alle Register müssen byteweise interpretiert werden.
- Die unteren 8 Bits (0 bis 7) in EAX (=AL!) geben die Anzahl von Aufrufen des Befehls CPUID mit dem Wert 2 in EAX zurück, die notwendig sind, um vollständige Informationen zu erhalten. Bei der Pentium-Pro-Familie ist der Wert 1.
- Die möglichen Deskriptorwerte sind:

00h	null descriptor
01h	instruction TBL: 4 kByte pages, 4-way set associative, 64 entries
02h	instruction TBL: 4 MByte pages, 4-way set associative, 4 entries
03h	data TBL: 4 kByte pages, 4-way set associative, 64 entries
04h	data TBL: 4 MByte pages, 4-way set associative, 4 entries
06h	instruction cache: 8 kByte, 4-way set associative, 32 byte line size
0Ah	data cache: 8 kByte, 2-way set associative, 32 byte line size
41h	unified cache: 128 kByte, 4-way set associative, 32 byte line size
42h	unified cache: 256 kByte, 4-way set associative, 32 byte line size
43h	unified cache: 512 kByte, 4-way set associative, 32 byte line size

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	14

Opcodes

#	B1	B2
	0F	A2

Exceptions Keine

Bemerkungen Intel empfiehlt, auf jeden Fall den *Vendor-String* zu decodieren, um sicherzugehen, daß die *Feature-Flags* richtig interpretiert werden. Offensichtlich tun sich hier Ansätze zu Inkompatibilitäten auf.

Allerdings ist es sowieso richtig, CPUID zunächst mit 0 in EAX aufzurufen, um über den Rückgabewert in EAX festzustellen, wie viele Steuercodes verfügbar sind.

Ob CPUID unterstützt wird, können Sie über Bit 21 des EFlagregisters eruieren: Kann dieses umgeschaltet werden, wird CPUID unterstützt.

Beschreibung Seite 147, 149

CWD 8086

Funktion Erweiterung eines Worts zu einem Doppelwort.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	CWD

Arbeitsweise CWD konvertiert ein Word in AX in ein Doppelwort in DX:AX. Dies erfolgt, indem das Bit 15 des Wortes in AX in die Bits 0 bis 15 in DX kopiert wird. Auf diese Weise ist sichergestellt, daß ein vorzeichenbehaftetes Wort korrekt in ein ebenfalls vorzeichenbehaftetes Doppelwort expandiert wird.

Takte

#	8086	80286	80386	80486	Pentium
	5	2	2	3	2

Opcodes

#	B1
	99

Exceptions Keine

Beschreibung Seite 78

CWDE 80386

Funktion Erweiterung eines Worts zu einem Doppelwort.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	CWDE

Arbeitsweise CWDE arbeitet wie CWD, konvertiert also ein Wort in ein Doppelwort. Der Unterschied zu CWD liegt darin, daß dieser Befehl die Registerkombination DX:AX zur Darstellung des Doppelwortes verwendet, CWDE jedoch das Register EAX, das ab den 80386-Prozessoren ein gesamtes 32-Bit- Doppelwort aufnehmen kann.

Takte	#	8086	80286	80386	80486	Pentium
		-	-	3	3	3

Opcodes	#	B1	Bemerkungen
		98	mit Präfix OPSIZE

Beschreibung Seite 138

DAA *8086*

Funktion Dieser Befehl dient zur Korrektur des Ergebnisses einer Addition mittels ADD, falls die beiden addierten Werte gültige gepackte BCDs waren.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Korrektur gesetzt, das Overflow-Flag ist undefiniert.

Verwendung

#	Parameter	Beispiel
	keine	DAA

Arbeitsweise DAA prüft zunächst, ob das Auxiliary-Flag (durch die vorhergehende Addition) gesetzt wurde. Ist dies der Fall oder ist das untere Nibble (die Bits 3 bis 0) des Wertes in AL größer als 9, so wird zum Wert in AL 6 addiert und das Auxiliary-Flag gesetzt. Andernfalls wird nur das Flag gelöscht! Im zweiten Durchgang wird wiederum geprüft, ob das Auxiliary-Flag (durch die eben stattgefundene Korrektur) gesetzt ist oder der Wert in AL größer als $9F ist (das obere Nibble codiert ja eine weitere Ziffer, muß also ebenfalls geprüft werden). Auch hier wird dann das Auxiliary-Flag gesetzt, falls eine der beiden Bedingungen erfüllt ist. Zum Wert in AL wird dann noch $60 addiert. Sind beide Bedingungen nicht erfüllt, wird das Auxiliary-Flag gelöscht. Ein gesetztes Auxiliary-Flag nach DAA zeigt also eine durchgeführte Korrektur an. Das AH-Register wird nicht verändert.

Takte	#	8086	80286	80386	80486	Pentium
		4	3	4	2	3

Opcodes	#	B1
		27

Exceptions Keine

Bemerkungen Mit diesem Befehl können nur gepackte BCDs bearbeitet werden. Für ungepackte BCDs steht der Befehl AAA zur Verfügung. DAA arbeitet nur *nach* einer Addition korrekt, da er das Auxiliary-Flag auswertet, mit dem ein Dezimalüberlauf signalisiert wird. Für Subtraktionen existiert der Befehl DAS, für Multiplikationen und Divisionen gibt es keine Korrekturbefehle!

Beschreibung Seite 77

DAS *8086*

Funktion Dieser Befehl dient zur Korrektur des Ergebnisses einer Subtraktion mittels SUB, falls die beiden subtrahierten Werte gültige gepackte BCDs waren.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Korrektur gesetzt, das Overflow-Flag ist undefiniert.

Verwendung

#	Parameter	Beispiel
	keine	DAS

Arbeitsweise DAS prüft zunächst, ob das Auxiliary-Flag (durch die vorhergehende Subtraktion) gesetzt wurde. Ist dies der Fall oder ist das untere Nibble (die Bits 3 bis 0) des Wertes in AL größer als 9, so wird vom Wert in AL 6 subtrahiert und das Auxiliary-Flag gesetzt. Andernfalls wird nur das Flag gelöscht. Im zweiten Durchgang wird nun wiederum geprüft, ob das Auxiliary-Flag (durch die eben stattgefundene Korrektur) gesetzt ist oder der Wert in AL größer als $9F ist (das obere Nibble codiert ja eine weitere Ziffer, muß also ebenfalls geprüft werden). Auch hier wird dann das Auxiliary-Flag gesetzt, falls eine der beiden Bedingungen erfüllt ist. Vom Wert in AL wird dann noch $60 subtrahiert. Sind beide Bedingungen nicht erfüllt, wird das Auxiliary-Flag gelöscht. Ein gesetztes Auxiliary-Flag nach DAS zeigt also eine durchgeführte Korrektur an! Das AH-Register wird nicht verändert.

Takte

#	8086	80286	80386	80486	Pentium
	4	3	4	2	3

Opcodes

#	B1
	2F

Exceptions Keine

Bemerkungen Mit diesem Befehl können nur gepackte BCDs bearbeitet werden. Für ungepackte BCDs steht der Befehlt AAS zur Verfügung. DAS arbeitet nur *nach* einer Subtraktion korrekt. Für Divisionen und Multiplikationen existiert kein Korrekturbefehl, für Additionen der Befehl DAA.

Beschreibung Seite 77

DEC *8086*

Funktion Verringerung eines Operanden um 1.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		

Die Flags werden in Abhängigkeit vom Ergebnis der Verringerung gesetzt. Achtung: Im Gegensatz zum SUB-Befehl wird bei DEC das Carry-Flag *nicht* verändert!

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	DEC AH	
2	m8	DEC BVar	
3	r16	DEC BX	
4	m16	DEC WVar	
5	r32	DEC EBX	ab 80386
6	m32	DEC DVar	ab 80386

Arbeitsweise DEC verringert den Inhalt des Operanden um 1.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	15+EA	7	6	3	3
3	3	2	2	1	1
4	15+EA	7	6	3	3
5	-	-	2	1	1
6	-	-	6	3	3

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	FE	/1			
2	FE	/1	a16		
3	48+i				*
4	FF	/1			mit Präfix OPSIZE
5	48+i				mit Präfix OPSIZE *
6	FF	/1	a16		mit Präfix OPSIZE

* i kann Werte zwischen 0 und 7 annehmen und definiert das zu verwendende Register. Die Register werden hierbei wie folgt codiert:

	0	1	2	3	4	5	6	7
r16	AX	CX	DX	BX	SP	BP	SI	DI
r32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 62

DIV *8086*

Funktion Division zweier vorzeichenloser Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				?	?		?		?		?

Die Flags sind nach diesem Befehl undefiniert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	DIV BL	AL implizit!
2	m8	DIV BVar	AL implizit!
3	r16	DIV BX	AX implizit
4	m16	DIV WVar	AX implizit
5	r32	DIV EBX	ab 80386
6	m32	DIV DVar	ab 80386

Arbeitsweise DIV führt eine Division aus, bei der die Werte der Operanden vorzeichenlos interpretiert werden. Dies bedeutet, daß das höchstwertige Bit (Bit 31, 15 bzw. 7) *nicht* als Vorzeichen gewertet wird!

Der erste Operand (»Ziel«; Dividend) wird nur implizit angegeben und bezeichnet *immer* den Akkumulator. »Implizit« bedeutet hierbei, daß der Akkumulator nicht als Operand angegeben wird (werden darf; sonst erzeugt der Assembler eine Fehlermeldung), jedoch tatsächlich vorhanden ist. Der zweite Operand (also der einzige tatsächlich angegebene, die »Quelle«) enthält den Divisor.

DIV führt eine sogenannte *Integerdivision* aus, also eine Division, bei der als Ergebnis ein Quotient und ein Divisionsrest entsteht. (Beide Werte sind keine gebrochenen Zahlen, also Integer; daher der Name Integerdivision! Da DIV das Vorzeichen nicht berücksichtigt, handelt es sich hier also um eine »unechte« Integerdivision, da Integer üblicherweise vorzeichenbehaftet sind!)

Die Größe des explizit übergebenen Operanden gibt automatisch die verwendeten Register/Registerkombinationen vor:

Operand	Dividend	Quotient	Rest
Byte	AX	AL	AH
Wort	DX:AX	AX	DX
DWort	EDX:EAX	EAX	EDX

So legt beispielsweise der Befehl *DIV EinByte* aufgrund der Operandengröße von 1 Byte des (expliziten) Operanden *EinByte* folgende Rahmenbedingungen für die Division fest:

- Der Dividend muß als Wort in AX angegeben werden. Sollen Bytes als Dividenden zur Verwendung kommen, so müssen diese vor der Division (z.B. durch CBW) auf Wortgröße gebracht werden.
- Der Divisor hat Bytegröße (weil er mit dem explizit angegebenen Operanden identisch ist). Somit haben auch der Quotient und der Divisionsrest Bytegröße.
- Das Ergebnis der Division, der Quotient, findet sich im Byteregister AL wieder, der Divisionsrest im Byteregister AH.

Bei den ab dem 80286 möglichen Divisionen von 32-Bit-Dividenden müssen diese in der Registerkombination DX:AX vorliegen, wobei das höherwertige Wort in DX, das niederwertige Wort in AX stehen muß. Auch hier muß bei Verwendung von 16-Bit-Dividenden ggf. mit CWD ein Wort auf DWort-Größe gebracht werden. Analoges gilt für 64-Bit-Divisionen bei 80386ern ff.

Eine Division durch den Divisor »0« ist mathematisch nicht erlaubt und wird somit abgefangen! In einem solchen Fall wird die Division nicht durchgeführt. Statt dessen erfolgt ein Aufruf des Interrupts $00. Dieser Interrupt sollte mit einer Interruptroutine belegt sein, die Divisionen durch 0 behandelt. Die meisten Hochsprachencompiler verwenden hierzu eine Routine, die lediglich die Meldung »Division durch 0« ausgibt und das Programm anhält.

Takte

#	8086	80286	80386	80486	Pentium
1	80-90	14	14	16	17
2	(80-96)+EA	17	17	16	17
3	144-162	22	22	24	25
4	(150-186)+EA	25	25	24	25
5	-	-	38	40	41
6	-	-	41	40	41

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	F6	/6			
2	F6	/6	a16		
3	F7	/6			

4	F7	/6	a16	
5	F7	/6		mit Präfix OPSIZE
6	F7	/6	a16	mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#DE	-	2, 3	-	2, 3	2, 3
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./
#SS	0	1	0	1	1

Bemerkungen

Beachten Sie bitte, daß das Ergebnis einer Division in die vorgegebenen Register passen muß. Ist dies nicht der Fall, so wird ebenfalls ein Interrupt Int$00 ausgelöst.

So würde z.B. der Befehl DIV BL trotz absolut richtiger Anwendung diesen Interrupt auslösen, wenn in AX der Wert 513 und in BL der Wert 2 steht. Der Grund dafür liegt lediglich darin, daß 513 ÷ 2 = 256, Rest 1 ist und das Divisionsergebnis 256 nicht in das Byte-Register AL paßt.

Daher sollte selbst bei solchen Byte-Divisionen besser eine Division mit Wort-Werten erfolgen, also *DIV BX*, da nun das Ergebnis sicher in das Wortregister AX paßt. Beachten Sie in diesem Fall die Erweiterung des Dividenden und des Divisiors auf die korrekte Größe vor der Division z.B. mit CWD/CBW.

Beschreibung

Seite 59

ENTER *80186*

Funktion

Erzeugung eines *Stackrahmens* für Parameter.

Flags

X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	i16, 0	ENTER 012h, 0
2	i16, 1	ENTER 034h, 1
3	i16, i8	ENTER 056h, 008h

Arbeitsweise

ENTER erzeugt einen sogenannten Stackrahmen, wie er von Routinen in Hochsprachen verwendet und verlangt wird. ENTER ist ein Befehl, der praktisch die folgende Befehlssequenz ersetzt:

```
push      bp
mov       bp, sp
sub       sp, Konstante
```

wobei *Konstante* der über den ersten Operanden übergebene Wert ist und die Anzahl von Bytes angibt, die für lokale Variablen reserviert werden soll.

Der zweite Parameter dient dazu, eine sogenannte Verschachtelungstiefe oder *Nesting Depth* anzugeben. Dies dient dazu, bei verschachtelten Routinen die Möglichkeit zu wahren, daß hierarchisch »tiefere« Routinen auch auf die lokalen Variablen der »höheren« zurückgreifen können. Die Arbeitsweise des Befehls ENTER läßt sich, *Nesting* berücksichtigt, wie folgt Pascal-ähnlich umschreiben:

```
NESTING := max(NESTING, 31);
push(BP);
TEMP := SP;
if NESTING <> 0 then
begin
   dec(NESTING);
   while NESTING <> 0 do
   begin
      dec(BP,2);
      push(BP);
      dec(NESTING);
   end;
   push(TEMP);
end;
BP := TEMP;
sub(SP, LOCALS);
```

Takte

#	8086	80286	80386	80486	Pentium
1	-	11	10	14	11
2	-	15	12	17	15
3	-	12+4·(n-1)	15+4·(n-1)	17+3·n	15+2·n

* n ist in diesem Fall identisch mit dem Wert, der als i8 übergeben wird und die *Nesting Depth* widerspiegelt.

Opcodes

#	B1	B2	B3	B4
1	C8	i16		0
2	C8	i16		1
3	C8	i16		i8

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#PF	?	1	?	1	./.
#SS	0	2	0	2	2

Bemerkungen Einige Hochsprachen wie z.B. C machen vom *Nesting* keinen Gebrauch, da sie keine lokalen (und somit verschachtelten) Routinen zulassen. In diesem Fall ist ENTER mit 0 als zweitem Operanden anzugeben. Pascal dagegen gestattet lokale Routinen. Hier kann ein Wert für das *Nesting* angegeben werden!

Beschreibung Seite 121

HLT *8086*

Funktion Anhalten des Prozessors.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	HLT

Arbeitsweise Halt stoppt die Ausführung von Befehlen durch den Prozessor. Dieser wird in einen *Stand-by*-Modus geschickt, in dem er lediglich überwacht, ob ein NMI (*Not Mascable Interrupt*), ein Interrupt oder ein *Reset* stattfindet. In diesen Fällen wird er reaktiviert.

Nach einem Interrupt oder NMI wird die Ausführung des Programms an der Stelle nach dem HLT-Befehl wieder aufgenommen.

Takte

#	8086	80286	80386	80486	Pentium
	∞	∞	∞	∞	∞

Opcodes

#	B1
	F4

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#GP	0	32	0	32	-

Beschreibung Seite 79

IDIV 8086

Funktion Integerdivision zweier vorzeichenbehafteter Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				?	?		?		?		?

Die Flags sind nach diesem Befehl undefiniert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	IDIV BL	AL implizit!
2	m8	IDIV BVar	AL implizit!
3	r16	IDIV BX	AX implizit
4	m16	IDIV WVar	AX implizit
5	r32	IDIV EBX	ab 80386
6	m32	IDIV DVar	ab 80386

Arbeitsweise IDIV führt eine »echte« Integerdivision aus, bei der die Werte der Operanden vorzeichenbehaftet interpretiert werden. Dies bedeutet, daß das höchstwertige Bit (Bit 31, 15 bzw. 7) sehr wohl als Vorzeichen gewertet wird! Zur weiteren Arbeitsweise siehe DIV.

Das Vorzeichen des Restes ist das gleiche wie das des Dividenden. Das heißt, bei der Integerdivision mit IDIV BL und 513 in AX und -4 in BL ergibt sich -128 in AL und +1 in AH, während die gleiche Division mit -513 in AX und 4 in BL zu -128 in AL und -1 in AH führt.

Takte

#	8086	80286	80386	80486	Pentium
1	101-112	17	19	19	22
2	(107-118)+EA	20	19	20	22
3	165-184	25	27	27	30
4	(171-190)+EA	28	27	28	30
5	-	-	43	43	46
6	-	-	43	44	46

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	F6	/7			
2	F6	/7	a16		
3	F7	/7			
4	F7	/7	a16		
5	F7	/7			mit Präfix OPSIZE
6	F7	/7	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#DE	-	2, 3	-	2, 3	2, 3
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./
#SS	0	1	0	1	1

Bemerkungen Siehe Bemerkungen zu DIV!

Beschreibung Seite 59

IMUL *8086*

Funktion Integermultiplikation zweier Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				?	?		?		?		*

Die Flags *Overflow* und *Carry* werden in Abhängigkeit vom Ergebnis der Multiplikation gesetzt, alle anderen sind undefiniert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	IMUL BL	AL implizit!
2	m8	IMUL BVar	AL implizit!
3	r16	IMUL BX	AX implizit
4	m16	IMUL WVar	AX implizit
5	r32	IMUL EBX	ab 80386
6	m32	IMUL DVar	ab 80386
7	r16, i8	IMUL BX, 012h	ab 80186
8	r16, r16, i8	IMUL BX, CX, 012h	ab 80186
9	r16, m16, i8	IMUL CX, WVar, 012h	ab 80186
10	r16, r16, i16	IMUL DX, AX, 01234h	ab 80186
11	r16, m16, i16	IMUL SI, WVar, 01234h	ab 80186
12	r16, i16	IMUL CX, 01234h	ab 80186
13	r16, r16	IMUL BX, AX	ab 80386
14	r16, m16	IMUL DX, WVar	ab 80386
15	r32, i8	IMUL ECX, 012h	ab 80386
16	r32, i32	IMUL EDI, 012345678h	ab 80386
17	r32, r32	IMUL ECX, EBX	ab 80386

18	r32, m32	IMUL EBX, DVar	ab 80386
19	r32, r32, i8	IMUL ECX, EDX, 012h	ab 80386
20	r32, m32, i8	IMUL EDI, DVar, 034h	ab 80386
21	r32, r32, i32	IMUL ESI, EDI, 012345678h	ab 80386
22	r32, m32, i32	IMUL EDX, DVar, 0123456h	ab 80386

Arbeitsweise

IMUL führt eine »echte« Integermultiplikation aus, bei der die Werte der Operanden vorzeichenbehaftet interpretiert werden. Dies bedeutet, daß das höchstwertige Bit (Bit 31, 15 bzw. 7) sehr wohl als Vorzeichen gewertet wird! Zur weiteren Arbeitsweise siehe MUL!

IMUL hat ab dem 80286 als einziger »Rechen«-Befehl eine erhebliche Erweiterung erfahren:

- Es können nun explizit angebbare Register mit Konstanten multipliziert werden (Verwendung #7, ab 80386 auch #12, #15 und #16). In dieser Form hat der IMUL-Befehl zwei explizit anzugebende Operatoren, von denen der erste ein Register und der zweite eine Konstante sein muß.
- Konstanten können auch mit dem Inhalt von Registern oder Variablen multipliziert werden, wobei das Produkt nicht in den verwendeten Operanden, sondern in einen zusätzlich anzugebenden Operanden eingetragen wird (Verwendung #8, #9, #10, #11, ab 80386 auch #19, #20, #21, #22). In dieser Form hat der IMUL-Befehl drei Operanden, wobei der erste (»Ziel«) ein Register sein muß; in dieses wird das Produkt des IMUL-Befehls eingetragen. Operand 2 bezeichnet ein Register oder eine Speicherstelle; der dort verzeichnete Inhalt (»Quelle«) ist Multiplikand des IMUL-Befehls und wird durch ihn nicht verändert. Schließlich wird als dritter Operand die Konstante angegeben, die als Multiplikator dient.
- Schließlich wurde der IMUL-Befehl ab dem 80386 verallgemeinert: So kann als Multiplikand einer Multiplikation jedes der 16- oder 32-Bit-Register dienen (Verwendung #13, #14, #17, #18). Diese Befehle stellen praktisch die Verallgemeinerung der Befehle #3, #4, #5 und #6 dar. Hier wird das Zielregister als erster Operand explizit angegeben, während die Quelle als zweiter Operand übergeben wird. Allerdings kann auch hier nur ein Register das Ziel sein! Im Unterschied zu den anderen Erweiterungen wird hierbei der Inhalt des ersten Operanden überschrieben!

Mit diesen Erweiterungen hat sich bei diesen Befehlen auch die Bedeutung des Carry- und des Overflow-Flags etwas geändert. Während bei den »normalen« IMUL-Befehlen (#1 bis #6) ein Überlauf bei der Multiplikation nicht möglich war (das Produkt aus zwei Bytes ist immer in einem Wort darstellbar!), kann es nun durchaus zu Überlaufproblemen kommen. Wenn nämlich der Inhalt eines Wortregisters mit einer Wortvariablen multipliziert wird, kann dabei ein Produkt herauskommen, das nicht mehr in ein 16-Bit-Register paßt (vgl. Verwendung #14), um so weniger, wenn dazu noch eine Konstante berücksichtigt wird (vgl. Verwendung #10).

Aus diesem Grunde signalisieren diese Flags nicht mehr lediglich redundant, daß die höherwertige Hälfte des Ergebnisses 0 ist. Vielmehr zeigen ein gelöschtes Carry- und Overflow-Flag nun an, daß das Ergebnis tatsächlich in das Ziel paßt. Sie haben also ihre eigentliche Funktion in diesen Fällen zurückerhalten.

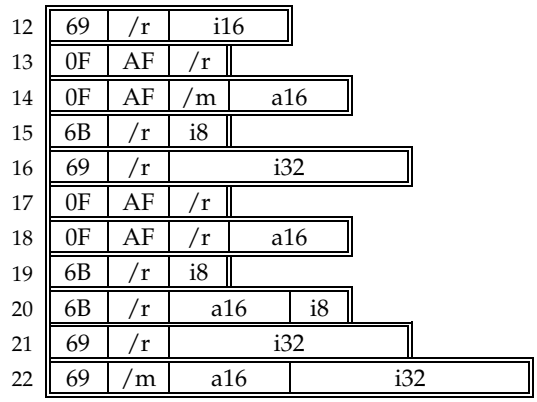
Nichtsdestoweniger gilt bei der Verwendung von #1 bis #6 aus Kompatibilitätsgründen das, was bei dem Befehl MUL gesagt wurde.

Takte

#	8086	80286	80386	80486	Pentium
1	101-112	17	19	19	11
2	(107-118)+EA	20	19	20	11
3	165-184	25	27	27	11
4	(171-190)+EA	28	27	28	11
5	-	-	43	43	10
6	-	-	43	44	10
7	-	21	9-14	13-26	10
8	-	21	9-14	13-26	10
9	-	24	12-17	13-26	10
10	-	21	9-22	13-26	10
11	-	24	12-25	13-26	10
12	-	-	9-22	13-26	10
13	-	-	9-22	13-26	10
14	-	-	12-25	13-26	10
15	-	-	9-14	13-42	10
16	-	-	9-38	13-42	10
17	-	-	9-38	13-42	10
18	-	-	12-41	13-42	10
19	-	-	9-14	13-42	10
20	-	-	12-17	13-42	10
21	-	-	9-38	13-42	10
22	-	-	12-41	13-42	10

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	F6	/5			
2	F6	/5	a16		
3	F7	/5			
4	F7	/5	a16		
5	F7	/5			mit Präfix OPSIZE
6	F7	/5	a16		mit Präfix OPSIZE
7	6B	/r	i8		
8	6B	/r	i8		
9	6B	/m	a16	i8	
10	69	/r	i16		
11	69	/m	a16	i16	



Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe Bemerkungen zu MUL!

Beschreibung Seite 57

IN *8086*

Funktion Auslesen eines Ports.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	AL, i8	IN AL, 012h	
2	AX, i8	IN AX, 012h	
3	EAX, i8	IN EAX, 012h	ab 80386
4	AL, DX	IN AL, DX	
5	AX, DX	IN AX, DX	
6	EAX, DX	IN EAX, DX	ab 80386

Arbeitsweise

IN liest einen spezifizierbaren Port aus. Ob ein Byte, ein Wort oder – ab dem 80386 – ein Doppelwort ausgelesen wird, wird durch die Angabe des Akkumulators als erstem Operanden gesteuert. So wird bei AL als Operanden ein Byte ausgelesen!

Die Angabe des Ports, der ausgelesen werden soll, kann auf zwei Arten erfolgen. Liegt die Portadresse im Bereich $00 bis $FF, so kann sie als Konstante direkt als zweiter Operand angegeben werden. Andernfalls muß sie im Register DX abgelegt werden. In diesem Fall ist der erweiterte Befehl von IN zu verwenden, bei dem DX als zweiter Operand fungiert.

Takte

#	8086	80286	80386	80486	Pentium
1	10	5	12	14	7
2	10	5	12	14	7
3	-	-	12	14	7
4	8	5	13	14	7
5	8	5	13	14	7
6	-	-	13	14	7

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	E4	i8			
2	E5	i8			
3	E5	i8			mit Präfix OPSIZE
4	EC				
5	ED				
6	ED				mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	34	0	48	-

Beschreibung

Seite 24

INC *8086*

Funktion

Erhöhung eines Operanden um 1.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		

Die Flags werden in Abhängigkeit vom Ergebnis der Erhöhung gesetzt. Achtung: Im Gegensatz zum ADD-Befehl wird bei INC das Carry-Flag *nicht* verändert!

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	INC AH	
2	m8	INC BVar	
3	r16	INC BX	
4	m16	INC WVar	
5	r32	INC EBX	ab 80386
6	m32	INC DVar	ab 80386

Arbeitsweise

INC erhöht den Inhalt des Operanden um 1.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	15+EA	7	6	3	3
3	3	2	2	1	1
4	15+EA	7	6	3	3
5	-	-	2	1	1
6	-	-	6	3	3

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	FE	/0			
2	FE	/0	a16		
3	40+i				*
4	FF	/0			mit Präfix OPSIZE
5	40+i				mit Präfix OPSIZE *
6	FF	/0	a16		mit Präfix OPSIZE

* i kann Werte zwischen 0 und 7 annehmen und definiert das zu verwendende Register. Die Register werden hierbei wie folgt codiert:

	0	1	2	3	4	5	6	7
r16	AX	CX	DX	BX	SP	BP	SI	DI
r32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung

Seite 62

INS 80186
INSB 80186
INSW 80186
INSD 80386

Funktion Auslesen eines Ports und Eintrag des Wertes in einen String.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	keine	INSB	
2a	keine	INSW	
3a	keine	INSD	ab 80386
1b	m8, DX	INS BString, DX	*
2b	m16, DX	INS WString, DX	*
3b	m32, DX	INS DString, DX	* ab 80386

*ACHTUNG! Bei der Verwendung der allgemeinen Form müssen mehrere Dinge beachtet werden:

- Durch die bloße Angabe des Pseudooperanden wird die Registerkombination ES:DI (ES:EDI) noch nicht korrekt besetzt! Dies hat unbedingt gesondert vor der Verwendung des Stringbefehls zu erfolgen!
- Der Assembler achtet auf die korrekte Angabe der Operanden. So muß als erster Operand der String angegeben werden, dessen Segmentanteil in ES steht. Als zweiter Operand wird nur DX erlaubt! Stimmt der Segmentanteil der Adresse nicht mit dem in ES stehenden überein, so erfolgt eine Fehlermeldung!
- Der verwendete String muß korrekt definiert worden sein, da nur über seine Definition der Assembler die benötigte Information über die zu verwendende Datumsgröße erhält! In Verwendung #1b muß daher eine Definition der Art

  ```
  BString DB 128 DUP (?)
  ```

  erfolgen. Durch die Anweisung DB kann der Assembler den Befehl *INS BString, DX* in den Befehl *INSB* übersetzen! Analoges gilt für die Verwendung #2b und #3b!

Arbeitsweise Mit INS ist ein Einlesen von Daten aus einem Port in einen String möglich. Als String bezeichnet man beim Assembler eine Folge von Variablen gleicher Größe. So gibt es *Byte-Strings*, *Wort-Strings* und *Doppelwort-Strings*. Die Größe der Strings ist keinen Beschränkungen unterworfen, wird aber durch das gewählte Speichermodell limitiert. So können im Real-Mode die Strings maximal 64 kByte entsprechend der Größe eines Segments sein.

INS ist der Überbegriff für die Befehle INSB, INSW und INSD und wird durch den Assembler in einen der drei Befehle übersetzt. INS hat deshalb zwei nur zur Assemblierung benötigte Pseudooperanden, die jedoch nur anzugeben haben, ob der Port byteweise (INSB), wortweise (INSW) oder doppelwortweise (INSD) ausgelesen werden soll.

Die eigentlichen Operanden des Befehls INS werden nicht explizit angegeben. Vielmehr liest INS in allen drei Variationen Daten aus dem Port, dessen Adresse in DX angegeben wird, in einen String, dessen Adresse in ES:DI (ES:EDI) steht.

INS liest pro Durchgang nur jeweils ein Byte, Wort oder Doppelwort in den String, der durch die Adresse in ES:DI adressiert wird. Stringweises Einlesen wird daher erst in Verbindung mit einem der Präfixe REPc möglich. Diese Präfixe bewirken, daß INS so lange ausgeführt wird, bis ein Abbruchkriterium erfüllt ist (siehe dort).

Damit diese Präfixe korrekt arbeiten können, verändert INS auch den Inhalt des Registers DI. So wird nach dem Einlesen in Abhängigkeit vom Status des Direction-Flags sowie von der Art und Größe des Datums der Inhalt dieses Registers um ein Byte (INSB), zwei Bytes (INSW) oder vier Bytes (INSD) erhöht (DF = 0) oder verringert (DF = 1). Auf diese Weise zeigt das Indexregister DI nach dem Einlesen *auf die nächste benutzte Speicherstelle!*

Takte

#	8086	80286	80386	80486	Pentium
1	-	5	15	17	9
2	-	5	15	17	9
3	-	-	15	17	9

Opcodes

#	B1	Bemerkungen
1	6C	
2	6D	
3	6D	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8, 20, 34	0	8, 48	8
#PF	?	1	?	1	./.

Bemerkungen Vor der Verwendung von INS müssen die Register DX, ES und DI (EDI) mit der Adresse des Strings und der Portnummer geladen werden.

Beschreibung Seite 122

INT *8086*
INTO *8086*

Funktion Dieser Befehl dient zur expliziten Interrupt-Auslösung.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
						0	0								

Durch den Interrupt-Befehl werden das Interrupt-Enable-Flag sowie das Trace-Flag explizit gelöscht.

Verwendung

#	Parameter	Beispiel
1	i8	INT 010h
2	keine	INT3
3	keine	INTO

Arbeitsweise INT löst einen sogenannten Software-Interrupt aus. Dies bedeutet, daß der Prozessor quasi ein Unterprogramm ausführt. INT besitzt in seiner allgemeinen Form einen Operanden, der die Nummer des auszuführenden Interrupts darstellt. Gültige Werte sind Zahlen im Bereich 0 bis 255.

Bei der Abarbeitung eines INT-Befehls simuliert der Prozessor einen Hardware-Interrupt, wie er vom *Interrupt-Controller* auf Anforderung bestimmter Hardwarekomponenten erzeugt wird. Hier wie dort sichert er zunächst den Inhalt des Flagregisters auf den Stack (führt also quasi ein PUSHF aus). Anschließend wird in einer vom Betriebssystem zur Verfügung gestellten Tabelle der Einsprungspunkt für eine Routine ermittelt, die bei dem gewählten Interrupt ausgeführt werden soll. Hierzu wird der im Operanden übergebene Wert als Index in diese *Interrupt Table* verwendet. Die dort verzeichnete Adresse der als *Interrupt-Handler* bezeichneten Routine wird dann in die Registerkombination CS:IP geladen, nicht bevor die Adresse des nach dem INT-Befehl stehenden Befehls als sogenannte Rücksprungadresse ebenfalls auf den Stack gebracht wird.

INT ist somit praktisch ein intersegmentaler CALL-Befehl mit einigen zusätzlichen Aktivitäten, der jedoch die Zieladresse nicht direkt oder indirekt als Parameter übergeben erhält, sondern sie aus einer genau definierten und spezifizierten Tabelle anhand eines Indexwertes selbst ermittelt.

Neben dem allgemeinen INT-Befehl gibt es noch zwei Sonderfälle: INT3 erwartet keinen zusätzlichen Operanden. Der Operand wird implizit durch den Opcode selbst übergeben. INT3 als Ein-Byte-Opcode wird z.B. bei Debuggern häufig verwendet.

Auch INTO erwartet keinen Operanden. Dieser Interrupt entspricht einem INT $04, allerdings ist seine Ausführung an die Prüfung des Overflow-Flags gebunden. INTO simuliert folgende Aktivitäten:

```
if overflow flag set then INT $04
```

Takte

#	8086	80286	80386	80486	Pentium
1	51	23	37	30	16
2	52	23	33	26	13
3 *	4/53	3/24	3/35	3/28	4/13

* Der jeweils erste Wert gilt, falls das Overflow-Flag nicht gesetzt ist, der Interrupt also nicht ausgelöst wird. Ansonsten gilt der jeweils zweite Wert.

Opcodes

#	B1	B2
1	CD	i8
2	CC	
3	CE	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#GP	0	17	0	17, 31	8, 16
	?	4, 12, 16, 24, 30, 42, 47, 64	?	4, 12, 16, 24, 30, 42, 47	./.
#NP	?	1	?	1	./.
#PF	?	1	?	1	./.
#SS	0	5	0	-	3, 4
	?	4, 8	?	4, 7	-
#TS	?	1, 3, 4, 5, 6	?	1, 3, 4, 5, 6	./.

Bemerkungen Üblicherweise wird eine Routine, die als *Interrupt-Handler* durch einen INT-Befehl aufgerufen wird, mit IRET beendet.

INVD *80486*

Funktion Den Cache für ungültig erklären.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	INVD

Arbeitsweise INVD löscht den Inhalt des internen Caches und signalisiert externen Caches, den Inhalt ebenfalls zu löschen.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	4	15

Opcodes

#	B1	B2
	0F	08

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	32	0	2	-

Bemerkungen

INVD ist eine privilegierte Funktion. Das heißt, sie kann nur innerhalb der höchsten Privilegstufe (CPL = 0) ausgeführt werden. Dies ist ausschließlich in Betriebssystemmodulen der Fall.

INVD wartet nicht darauf, daß externe Caches ihren Inhalt tatsächlich gelöscht haben. Nach dem Löschen des internen Caches wird die Programmausführung sofort fortgesetzt.

Die Daten in den Caches werden nicht gespeichert, bevor sie aus dem Cache gelöscht werden – dazu dient WBINVD. Daten, die vorher nicht gesichert wurden, gehen daher nach INVD unweigerlich verloren. Wenn es nicht ausdrücklich Sinn macht, die Daten vor einer Invalidierung des Caches *nicht* zu sichern, sollte daher besser WBINVD benutzt werden.

INVDPG *80486*

Funktion

Den *Page*-Eintrag im Translation Lookaside Buffer (TLB) für ungültig erklären.

Flags

X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	Adresse	INVDPG Adr

Arbeitsweise

Der Operand ist eine Adresse. INVDPG sucht im TLB die *Page*, in der diese Adresse steht und löscht dann den dazugehörigen TLB-Eintrag für diese *Page*.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	12	25

Opcodes

#	B1	B2	B3	B4	B5
	0F	01	/7	i16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#GP	0	32	0	2	./.
#UD	-	4	-	4	4

Bemerkungen INVDPG ist eine privilegierte Funktion, was bedeutet, daß sie nur auf höchster Privilegstufe (CPL = 0; nur in Betriebssystemmodulen) ausgeführt werden kann.

IRET *8086*
IRETD *80386*
IRETW *80386*

Funktion Abschluß einer Interrupt-Routine; Rücksprung ins unterbrochene Programm.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*	*	*	*	*	*		*		*		*

Die Flags werden in Abhängigkeit vom Zustand *vor* dem Interrupt-Aufruf gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	keine	IRET	
2	keine	IRETD	ab 80386
3	keine	IRETW	ab 80386

Arbeitsweise IRET bildet den Abschluß einer Routine, die durch einen INT-Befehl oder einen *Hardware-Interrupt* aufgerufen wird. IRET entnimmt dem Stack die vom INT-Befehl dort abgelegte Adresse des auf den INT-Befehl folgenden Befehls und lädt sie in die Registerkombination CS:IP, wodurch die Ausführung unmittelbar hinter dem aufrufenden INT-Befehl fortgesetzt wird. Zusätzlich restauriert IRET das Flagregister mit dem Wort, das der INT-Befehl ebenfalls auf den Stack gesichert hat.

IRETD und IRETW sind lediglich Abwandlungen des IRET-Befehls, die ab dem 80386 die Möglichkeit der 32-Bit-Adressierung im *Flat-Modell* ermöglichen oder in genau diesem Modell eine 16-Bit-Adressierung erzwingen.

Takte

#	8086	80286	80386	80486	Pentium
1	32	17	22	15	10
2	-	-	22	15	10
3	-	-	22	15	10

Opcodes

#	B1	Bemerkungen
1	CF	
2	CF	mit Präfix OPSIZE
3	CF	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	7, 10	0	10, 31	10
	?	14, 27, 35, 37, 39, 43, 45, 47, 64	?	-	./.
#NP	?	5	?	-	./.
#PF	?	1	?	1	./.
#SS	0	9	0	9	9

Jcc *8086*

Funktion Bedingter Sprung an eine neue Befehlsadresse.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	i8	JNE ShortLabel	
2	i16	JZ NearLabel	ab 80386
3	i32	JNAE NearLabel	ab 80386

Arbeitsweise Bei den bedingten Sprüngen handelt es sich um sogenannte *Relativsprünge*. Bei solchen Sprüngen wird die Zieladresse relativ zur aktuellen Adresse angegeben, also als Operand eine Konstante verwendet, die zur aktuellen Adresse addiert wird. Um Relativsprünge, die auch *Short Jumps* genannt werden, in beide Richtungen zu ermöglichen – vor und hinter die aktuelle Adresse – wird die Sprungdistanz (die Konstante) vorzeichenbehaftet interpretiert. Dies bedeutet, daß das jeweilige höchstwertige Bit (Bit 7 bei Bytes, ab 80386 Bit 15 bei Worten bzw. Bit 31 bei Doppelworten) als Vorzeichen interpretiert wird. Relativsprünge können daher maximal eine Distanz von 127 Bytes (bzw. 32 kByte bzw. 2 GByte) überwinden.

Wie der Name schon sagt, sind bedingte Sprünge an eine Bedingung geknüpft. So erfolgt der Relativsprung nur dann, wenn die Bedingung erfüllt ist, andernfalls wird die Programmausführung mit dem folgenden Befehl fortgesetzt. Es gibt folgende bedingte Sprünge (zur Erklärung siehe Teil 1!):

- JA, JAE, JB, JBE, JNA, JNAE, JNB, JNBE; Bedingungen, die bei einem Vergleich vorzeichenloser Zahlen bestehen. Ausgewertet werden das Carry- und/oder Zero-Flag.
- JG, JGE, JL, JLE, JNG, JNGE, JNL, JNLE; Bedingungen, die bei einem Vergleich vorzeichenbehafteter Zahlen bestehen. Ausgewertet werden das Sign-, Zero- und/oder Overflow-Flag.
- JE, JNE, JNZ, JZ; Bedingungen, die bei einem Vergleich von Zahlen allgemein bestehen. Ausgewertet wird das Zero-Flag.
- JC, JNC, JNO, JNP, JNS, JO, JP, JPE, JPO, JS; Bedingungen, die einzelne Flags betreffen und von verschiedenen Befehlen erzeugt werden.
- JCXZ, JECXZ; Bedingungen, die das Count-Register betreffen (ab 80386 auch ECX!). Dies sind die einzigen bedingten Sprungbefehle, bei denen *nicht* der Flagzustand geprüft wird!

Takte

#	8086	80286	80386	80486	Pentium
	16/4	7/3*	7+m/3	3/1	1
jcxz	18/6	8/4	9+m/5	3/1	6/5

Die jeweils ersten Werte gelten für den Fall, daß die Bedingung erfüllt ist, also ein Sprung erfolgt. Der zweite Wert gilt für den Fall, daß die Bedingung nicht erfüllt ist, also der nächste Befehl ausgeführt wird.

Alle Taktangaben gelten für alle Sprungbefehle, unabhängig von der Bedingung und der Sprungweite (ab 80386!).

* Beim 80286 muß für jedes Byte des folgenden Befehls ein Takt addiert werden, falls ein Sprung erfolgt.

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	Bemerkungen
	77	i8						JA
	73	i8						JAE
	72	i8						JB
	76	i8						JBE
	72	i8						JC
	E3	i8						JCXZ
	66	E3	i8					JECXZ; ab 80386
	74	i8						JE
	7F	i8						JG
	7D	i8						JGE
	7C	i8						JL
	7E	i8						JLE
	76	i8						JNA
	72	i8						JNAE
	73	i8						JNB

77	i8	JNBE
73	i8	JNC
75	i8	JNE
7E	i8	JNG
7C	i8	JNGE
7D	i8	JNL
7F	i8	JNLE
71	i8	JNO
7B	i8	JNP
79	i8	JNS
75	i8	JNZ
70	i8	JO
7A	i8	JP
7A	i8	JPE
7B	i8	JPO
78	i8	JS
74	i8	JZ

66	0F	87	i16	*i32*	JA; ab 80386*
66	0F	83	i16	*i32*	JAE; ab 80386*
66	0F	82	i16	*i32*	JB; ab 80386*
66	0F	86	i16	*i32*	JBE; ab 80386*
66	0F	82	i16	*i32*	JC; ab 80386*
66	0F	84	i16	*i32*	JE; ab 80386*
66	0F	8F	i16	*i32*	JG; ab 80386*
66	0F	8D	i16	*i32*	JGE; ab 80386*
66	0F	8C	i16	*i32*	JL; ab 80386*
66	0F	8E	i16	*i32*	JLE; ab 80386*
66	0F	86	i16	*i32*	JNA; ab 80386*
66	0F	82	i16	*i32*	JNAE; ab 80386*
66	0F	83	i16	*i32*	JNB; ab 80386*
66	0F	87	i16	*i32*	JNBE; ab 80386*
66	0F	83	i16	*i32*	JNC; ab 80386*
66	0F	85	i16	*i32*	JNE; ab 80386*
66	0F	8E	i16	*i32*	JNG; ab 80386*
66	0F	8C	i16	*i32*	JNGE; ab 80386*
66	0F	8D	i16	*i32*	JNL; ab 80386*
66	0F	8F	i16	*i32*	JNLE; ab 80386*
66	0F	81	i16	*i32*	JNO; ab 80386*
66	0F	8B	i16	*i32*	JNP; ab 80386*
66	0F	89	i16	*i32*	JNS; ab 80386*

66	0F	85	i16	*i32*	JNZ; ab 80386*
66	0F	80	i16	*i32*	JO; ab 80386*
66	0F	8A	i16	*i32*	JP; ab 80386*
66	0F	8A	i16	*i32*	JPE; ab 80386*
66	0F	8B	i16	*i32*	JPO; ab 80386*
66	0F	88	i16	*i32*	JS; ab 80386*
66	0F	84	i16	*i32*	JZ; ab 80386*

* Die kursiv gedruckten Bytefolgen beziehen sich auf die 32-Bit-Versionen des entsprechenden Befehls. Hier dient wie üblich das Präfix $66 als Unterscheidungsmerkmal, daß der folgenden 2-Byte-Opcode-Sequenz statt einer 16-Bit-Konstanten eine 32-Bit-Konstante folgt!

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	10	0	11	11

Bemerkungen Sollen bedingte Sprünge ausgeführt werden, deren Sprungziele weiter entfernt liegen als 127 Bytes (32 kByte; 2 GByte), so muß ein »Umweg« programmiert werden. Anstelle von

```
JZ FarLabel
```

muß der korrespondierende »gegenteilige« Sprungbefehl auf ein Label direkt hinter einem unbedingten Sprung erfolgen:

```
        JNZ Near
        JMP FarLAbel
Near:
```

Beschreibung Seite 33

JMP *8086*

Funktion Unbedingter Sprung an eine neue Befehlsadresse.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	i8	JMP ShortLabel	short; relativ
2	i16	JMP NearLabel	near; relativ
3	m16	JMP [WVar]	indirekt near

4	r16	JMP [BX]	indirekt near
5	i16:16	JMP FarLabel	far; absolut
6	m16:16	JMP [DVar]	indirekt far
7	i32	JMP NearLabel	ab 80386
8	m32	JMP [DVar]	ab 80386
9	r32	JMP [EAX]	ab 80386
10	i32:16	JMP FlatLabel	ab 80386
11	m32:16	JMP [Var]	ab 80386

Arbeitsweise

JMP verändert den Inhalt der Register IP bzw. CS:IP. Somit wird der Prozessor gezwungen, die Programmausführung an einer anderen Adresse weiterzuführen. JMP ist ein sogenannter *unbedingter Sprung*, da seine Ausführung im Gegensatz zu Jcc nicht an Bedingungen gebunden ist!

JMP hat verschiedene Möglichkeiten, die Registerinhalte zu verändern:

- Addition einer Byte- oder Wortkonstanten zum Inhalt von IP. Hierbei handelt es sich um sogenannte Relativsprünge, da die verwendete Konstante eine Distanz zur derzeitigen Adresse angibt. Daher ist der als Operand anzugebende Wert vorzeichenbehaftet zu interpretieren. Relativsprünge können nur innerhalb eines Segments ausgeführt werden und überwinden (wegen des Vorzeichenbits) maximal 128 bzw. 32.768 Bytes. Aus diesem Grunde heißen sie auch *Short Jumps* (bis 128 Bytes) bzw. *Near Jumps*. Siehe hierzu Verwendung #1, #2 und #7.

- Ersetzen des Inhalts von IP durch einen 16-Bit-Wert. Auf diese Weise wird ein anderer Adressenoffset in das *Instruction-Pointer-Register* eingetragen, was dazu führt, daß der Prozessor an der neuen Adresse seine Tätigkeit fortsetzt. Diese Sprünge sind somit absolute Sprünge. Sie können, da nur das IP-Register manipuliert wird, nur innerhalb des Segments erfolgen, weshalb sie auch als Near Jumps bezeichnet werden. Es spielt dabei keine Rolle, ob der Segmentoffset 16 Bits (Verwendung #3, #4) oder, ab dem 80386, 32 Bits groß ist (Verwendung #8, #9).

  Es ist bei den Near Jumps also absolut nicht unerheblich, woher die Adresse, die in IP eingetragen wird, kommt. Wird sie als Konstante angegeben (Verwendung #2, #7), so spricht man von *direkten* Sprüngen, da die Adresse direkt angegeben wird. Direkte Near Jumps sind Relativsprünge. Es ist jedoch auch möglich, die Adresse erst zur Laufzeit eines Programms zu berechnen. Sie muß dann entweder in ein Register (Verwendung #4, #9) oder in eine geeignete Speicherstelle (Verwendung #3, #8) eingetragen werden, aus der der Prozessor sie dann ausliest. Solche Sprünge heißen daher *indirekt*. Indirekte Near Jumps sind also Absolutsprünge.

- Werden die Inhalte von IP und CS verändert, so spricht man von *Far Jumps* oder *Intersegmentsprüngen*, da nun die zu ladende Adresse den gesamten adressierbaren Raum des Prozessors angeben kann. Diese Sprünge können ebenfalls entweder direkt ausgeführt werden (Verwendung #5, ab 80386

auch #10) oder indirekt (Verwendung #6, ab 80386 auch #9). Hierbei wird die Adresse als *vollständig qualifizierte* Adresse mit Segment- und Offsetanteil entweder als Konstante angegeben oder in einer Variablen, die als Operand übergeben wird. Die »Register«-Version ist hierbei nicht möglich. *Far Jumps* sind immer Absolutsprünge.

Takte

#	8086	80286	80386	80486	Pentium
1	15	7	7+m	3	1
2	15	7	7+m	3	1
3	18+EA	10+m	11	5	2
4	11	7+m	7	5	2
5	15	11	12+m	17	3
6	24+EA	15	43+m	13	4
7	-	-	7+m	3	1
8	-	-	10+m	5	2
9	-	-	7+m	5	2
10	-	-	12+m	13	3
11	-	-	43+m	13	4

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	Bemerkungen
1	EB	i8						
2	E9	i16						
3	FF	/4	a16					
4	FF	/4						
5	EA	a16	a16					
6	FF	/5	a16					
7	E9	a32						mit Präfix OPSIZE
8	FF	/4	a16					mit Präfix OPSIZE
9	FF	/4						mit Präfix OPSIZE
10	EA	a32	a16					mit Präfix OPSIZE
11	FF	/5	a16					mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 4, 8, 10	0	8, 10	8
	?	15, 23, 24, 26, 35, 36, 40, 43, 47, 64	?	-	./.
#NP	?	1	?	-	./.
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Unbedingte Sprünge verändern den Stack nicht!

Der JMP-Befehl hat im Protected-Mode des 80286 und im Virtual-8086-Mode der 80386-Prozessoren sowie deren Nachfolger eine Erweiterung erfahren, die die Privilegstufen und Taskwechsel berücksichtigt. Dies soll jedoch in diesem Zusammenhang nicht beschrieben werden.

Beschreibung Seite 32

LAHF *8086*

Funktion Kopieren des Flagregisterinhaltes in das AH-Register.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	LAHF

Arbeitsweise Dieser Befehl kopiert das untere Byte im Flagregister in das AH-Register. Somit werden die Flagzustände von *sign*, *zero*, *auxiliary*, *parity* und *carry* in ein Rechenregister geladen. Das »Statusbyte« in AH ist bitweise wie folgt codiert:

7	6	5	4	3	2	1	0
S	**Z**		**A**		**P**		**C**

Takte

#	8086	80286	80386	80486	Pentium
	4	2	2	3	2

Opcodes

#	B1
	9F

Exceptions Keine

Beschreibung Seite 40

LAR *80286*

Funktion Zugriffsrechte laden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									?						

Konnte der Befehl erfolgreich durchgeführt werden, ist das Zero-Flag gesetzt, andernfalls gelöscht.

Verwendung

#	Parameter	Beispiel
1	r16, r16	LAR AX, BX
2	r16, m16	LAR CX, WVar
3	r32, r32	LAR EAX, EBX
4	r32, m32	LAR ECX, DVar

Arbeitsweise Der zweite Operand von LAR ist ein Selektor auf einem Segmentdeskriptor, der das Segment beschreibt, auf das zugegriffen werden soll. LAR prüft zunächst, ob dieser Selektor nicht null ist und auf einen gültigen Deskriptor innerhalb der *globalen Deskriptortabelle* (GDT) oder *lokalen Deskriptortabelle* (LDT) zeigt. Anschließend wird geprüft, ob der Deskriptor ein »erlaubtes« Segment beschreibt. Nicht erlaubt zum Einsatz bei LAR sind reservierte Segmente (Typ 0, 8, A und D), 16-Bit-Trap und Interrupt-Gates (Typ 6 bzw. 7) sowie deren 32-Bit-Pendants (Typ E bzw. F). Alle Daten- und Codesegmente sind erlaubt.

Sollten diese Tests die Ausführung von LAR erlauben, so wird das Zero-Flag gesetzt und die Zugriffsrechte vom zweiten auf den ersten Operanden kopiert. Andernfalls wird das Zero-Flag gelöscht.

Bei 32-Bit-Operanden umfassen die Zugriffsrechte das Typ- und DPL-Feld sowie die Flags S, P, AVL, D/B und G des zweiten Doppelwortes des Deskriptors. Zum Kopieren wird dieses Doppelwort des zweiten Operanden mit $00F1FF00 maskiert, so daß nur die entsprechenden Bits übertragen werden. Bei 16-Bit-Operanden erfolgt die Maskierung mit $FF00. Der so veränderte Wert wird in den ersten Operanden übertragen.

Takte

#	8086	80286	80386	80486	Pentium
1	-	14	15	11	8
2	-	16	16	11	8
3	-	-	15	11	8
4	-	-	16	11	8

Opcodes

#	B1	B2	B3		
1	0F	02	/r		
2	0F	02	/m	a16	
3	0F	02	/r		mit Präfix OPSIZE
4	0F	02	/m	a16	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	./.	./.
#GP	0	3, 8	0	./.	./.
#PF	?	1	?	./.	./.
#SS	0	1	0	./.	./.
#UD	-	-	-	1	1

Beschreibung Seite 235

LDS *8086*

Funktion DS-Offset-Kombination laden.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16, m16	LDS SI, WVar	
2	r32, m32	LDS EAX, DVar	ab 80386

Arbeitsweise LDS lädt den Segmentanteil einer Adresse, die über den zweiten Operanden übergeben wird, in das DS-Register. Der dazugehörige Offsetanteil wird in das durch den ersten Operanden spezifizierte Register geladen.

Der zweite Operand muß hierzu eine Adresse beinhalten, die gemäß Intel-Konvention einen führenden 16-Bit-Offset, gefolgt von einem 16-Bit-Segment codiert. Bei der 32-Bit-Version folgt auf einen 32-Bit-Offset ein 16-Bit-Selektor. In diesem Fall wird DS mit dem 16-Bit-Selektor, das durch den ersten Operanden spezifizierte Register mit dem 32-Bit-Offset geladen.

Takte

#	8086	80286	80386	80486	Pentium
1	16+EA	7	7	6	4
2	-	-	7	6	4

Opcodes

#	B1	B2	B3	B4	Bemerkungen
	C5	/r	a16		
	C5	/r	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 6, 8	0	8	8
	?	62, 63	?	-	./.
#NP	?	4	?	-	./.
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
	?	4	?	-	./.
#UD	-	2	-	2	2

Bemerkungen Beachten Sie bitte, daß LDS anders agiert als LEA! LEA lädt das übergebene Register mit der *Adresse* der übergebenen Variablen. Der zweite Operand dient also *direkt* zur Berechnung der zu ladenden Adresse! LDS dagegen benutzt den zweiten Operanden dazu, aus der durch ihn spezifizierten Speicherstelle *eine vollständige Adresse zu holen*, die dann in die Register geladen wird. Dieser Befehl arbeitet also *indirekt*.

Beschreibung Seite 19

LEA *8086*

Funktion Effektive Adresse laden.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16, Var	LEA SI, EinByte	
2	r32, Var	LEA EDI, EinWort	ab 80386

Arbeitsweise LEA speichert in dem als ersten Operanden übergebenen Register den Offsetanteil der Variablen, die als zweiter Operand übergeben wird.

ACHTUNG: LEA lädt im Beispiel #1 **nicht** den **Inhalt** der Variablen *EinByte* in das Register SI, sondern den Offsetanteil der **Adresse** von *EinByte*! LEA dient also dazu, die Adresse der Variablen im Speicher in ein Register zu kopieren. Verwendet wird dieser Befehl z.B. häufig bei den Stringbefehlen, um DS:SI bzw. ES:DI mit der Adresse der Stringvariablen zu belegen. Beachten Sie hierbei bitte, daß DS bzw. ES mit gesonderten Befehlen mit dem korrekten Segmentanteil der Adresse geladen werden müssen! LEA tut dies **nicht**!

Tip: LEA erledigt die gleiche Aufgabe, die auch mit der Assembleranweisung OFFSET bewerkstelligt wird. So ist

```
lea di, EinByte
```

identisch mit

```
mov di, OFFSET EinByte
```

Beide Zeilen erzeugen den gleichen Assemblercode.

Takte

#	8086	80286	80386	80486	Pentium
1	2+EA	3	2	1	1
2	-	-	2	1	1

Opcodes

#	B1	B2	Bemerkungen
1	8D	/r	
2	8D	/r	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#UD	-	2	-	2	2

Bemerkungen Beachten Sie bitte, daß LEA anders agiert als die Ladebefehle LDS, LES, LFS, LGS und LSS! LEA belädt das übergebene Register mit der ***Adresse*** der übergebenen Variablen. Der zweite Operand dient also ***direkt*** zur Berechnung der zu ladenden Adresse!

Die anderen Ladebefehle benutzen den zweiten Operanden dazu, um aus der durch ihn spezifizierten Speicherstelle *eine vollständige Adresse zu holen*, die dann in die Register geladen wird. Diese Befehle arbeiten also *indirekt*.

Beschreibung Seite 19

LEAVE *80186*

Funktion Stackrahmen entfernen.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keinen	LEAVE

Arbeitsweise LEAVE entfernt einen mit ENTER eingerichteten Stackrahmen wieder. Die Funktion von LEAVE entspricht der Befehlsfolge:

```
mov     sp, bp
pop     bp
```

Takte

#	8086	80286	80386	80486	Pentium
	-	5	4	5	3

Opcodes

#	B1
	C9

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	18	0	19	19
#PF	?	1	?	1	./.

Beschreibung Seite 19

LES *8086*

Funktion ES-Offset-Kombination laden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16, m16	LES SI, WVar	
2	r32, m32	LES EAX, DVar	ab 80386

Arbeitsweise LES lädt den Segmentanteil einer Adresse, die über den zweiten Operanden übergeben wird, in das ES-Register. Der dazugehörige Offsetanteil wird in das durch den ersten Operanden spezifizierte Register geladen.

Der zweite Operand muß hierzu eine Adresse beinhalten, die gemäß Intel-Konvention einen führenden 16-Bit-Offset, gefolgt von einem 16-Bit-Segment codiert. Bei der 32-Bit-Version folgt auf einen 32-Bit-Offset ein 16-Bit-Selektor! In diesem Fall wird ES mit dem 16-Bit-Selektor, das durch den ersten Operanden spezifizierte Register mit dem 32-Bit-Offset geladen.

Takte

#	8086	80286	80386	80486	Pentium
1	16+EA	7	7	6	4
2	-	-	7	6	4

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	C4	/r	a16		
2	C4	/r	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 6, 8	0	8	8
	?	62, 63	?	-	./.
#NP	?	4	?	-	./.
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
	?	4	?	-	./.
#UD	-	2	-	2	2

Bemerkungen Beachten Sie bitte, daß LES anders agiert als LEA! LEA lädt das übergebene Register mit der *Adresse* der übergebenen Variablen. Der zweite Operand dient also *direkt* zur Berechnung der zu ladenden Adresse! LES dagegen benutzt den zweiten Operanden dazu, aus der durch ihn spezifizierten Speicherstelle *eine vollständige Adresse zu holen*, die dann in die Register geladen wird. Dieser Befehl arbeitet also *indirekt*.

Beschreibung Seite 19

LFS *80386*

Funktion FS-Offset-Kombination laden.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16, m16	LFS SI, WVar	ab 80386
2	r32, m32	LFS EAX, DVar	ab 80386

Arbeitsweise

LFS lädt den Segmentanteil einer Adresse, die über den zweiten Operanden übergeben wird, in das FS-Register. Der dazugehörige Offsetanteil wird in das durch den ersten Operanden spezifizierte Register geladen.

Der zweite Operand muß hierzu eine Adresse beinhalten, die gemäß Intel-Konvention einen führenden 16-Bit-Offset, gefolgt von einem 16-Bit-Segment codiert. Bei der 32-Bit-Version folgt auf einen 32-Bit-Offset ein 16-Bit-Selektor! In diesem Fall wird FS mit dem 16-Bit-Selektor, das durch den ersten Operanden spezifizierte Register mit dem 32-Bit-Offset geladen.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	7	6	4
2	-	-	7	6	4

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	0F	B4	/r	a16	
2	0F	B4	/r	a16	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 6, 8	0	8	8
	?	62, 63	?	-	./.
#NP	?	4	?	-	./.
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
	?	4	?	-	./.
#UD	-	2	-	2	2

Bemerkungen

Beachten Sie bitte, daß LFS anders agiert als LEA! LEA lädt das übergebene Register mit der *Adresse* der übergebenen Variablen. Der zweite Operand dient also *direkt* zur Berechnung der zu ladenden Adresse! LFS dagegen benutzt den zweiten Operanden dazu, aus der durch ihn spezifizierten Speicherstelle *eine vollständige Adresse zu holen*, die dann in die Register geladen wird. Dieser Befehl arbeitet also *indirekt*.

Beschreibung

Seite 132

LGDT 80286

Funktion Diese Funktion dient zum Laden einer *globalen Deskriptortabelle*.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	Pointer	LGDT StrukturVar.

Arbeitsweise Der Operand ist ein Zeiger auf eine Struktur, die 6 Bytes Daten umfaßt. Diese 6 Bytes Daten sind eine 32-Bit-Basisadresse der zu benutzenden Tabelle sowie ein 16-Bit-Limit (Tabellengröße in Bytes). Falls die Operandengröße 32 Bit ist, werden die gesamten 32 Bit der Basisadresse (die oberen 4 Bytes der Struktur) und die 16 Bit des Limits (die unteren beiden Bytes) in das *Global-Descriptor-Table-Register* (GDTR) des Prozessors geladen. Ist die Operandengröße dagegen 16 Bit, so werden als Basisadresse die 24 Bits der Bytes 3, 4 und 5 der Struktur verwendet und die obersten 8 Bits im GDTR gelöscht. Für das Limit werden die gesamten beiden unteren Bytes der Struktur (16 Bit) verwendet.

Takte

#	8086	80286	80386	80486	Pentium
1	-	11	11	11	6
2	-	11	11	11	6

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	
1	0F	01	/2	a16				
2	0F	01	/2	a32				mit OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#GP	0	3, 8, 32	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
#UD	-	2	-	2	2

Bemerkungen Die Unterscheidung 16-Bit-/32-Bit-Operanden ist traditionell bedingt. So besitzt bereits der 80286 einen Protected-Mode mit allen wichtigen Registern und Konzepten. Er hatte jedoch lediglich 24 Adreßleitungen, so daß nur 24 Bits für die Adressen zur Verfügung standen. Über die Operandengröße kann man daher auf diese Situation Rücksicht nehmen.

LGDT ist ein Befehl, der für Betriebssysteme reserviert ist! Es ist zusammen mit LIDT der einzige Befehl, der direkte, lineare 32-Bit-Adressen (bzw. 24-Bit-Adressen beim 80286) in ein Prozessorregister lädt. Alle anderen Befehle, die Adressen verwenden, benutzen die Speicheradressierung des Protected-Mode mit ihren Tabellen und Selektoren.

Beschreibung Seite 185

LGS *80386*

Funktion GS-Offsetkombination laden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16, m16	LGS SI, WVar	ab 80386
2	r32, m32	LGS EAX, DVar	ab 80386

Arbeitsweise LGS lädt den Segmentanteil einer Adresse, die über den zweiten Operanden übergeben wird, in das GS-Register. Der dazugehörige Offsetanteil wird in das durch den ersten Operanden spezifizierte Register geladen.

Der zweite Operand muß hierzu eine Adresse beinhalten, die gemäß Intel-Konvention einen führenden 16-Bit-Offset, gefolgt von einem 16-Bit-Segment codiert. Bei der 32-Bit-Version folgt auf einen 32-Bit-Offset ein 16-Bit-Selektor! In diesem Fall wird GS mit dem 16-Bit-Selektor, das durch den ersten Operanden spezifizierte Register mit dem 32-Bit-Offset geladen.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	7	6	4
2	-	-	7	6	4

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	0F	B5	/r	a16		
2	0F	B5	/r	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 6, 8	0	8	8
	?	62, 63	?	-	./.

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NP	?	4	?	-	./.
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
	?	4	?	-	./.
#UD	-	2	-	2	2

Bemerkungen Beachten Sie bitte, daß LGS anders agiert als LEA! LEA lädt das übergebene Register mit der *Adresse* der übergebenen Variablen. Der zweite Operand dient also *direkt* zur Berechnung der zu ladenden Adresse! LGS dagegen benutzt den zweiten Operanden dazu, aus der durch ihn spezifizierten Speicherstelle *eine vollständige Adresse zu holen*, die dann in die Register geladen wird. Dieser Befehl arbeitet also *indirekt*.

Beschreibung Seite 132

LIDT *80286*

Funktion Diese Funktion dient zum Laden einer *Interrupt-Deskriptortabelle*.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	Pointer	LIDT StrukturVar.

Arbeitsweise Der Operand ist ein Zeiger auf eine Struktur, die 6 Bytes Daten umfaßt. Diese 6 Bytes Daten sind eine 32-Bit-Basisadresse der zu benutzenden Tabelle sowie ein 16-Bit-Limit (Tabellengröße in Bytes). Falls die Operandengröße 32 Bit ist, werden die gesamten 32 Bit der Basisadresse (die oberen 4 Bytes der Struktur) und die 16 Bit des Limits (die unteren beiden Bytes) in das *Interrupt-Descriptor-Table-Register* (IDTR) des Prozessors geladen. Ist die Operandengröße dagegen 16 Bit, so werden als Basisadresse die 24 Bits der Bytes 3, 4 und 5 der Struktur verwendet und die obersten 8 Bits im GDTR gelöscht. Für das Limit werden die gesamten beiden unteren Bytes der Struktur (16 Bit) verwendet.

Takte

#	8086	80286	80386	80486	Pentium
1	-	12	11	11	6
2	-	12	11	11	6

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	
1	0F	01	/3	a16				
2	0F	01	/3	a32				mit OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	3, 8, 32	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
#UD	-	2	-	2	2

Bemerkungen

Die Unterscheidung 16-Bit-/32-Bit-Operanden ist traditionell bedingt. So besitzt bereits der 80286 einen Protected-Mode mit allen wichtigen Registern und Konzepten. Er hatte jedoch lediglich 24 Adreßleitungen, so daß nur 24 Bits für die Adressen zur Verfügung standen. Über die Operandengröße kann man daher auf diese Situation Rücksicht nehmen.

LIDT ist ein Befehl, der für Betriebssysteme reserviert ist! Es ist zusammen mit LGDT der einzige Befehl, der direkte, lineare 32-Bit-Adressen (bzw. 24-Bit-Adressen beim 80286) in ein Prozessorregister lädt. Alle anderen Befehle, die Adressen verwenden, benutzen die Speicheradressierung des Protected-Mode mit ihren Tabellen und Selektoren.

Beschreibung

Seite 231

LLDT *80286*

Funktion

Diese Funktion dient zum Laden einer *lokalen Deskriptortabelle*.

Flags

X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16	LLDT BX
2	m16	LLDT WVar

Arbeitsweise

Der Operand ist ein Selektor, der auf eine *lokale Deskriptortabelle* zeigt. Nachdem mittels LLDT dieser Selektor in das *Local-Descriptor-Table-Register* (LDTR) geladen wurde, benutzt ihn der Prozessor als Zeiger in die *globale Deskriptortabelle*. Der entsprechende Eintrag dort muß ein Segmentdeskriptor sein, der eine *lokale Deskriptortabelle* beschreibt. Diesem Deskriptor entnimmt der Prozessor dann die Basisadresse und das Limit (Größe der *lokalen Deskriptortabelle*) und trägt sie in den nicht frei zugänglichen Teil des LDTR ein.

Takte

#	8086	80286	80386	80486	Pentium
1	-	17	20	11	9
2	-	19	20	11	9

Opcodes

#	B1	B2	B3	B4	B5
1	0F	00	/2		
2	0F	00	/2	a16	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	3, 8, 32	0	./.	./.
#NP	?	2	?	./.	./.
#PF	?	1	?	./.	./.
#SS	0	1	0	./.	./.
#UD	-	-	-	1	1

Bemerkungen

Die Operandengröße (16 oder 32 Bit) hat keinen Einfluß auf den Befehl.

Wenn der Operand den Wert 0 hat, wird das LDTR als ungültig markiert. Falls dann durch einen Adressierbefehl ein Bezug auf die LDT hergestellt werden sollte, wird eine *General-Protection-Exception* (#GP) ausgelöst. Dies gilt nicht für LAR, VERR, VERW oder LSL.

Die Register CS, DS, ES, FS, GS und SS sowie das LDTR-Feld im *Task-State-Segment* (TSS) des aktuellen Tasks werden nicht beeinflußt!

LLDT ist ein Betriebssystembefehl und sollte daher nicht von Applikationen benutzt werden!

Beschreibung

Seite 188

LMSW *80286*

Funktion Maschinenwort laden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16	LMSW BX
2	m16	LMSW WVar

Arbeitsweise LMSW kopiert aus dem Operanden die vier niedrigstwertigen Bits und lädt sie an die Bitpositionen 3 bis 0 des Kontrollregisters CR0 des Prozessors. Im einzelnen handelt es sich um die Flags PE (Bit 0), MP (Bit 1), EM (Bit 2) und TS (Bit 3). Die Flags ET, NE, WP, AM, NW, CD und PG werden nicht beeinflußt.

Takte

#	8086	80286	80386	80486	Pentium
1	-	3	10	13	8
2	-	6	13	13	8

Opcodes

#	B1	B2	B3	
1	0F	01	/6	
2	0F	01	/6	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#GP	0	3, 8, 20	0	8, 20	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Falls Bit 0 des Operanden gesetzt ist (entspricht dem PE-Flag in CR0), wird der Prozessor in den Protected-Mode geschaltet. ACHTUNG: das PE-Flag ist ein *Sticky*-Flag. Das heißt, daß es, einmal gesetzt, seinen Zustand nicht mehr ändert. LMSW kann daher *nicht* dazu verwendet werden, den Prozessor aus dem Protected-Mode wieder in den Real-Mode zurückzuschalten!

LMSW ist eine privilegierte Funktion, was bedeutet, daß sie nur auf höchster Privilegstufe (CPL = 0; nur in Betriebssystemmodulen) ausgeführt werden kann. Sie dient nur der Abwärtskompatibilität. Ab dem 80386 sollte anstelle dieser Funktion der MOV-Befehl mit seinen Erweiterungen auf die Kontrollregister verwendet werden!

Beschreibung Seite 946

LOCK 8086

Funktion Lock-Signal voranstellen.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	LOCK

Arbeitsweise LOCK bewirkt ein »Verschließen« des nächsten Befehls für den Zugriff auf Speicherstellen. LOCK ist nur bei folgenden Befehlen wirksam: ADC, ADD, AND, BT, BTC, BTR, BTS, DEC, INC, NEG, NOT, OR, SBB, SUB, XCHG und XOR.

Während LOCK aktiv ist, hat der Prozessor das alleinige Zugriffsrecht auf diese Speicherstelle. Damit wird sichergestellt, daß kein anderer Prozessor gleichzeitig auf die gleiche Speicherstelle zugreifen kann.

Takte

#	8086	80286	80386	80486	Pentium
	2	0	0	1	1

Opcodes

#	B1
	F0

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#UD	-	7	-	7	7

Bemerkungen LOCK ist nur in Multiprozessorsystemen sinnvoll anwendbar, da nur dort die Gefahr des gleichzeitigen Zugriffs mehrerer Prozessoren auf die gleiche Speicherstelle besteht!

Beschreibung Seite 79

LODS *8086*
LODSB *8086*
LODSW *8086*
LODSD *80386*

Funktion Laden eines Operanden aus einem String.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	keine	LODSB	
2a	keine	LODSW	
3a	keine	LODSD	ab 80386
1b	m8	LODS BString	*
2b	m16	LODS WString	*
3b	m32	LODS DString	* ab 80386

*ACHTUNG! Bei der Verwendung der allgemeinen Form müssen mehrere Dinge beachtet werden:

- Durch die bloße Angabe des Pseudooperanden werden die Registerkombinationen DS:SI (DS:EDI) noch nicht korrekt besetzt! Dies hat unbedingt gesondert vor der Verwendung des Stringbefehls zu erfolgen!
- Der Assembler achtet auf die korrekte Definition des Operanden. So muß als Operand ein String angegeben werden, dessen Segmentanteil in DS steht. Stimmt dieser Segmentanteil der Adresse nicht mit dem in DS stehenden überein, so erfolgt eine Fehlermeldung!
- Der verwendete String muß korrekt definiert worden sein, da nur über seine Definition der Assembler die benötigte Information über die zu verwendende Datumsgröße erhält! In Verwendung #1b muß daher eine Definition der Art

  ```
  BString DB 128 DUP (?)
  ```

  erfolgen. Durch die Anweisung DB kann der Assembler den Befehl LODS *BString* in den Befehl *LODSB* übersetzen! Analoges gilt für Verwendung #2b und #3b!

Arbeitsweise Mit LODS ist ein Auslesen eines Strings möglich. Als String bezeichnet man beim Assembler eine Folge von Variablen gleicher Größe. So gibt es *Byte-Strings*, *Wortstrings* und *Doppelwortstrings*. Die Größe der Strings ist keinen Beschränkungen unterworfen, wird aber durch das gewählte Speichermodell limitiert. So können im Real-Mode die Strings maximal 64 kByte entsprechend der Größe eines Segments sein.

LODS (*LOaD* from *S*tring) ist der Überbegriff für die Befehle LODSB, LODSW und LODSD und wird durch den Assembler in einen der drei Befehle übersetzt. LODS hat deshalb einen nur zur Assemblierung benötigten Pseudooperanden, der jedoch nur anzugeben hat, ob byteweise (LODSB), wortweise (LODSW) oder doppelwortweise (LODSD) geladen werden soll.

Die eigentlichen Operanden des Befehls LODS werden nicht explizit angegeben. Vielmehr lädt LODS in allen drei Variationen aus einem String, dessen Adresse in DS:SI (DS:ESI bei LODSD) verzeichnet ist, ein Byte in AL (LODSB), ein Wort in AX (LODSW) oder ein Doppelwort in EAX (LODSD).

LODS lädt pro Durchgang nur jeweils ein Byte, Wort oder Doppelwort aus einem String, der durch die Adresse in DS:SI adressiert wird. Stringweises Auslesen wird daher erst in einer Schleife möglich. Das Präfix REPc, der bei CMPS, INS, OUTS, SCAS und eingeschränkt auch bei STOS sinnvoll angewendet werden kann, ist hier nicht zu gebrauchen, da mit jedem Durchgang der eben ausgelesene Inhalt wieder überschrieben würde!

Damit Schleifen einfach programmierbar sind, verändert LODS auch den Inhalt des Registers SI (ESI). So wird nach dem Auslesen in Abhängigkeit vom Status des Direction-Flags sowie von der Größe des Datums der Inhalt dieser Register um ein Byte (LODSB), zwei Bytes (LODSW) oder vier Bytes (LODSD) erhöht (DF = 0) oder verringert (DF = 1). Auf diese Weise zeigt das Indexregister SI nach dem Laden *auf das nächste zu ladende Datum*!

Takte

#	8086	80286	80386	80486	Pentium
1	12	5	5	5	2
2	12	5	5	5	2
3	-	-	5	5	2

Opcodes

#	B1	Bemerkungen
1	AC	
2	AD	
3	AD	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Vor der Verwendung von LODS müssen die Register DS und SI (ESI) mit der Adresse des Strings beladen werden.

Beschreibung Seite 65

LOOP *8086*
LOOPcc *8086*

Funktion Dieser Befehl dient zum Programmieren von Schleifen.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	i8	LOOP NearLabel
2	i8	LOOPE NearLabel
3	i8	LOOPNE NearLabel

Arbeitsweise Der Befehl LOOP simuliert folgende Befehlssequenz:

```
dec     cx
jnz     NearLabel
```

Im Unterschied zu dieser Sequenz wird bei LOOP allerdings durch das Dekrementieren des Registers der Zustand der Flags *nicht* verändert! LOOP vergleicht also »intern« das CX-Register mit 0 und springt zum Label *NearLabel*, wenn CX noch nicht 0 ist.

Die bedingten Befehle LOOPE (*LOOP* while *E*qual) und LOOPNE (*LOOP* while *N*ot *E*qual) prüfen zusätzlich noch den Zustand des Zero-Flags ab: sollte dies durch einen Befehl innerhalb der Schleife verändert werden, ist auf diese Weise ein bedingtes Beenden der Schleife möglich. So beendet LOOPE (Synonym LOOPZ, *LOOP* while Zero-Flag set) die Ausführung der Befehle in der Schleife dann, wenn entweder CX Null ist oder das Zero-Flag (z.B. durch einen CMP-Befehl innerhalb der Schleife) gelöscht wurde. LOOPNE (Synonym LOOPNZ; *LOOP* while *N*ot Zero-Flag set) dagegen bricht die Schleife bei gesetztem Zero-Flag ab (und natürlich, falls CX = 0!).

Takte

#	8086	80286	80386	80486	Pentium
1	17	8	11+m	2	5/6
2	18	8	11+m	9	7/8
3	19	8	11+m	9	7/8

* Die jeweils ersten Werte gelten, falls kein Sprung erfolgt. Andernfalls sind die zweiten Werte zu berücksichtigen.

Opcodes

#	B1	B2
1	E2	i8
2	E1	i8
3	E0	i8

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	10	0	-	-

Bemerkungen Im Unterschied zu REP/REPc lassen sich mit dem LOOP/LOOPcc-Befehl mehrere Assemblerbefehle in einer Schleife ausführen! Beachten Sie bitte, daß die durch LOOP/LOOPc durchgeführten Sprünge Relativsprünge mit einer maximalen Distanz von 127 Bytes sind (siehe hierzu JMP/Jcc!).

Beschreibung Seite 37

LSL *80286*

Funktion Berechnung einer Segmentgrenze in Bytes.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									?						

Falls die Funktion erfolgreich beendet werden konnte, ist das Zero-Flag gesetzt, andernfalls gelöscht.

Verwendung

#	Parameter	Beispiel
1	r16, r16	LSL AX, BX
2	r16, m16	LSL CX, WVar
3	r32, r32	LSL EAX, EBX
4	r32, m32	LSL ECX, DVar

Arbeitsweise LSL sammelt aus den zerstückelten Einträgen für das Segmentlimit in einem Segmentdeskriptor die Bits und erstellt daraus einen korrekten 32-Bit-Wert. Im zweiten Operanden muß dazu ein Selektor stehen, der auf einen Segmentdeskriptor in der *globalen Deskriptortabelle* oder der *lokalen Deskriptortabelle* verweist. (Welche Tabelle verwendet wird, entscheidet Bit 2, der *Tabellenindikator* TI, des Selektors. Ist dieser gesetzt, wird die lokale Deskriptortabelle verwendet, andernfalls die globale Deskriptortabelle.)

Das Segmentlimit ist in den Bits 3 bis 0 von Byte 6 sowie in den Bytes 1 und 0 des Deskriptors codiert. LSL entnimmt diese Bits und bringt sie in die korrekte Reihenfolge, um einen 20-Bit-Wert zu ergeben. Sie stellen die unteren 20 Bit eines 32-Bit-Wertes dar, die oberen 12 Bits werden auf 0 gesetzt. Falls das *Granularity*- Flag G im Deskriptor gelöscht ist, repräsentiert der so erhaltene 32-Bit-Wert die Größe des Segments in Byte. In diesem Fall kann das Segment also maximal 2^{20} Byte = 1 MByte groß werden. Ist das *Granularity-Bit* dagegen gesetzt, so werden die 20 Bits 12 Stellen nach links geschoben, wobei die frei werdenden unteren Bits mit 1 besetzt werden. In diesem Fall repräsentiert die 32-Bit-Zahl ebenfalls die Größe des Segments in Byte, sie kann nun zwischen 4 kByte und 4 GByte liegen.

Nach diesen Operationen wird die berechnete Segmentgröße in Bytes in den ersten Operanden kopiert. Ist dieser Operand ein 32-Bit-Register, so werden die gesamten 32 Bit des berechneten Segmentlimits übertragen. Bei einem 16-Bit-Register werden lediglich die unteren 16 Bits des 32-Bit-Wertes kopiert.

Takte

#	8086	80286	80386	80486	Pentium
1	-	14	20*	10	8
2	-	16	21*	10	8
3	-	-	20*	10	8
4	-	-	21*	10	8

Diese Werte gelten, wenn das Segment byte-granular ist. Bei Page-Granularität des Segments (*Granularity-Bit G* = 1) gilt 25 bzw. 26 Takte.

Opcodes

#	B1	B2	B3		
1	0F	03	/r		
2	0F	03	/m	a16	
3	0F	03	/r		mit Präfix OPSIZE
4	0F	03	/m	a16	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	./.	./.
#GP	0	3, 8	0	./.	./.
#PF	?	1	?	./.	./.
#SS	0	1	0	./.	./.
#UD	-	-	-	1	1

Bemerkungen

LSL kann dazu benutzt werden, einfach einen Offset in ein Segment mit dem Segmentlimit dieses Segments zu vergleichen, da LSL die Segmentgröße in Bytes unabhängig von der Granularität des Segments berechnet.

LSL führt eine Reihe von Prüfungen aus. So wird geprüft, ob der Selektor 0 ist oder nicht auf gültige Einträge in der GDT oder LDT zeigt. Ferner wird geprüft, ob der Segmentdeskriptor auf ungültige Segmente zeigt und ob die Privilegstufen CPL und RPL größer als der DPL sind. In allen diesem Fällen wird das Zero-Flag gelöscht und der erste Operand bleibt unverändert. Ungültige Segmente sind Segmente der Typen 0, 8, A und D (reservierte Typen), 4, 5, 6, 7 (16-Bit-*Call-*, *-Task,-* *-Trap-* und *-Interrupt*-Gates) sowie C, E und F (deren 32-Bit-Pendants). Gültige Segmente sind alle Code- und Datensegmente sowie Segmente der Typen 1 (16-Bit-*Task-State-Segment*), 2 (*lokale Deskriptortabelle*), 3 (aktives 16-Bit-*Task-State-Segment*), 9 (32-Bit-*Task-State-Segment*) und B (aktives 32-Bit-*Task-State-Segment*).

Beschreibung

Seite 181

LSS 80386

Funktion SS-Offsetkombination laden.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16, m16	LSS SI, WVar	ab 80386
2	r32, m32	LSS EAX, DVar	ab 80386

Arbeitsweise LSS lädt den Segmentanteil einer Adresse, die über den zweiten Operanden übergeben wird, in das SS-Register. Der dazugehörige Offsetanteil wird in das durch den ersten Operanden spezifizierte Register geladen.

Der zweite Operand muß hierzu eine Adresse beinhalten, die gemäß Intel-Konvention einen führenden 16-Bit-Offset, gefolgt von einem 16-Bit-Segment codiert. Bei der 32-Bit-Version folgt auf einen 32-Bit-Offset ein 16-Bit-Selektor. In diesem Fall wird SS mit dem 16-Bit-Selektor, das durch den ersten Operanden spezifizierte Register mit dem 32-Bit-Offset geladen.

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	7	6	4
2	-	-	7	6	4

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	0F	B2	/r	a16		
2	0F	B2	/r	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 6, 8	0	8	8
	?	62, 63	?	-	./.
#NP	?	4	?	-	./.
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
	?	4	?	-	./.
#UD	-	2	-	2	2

Bemerkungen Beachten Sie bitte, daß LSS anders agiert als LEA! LEA lädt das übergebene Register mit der *Adresse* der übergebenen Variablen. Der zweite Operand dient also *direkt* zur Berechnung der zu ladenden Adresse! LSS dagegen benutzt den zweiten Operanden dazu, aus der durch ihn spezifizierten Speicherstelle *eine vollständige Adresse zu holen*, die dann in die Register geladen wird. Dieser Befehl arbeitet also *indirekt*.

Beschreibung Seite 132

LTR *80286*

Funktion Dieser Befehl dient zum Laden des *Task-Registers* des Prozessors.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16	LTR BX
2	m16	LTR WVar

Arbeitsweise LTR kopiert den im Operanden stehenden Selektor in den veränderbaren Teil des Task-Registers. Anschließend benutzt LTR diesen Selektor als Zeiger in die globale Deskriptortabelle, um den korrespondierenden *Task-State-Segmentdeskriptor* auszulesen. Diesem entnimmt LTR die Basisadresse und das Limit des Task-State-Segments und kopiert es in den von außen nicht veränderbaren Teil des Task-State- Registers. Damit ist ein neuer Task »*busy*« und wird als solches markiert.

Takte

#	8086	80286	80386	80486	Pentium
1	-	17	23	20	10
2	-	19	27	20	10

Opcodes

#	B1	B2		
1	0F	00	/3	
2	0F	00	/3	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#GP	0	3, 8, 32	0	./.	./.
	?	15, 65	?	./.	./.
#NP	?	3	?	./.	./.
#PF	?	1	?	./.	./.
#SS	0	1	0	./.	./.
#UD	-	-	-	1	1

Bemerkungen

ACHTUNG: Durch das Kopieren der Segmentangaben des neuen Tasks und durch die »*Busy*«-Markierung ist dieser Task noch nicht aktiv! LTR aktiviert keine Tasks, der *Task-Switch* muß noch erfolgen!

LTR ist eine privilegierte Funktion, was bedeutet, daß sie nur auf höchster Privilegstufe (CPL = 0; nur in Betriebssystemmodulen) ausgeführt werden kann. Sie dient zur Implementation von Betriebssystemvorgängen und sollte daher nicht von einem Anwendungsprogramm ausgeführt werden. Üblicherweise wird sie für die Initialisierung des ersten auszuführenden Tasks nach einem Bootvorgang verwendet.

Die Operandengröße hat keinen Einfluß auf die Funktion. Es werden in jedem Fall 16 Bits an Information ausgelesen.

Beschreibung

Seite 220

MOV *8086*

Funktion

Kopieren eines Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	MOV AH, BL	
2	r8, m8	MOV CL, BVar	
3	m8, r8	MOV BVar, CH	
4	r8, i8	MOV AL, 012h	
5	m8, i8	MOV BVar, 034h	
6	r16, r16	MOV AX, BX	
7	r16, m16	MOV CX, WVar	
8	m16, r16	MOV WVar, DX	
9	r16, i16	MOV AX, 04711h	
10	m16, i16	MOV WVar, 00815h	
11	r32, r32	MOV EAX, EBX	ab 80386
12	r32, m32	MOV ECX, DVar	ab 80386
13	m32, r32	MOV DVar, EDX	ab 80386
14	r32, i32	MOV EAX, 012345678h	ab 80386
15	m32, i32	MOV DVar, 098765432h	ab 80386
16	AL,m8	MOV AL, BVar	nur AL!
17	AX, m16	MOV AX, WVar	nur AX!
18	EAX, m32	MOV EAX, DVar	nur EAX!
19	m8, AL	MOV BVar, AL	nur AL!

20	m16, AX	MOV WVar, AX	nur AX!
21	m32, EAX	MOV DVar, EAX	nur EAX!
22	r16, sreg	MOV AX, CS	nur Segment!
23	m16, sreg	MOV WVar, CS	nur Segment!
24	sreg, r16	MOV DS, BX	nur Segment!
25	sreg, m16	MOV ES, WVar	nur Segment!
26	r8, i8	MOV AL, 012h	
27	r16, i16	MOV DS, 01234h	
28	r32, i32	MOV EDI, 012345678h	ab 80386

Arbeitsweise MOV (*MOV*e) kopiert den Inhalt des zweiten Operanden (»Quelle«) in den ersten Operanden (»Ziel«).

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	2	1	1
2	8+EA	5	4	1	1
3	9+EA	3	2	1	1
4	4	2	2	1	1
5	10+EA	3	2	1	1
6	2	2	2	1	1
7	8+EA	5	4	1	1
8	9+EA	3	2	1	1
9	4	2	2	1	1
10	10+EA	3	2	1	1
11	-	-	2	1	1
12	-	-	2	1	1
13	-	-	2	1	1
14	-	-	2	1	1
15	-	-	2	1	1
16	10	5	4	1	1
17	10	5	4	1	1
18	-	-	4	1	1
19	10	3	2	1	1
20	10	3	2	1	1
21	-	-	2	1	1
22	2	2	2	3	1
23	9+EA	3	2	3	1
24	2	2	2	3	2
25	8+EA	5	5	9	3
26	4	2	2	1	1
27	4	2	2	1	1
28	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	88	/r							
2	8A	/m	a16						
3	88	/m	a16						
4	C6	i8							
5	C6	a16		i8					
6	89	/r							
7	8B	/m	a16						
8	89	/m	a16						
9	C7	i16							
10	C7	a16		i16					
11	89	/r							mit Präfix OPSIZE
12	8B	/m	a16						mit Präfix OPSIZE
13	89	/m	a16						mit Präfix OPSIZE
14	C7	i32							mit Präfix OPSIZE
15	C7	a16			i32				mit Präfix OPSIZE
16	A0								
17	A1								
18	A1								mit Präfix OPSIZE
19	A2								
20	A3								
21	A3								mit Präfix OPSIZE
22	8C	/r							
23	8C	/m	a16						
24	8D	/r							
25	8D	/m	a16						
26	B0+i	i8							*
27	B8+i	i16							*
28	B8+i	i32							mit Präfix OPSIZE *

* i kann Werte zwischen 0 und 7 annehmen und definiert das zu verwendende Register. Die Register werden hierbei wie folgt codiert:

	0	1	2	3	4	5	6	7
r8	AL	CL	DL	BL	AH	CH	DH	BH
r16	AX	CX	DX	BX	SP	BP	SI	DI
r32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#DB	-	1	-	1	1
#GP	0	3, 6, 8, 20, 32, 49, 50, 51	0	2, 8, 56	8, 50
	?	14, 21, 22, 41, 46	?	-	./.
#NP	?	4	?	4	./.
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
	?	7	?	-	./.
#UD	-	8, 9	-	8, 9	8, 9

Bemerkungen

Ab dem 80386 wurde der MOV-Befehl noch wesentlich erweitert! So kann ab diesem Prozessortyp auch ein Datentransfer zwischen den Spezialregistern und den »normalen« Registern des Prozessors durchgeführt werden. Da diese Spezialregister in diesem Buch nicht besprochen werden, wurde auf die Auflistung dieser Erweiterungen verzichtet!

Zur Orientierung: Die Erweiterungen beziehen sich auf das Ansprechen der Kontrollregister CR0 bis CR3, der Testregister und der Debug-Register DR0 bis DR7.

Beim Pentium kam ein neues Kontrollregister (CR4) hinzu, weshalb auch der MOV-Befehl ein weiteres Mal erweitert wurde, um den Datenaustausch mit diesem Register zu ermöglichen. Da aber auch die Testregister erheblich erweitert wurden, wurden sie zwecks Zugriff aus dem MOV-Befehl herausgenommen und zwei neue Befehle kreiert: RDMSR und WRMSR. Achtung: Dies betrifft nur die Testregister!

Beschreibung

Seite 20

MOVS *8086*
MOVSB *8086*
MOVSW *8086*
MOVSD *80386*

Funktion Kopieren eines Operanden in einen String.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	keine	MOVSB	
2a	keine	MOVSW	
3a	keine	MOVSD	ab 80386
1b	m8, m8	MOVS BString1, BString2	*
2b	m16, m16	MOVS WString1, WString2	*
3b	m32, m32	MOVS DString1, DString2	* ab 80386

*ACHTUNG! Bei der Verwendung der allgemeinen Form müssen mehrere Dinge beachtet werden:

- Durch die bloße Angabe der beiden Pseudooperanden werden die Registerkombinationen DS:SI (DS:EDI) und ES:DI (ES:EDI) noch nicht korrekt besetzt! Dies muß unbedingt gesondert vor der Verwendung des Stringbefehls erfolgen!
- Der Assembler achtet auf die korrekte Reihenfolge der Operanden. So muß als erster Operand der String angegeben werden, dessen Segmentanteil in ES steht. Als zweiter Operand wird dann der String erwartet, dessen Segmentanteil in DS steht! Stimmen diese Segmentanteile der Adressen nicht mit den in DS/ES stehenden überein, so erfolgt eine Fehlermeldung!
- Die verwendeten Strings müssen korrekt definiert worden sein, da nur über ihre Definition der Assembler die benötigte Information über die zu verwendende Datumsgröße erhält! In Verwendung #1b muß daher eine Definition der Art

  ```
  BString1 DB 128 DUP (?)
  ```

  erfolgen. Durch die Anweisung DB kann der Assembler den Befehl *MOVS BString1, BString2* in den Befehl *MOVSB* übersetzen! Analoges gilt für Verwendung #2b und #3b!

Arbeitsweise Mit MOVS ist ein einfaches Kopieren eines Strings möglich. Als String bezeichnet man beim Assembler eine Folge von Variablen gleicher Größe. So gibt es *Byte-Strings*, *Wortstrings* und *Doppelwortstrings*. Die Größe der Strings ist keinen Beschränkungen unterworfen, wird aber durch das ge-

wählte Speichermodell limitiert. So können im Real-Mode die Strings maximal 64 kByte entsprechend der Größe eines Segments sein.

MOVS (*MOV*e *S*trings) ist der Überbegriff für die Befehle MOVSB, MOVSW und MOVSD und wird durch den Assembler in einen der drei Befehle übersetzt. MOVS hat deshalb zwei nur zur Assemblierung benötigte Pseudooperanden, die jedoch nur anzugeben haben, ob byteweise (MOVSB), wortweise (MOVSW) oder doppelwortweise (MOVSD) kopiert werden soll.

Die eigentlichen Operanden des Befehls MOVS werden nicht explizit angegeben. Vielmehr kopiert MOVS in allen drei Variationen einen String, dessen Adresse in DS:SI (DS:ESI bei MOVSD) verzeichnet ist, in einen String, dessen Adresse in ES:DI (ES:EDI) steht.

MOVS kopiert pro Durchgang nur jeweils ein Byte, Wort oder Doppelwort. Stringweises Kopieren wird daher erst in Verbindung mit einem der Präfixe REPc möglich. Diese Präfixe bewirken, daß MOVS so lange ausgeführt wird, bis ein Abbruchkriterium erfüllt ist (siehe dort).

Damit diese Präfixe korrekt arbeiten können, verändert MOVS auch den Inhalt der Register SI und DI. So wird nach dem Kopieren in Abhängigkeit vom Status des Direction-Flags sowie der Größe des Datums der Inhalt dieser Register um ein Byte (MOVSB), zwei Bytes (MOVSW) oder vier Bytes (MOVSD) erhöht (DF = 0) oder verringert (DF = 1). Auf diese Weise zeigen die Indexregister SI und DI nach dem Vergleich *auf das nächste zu kopierende Datum und die nächste zu belegende Speicherstelle*!

Takte

#	8086	80286	80386	80486	Pentium
1	18	5	7	7	4
2	18	5	7	7	4
3	-	-	7	7	4

Opcodes

#	B1	Bemerkungen
1	A4	
2	A5	
3	A5	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Vor der Verwendung von MOVS müssen die Register DS, SI (ESI), ES und DI (EDI) mit den Adressen der Strings geladen werden.

Beschreibung Seite 68

MOVSX 80386

Funktion Kopieren eines Operanden mit Vorzeichenerweiterung.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16, r8	MOVSX AX, BL
2	r16, m8	MOVSX BX, BVar
3	r32, r8	MOVSX ECX, CL
4	r32, m8	MOVSX EDX, BVar
5	r32, r16	MOVSX ESI, DI
6	r32, m16	MOVSX EDI, WVar

Arbeitsweise MOVSX kopiert wie der »normale« MOV-Befehl den Inhalt des zweiten Operanden (»Quelle«) in den ersten (»Ziel«). Hierbei findet jedoch eine Erweiterung der Größe der Quelle auf die des Ziels statt. MOVSX interpretiert den Wert in der Quelle als vorzeichenbehaftete Zahl.

Im Prinzip macht z.B. MOVSX AX, BVar nichts anderes als:

```
mov     al, BVar
cbw
```

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	3	3	3
2	-	-	6	3	3
3	-	-	3	3	3
4	-	-	6	3	3
5	-	-	3	3	3
6	-	-	6	3	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
	0F	BE	/r			
	0F	BE	/m	a16		
	0F	BE	/r			mit Präfix OPSIZE
	0F	BE	/m	a16		mit Präfix OPSIZE
	0F	BF	/r			mit Präfix OPSIZE
	0F	BF	/m	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen MOVZX führt die gleiche Operation aus, nur daß hierbei ein Vorzeichen *nicht* berücksichtigt wird! Die Erweiterung erfolgt also durch das Auffüllen mit Nullen!

Beschreibung Seite 133

MOVZX *80386*

Funktion Kopieren eines Operanden mit »Nullerweiterung«.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16, r8	MOVZX AX, BL
2	r16, m8	MOVZX BX, BVar
3	r32, r8	MOVZX ECX, CL
4	r32, m8	MOVZX EDX, BVar
5	r32, r16	MOVZX ESI, DI
6	r32, m16	MOVZX EDI, WVar

Arbeitsweise MOVZX kopiert wie der »normale« MOV-Befehl den Inhalt des zweiten Operanden (»Quelle«) in den ersten (»Ziel«). Hierbei findet jedoch eine Erweiterung der Größe der Quelle auf die des Ziels statt. MOVZX interpretiert den Wert in der Quelle als vorzeichenlose Zahl.

Im Prinzip macht z.B. MOVZX AX, BVar nichts anderes als:

```
mov     al, BVar
xor     ah,ah
```

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	3	3	3
2	-	-	6	3	3
3	-	-	3	3	3
4	-	-	6	3	3
5	-	-	3	3	3
6	-	-	6	3	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
	0F	BE	/r			
	0F	BE	/m	a16		
	0F	BE	/r			mit Präfix OPSIZE
	0F	BE	/m	a16		mit Präfix OPSIZE
	0F	BF	/r			mit Präfix OPSIZE
	0F	BF	/m	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen MOVSX führt die gleiche Operation aus, nur daß hierbei ein Vorzeichen berücksichtigt wird! Die Erweiterung erfolgt also durch das Auffüllen mit dem Vorzeichen aus der Quelle!

Beschreibung Seite 133

MUL *8086*

Funktion Multiplikation zweier vorzeichenloser Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				?	?		?		?		*

Die Flags *Overflow* und *Carry* werden in Abhängigkeit vom Ergebnis der Multiplikation gesetzt, alle anderen sind undefiniert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	MUL BL	AL implizit!
2	m8	MUL BVar	AL implizit!
3	r16	MUL BX	AX implizit
4	m16	MUL WVar	AX implizit
5	r32	MUL EBX	ab 80386
6	m32	MUL DVar	ab 80386

Arbeitsweise MUL führt eine Multiplikation aus, bei der die Werte der Operanden vorzeichenlos interpretiert werden. Dies bedeutet, daß das höchstwertige Bit (Bit 31, 15 bzw. 7) *nicht* als Vorzeichen gewertet wird!

Der erste Operand (»Ziel«; Multiplikand) wird nur implizit angegeben und bezeichnet *immer* den Akkumulator! Implizit bedeutet hierbei, daß der Akkumulator nicht als Operand angegeben wird, jedoch tatsächlich vorhanden ist. Der zweite Operand (also der einzige tatsächlich angegebene, die »Quelle«) enthält den Multiplikator.

Die Größe des explizit übergebenen Operanden gibt automatisch die verwendeten Register/Registerkombinationen vor:

Operand	Multiplikand	Produkt
Byte	AL	AX
Word	AX	DX:AX
Dword	EAX	EDX:EAX

So legt beispielsweise der Befehl *MUL EinByte* aufgrund der Operandengröße von 1 Byte des (expliziten) Operanden *EinByte* folgende Rahmenbedingungen für die Multiplikation fest:

- Der Multiplikand muß als Byte in AL angegeben werden.
- Der Multiplikator hat Bytegröße (weil er mit dem explizit angegebenen Operanden identisch ist). Damit hat das Produkt Wortgröße!
- Das Ergebnis der Multiplikation, das Produkt, findet sich im Wortregister AX wieder.

Die Operation MUL ist *nicht* die Umkehrung der Operation DIV! Während letztere als Ergebnis einen Quotienten und einen Divisionsrest erzeugt, berücksichtigt MUL einen eventuell zu addierenden Wert in AH, DX oder EDX *nicht*! Im Gegenteil: dieser wird durch MUL überschrieben.

Takte

#	8086	80286	80386	80486	Pentium
1	80-90		14	16	11
2	(80-96)+EA		17	16	11
3	144-162		22	24	11
4	(150-186)+EA		25	24	11
5	-	-	38	40	10
6	-	-	41	40	10

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	F6	/2			
2	F6	/2	a16		
3	F7	/2			
4	F7	/2	a16		
5	F7	/2			mit Präfix OPSIZE
6	F7	/2	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen

Eine Multiplikation zweier Werte durch MUL kann keinen Überlauf erzeugen! Da das Produkt zweier Werte immer innerhalb des darstellbaren Bereichs des Zielregisters liegt (liegen muß: Byte · Byte = Wort; Wort · Wort = DWort etc.), kann es keine Ausnahmesituationen geben, weshalb keine Interrupt-Auslösung wie bei DIV vorgesehen ist. Somit hat auch das Carry-Flag lediglich die Funktion anzuzeigen, ob das Produkt noch innerhalb der Größe von Multiplikand und Multiplikator liegt. Wird z.B. 25 mit 8 multipliziert, so ist das Ergebnis noch durch ein Byte darstellbar! Das höherwertige Byte des Produkts in AX (also der Inhalt von AH) ist damit 0. Dies wird auch durch ein gelöschtes Carry- und Overflow-Flag signalisiert.

Die Multiplikation von 25 mit 16 dagegen läßt sich nicht mehr mit einem Byte darstellen! AH ist somit als höherwertiges Byte des Produkt-Wortes von 0 verschieden. Gleiches zeigen auch das gesetzte Carry- und Overflow-Flag. Analoges gilt für die Multiplikation zweier Worte (DX = 0; Carry- und Overflow-Flag gelöscht; DX <> 0; Carry- und Overflow-Flag gesetzt) und zweier Doppelworte.

Beschreibung

Seite 56

NEG *8086*

Funktion

(Arithmetische) Negation eines Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Negation gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	NEG AL	
2	m8	NEG BVar	
3	r16	NEG CX	
4	m16	NEG WVar	
5	r32	NEG EDX	ab 80386
6	m32	NEG DVar	ab 80386

Arbeitsweise

NEG bildet das sogenannte Zweierkomplement des Wertes im Operanden. Hierbei wird dieser Wert von 0 abgezogen und in den Operanden zurückgeschrieben.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	16+EA	7	6	3	3
3	3	2	2	1	1
4	16+EA	7	6	3	3
5	-	-	2	1	1
6	-	-	6	3	3

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	F6	/3			
2	F6	/3	a16		
3	F7	/3			
4	F7	/3	a16		
5	F7	/3			mit Präfix OPSIZE
6	F7	/3	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung

Seite 63

NOP *8086*

Funktion

Dieser Befehl (*No OPeration*) bewirkt, daß eine leere Prozessoranweisung ausgeführt wird. Es werden lediglich einige Prozessortakte verbraucht.

Flags

X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keiner	NOP

Arbeitsweise NOP führt eine »Null«-Operation aus. Das bedeutet, daß keine Operation durchgeführt wird. Nach den durch diesen Befehl abgelaufenen Takten wird der Programmablauf mit dem nächsten Befehl fortgesetzt.

Takte

#	8086	80286	80386	80486	Pentium
	3	3	3	1	1

Opcodes

#	B1
	90

Exceptions Keine

Beschreibung Seite 78

NOT *8086*

Funktion (Logische) Negation eines Operanden.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden im Gegensatz zu den Verhältnissen bei NEG *nicht* verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8	NOT AL	
2	m8	NOT BVar	
3	r16	NOT CX	
4	m16	NOT WVar	
5	r32	NOT EDX	ab 80386
6	m32	NOT DVar	ab 80386

Arbeitsweise NOT bildet das sogenannte Einerkomplement des Wertes im Operanden. Dies bedeutet, daß jedes Bit in sein Komplement überführt wird: 1 wird zu 0 und umgekehrt!

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	16+EA	7	6	3	3
3	3	2	2	1	1
4	16+EA	7	6	3	3
5	-	-	2	1	1
6	-	-	6	3	3

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	F6	/3			
2	F6	/3	a16		
3	F7	/3			
4	F7	/3	a16		
5	F7	/3			mit Präfix OPSIZE
6	F7	/3	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 46

OR *8086*

Funktion (Logische) ODER-Verknüpfung zweier Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				0				*	*		?		*		0

Die Flags werden in Abhängigkeit vom Ergebnis der Verknüpfung gesetzt, das Overflow- und Carry-Flag werden explizit gelöscht.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	OR AH, BL	
2	r8, m8	OR CL, BVar	
3	m8, r8	OR BVar, CH	
4	r8, i8	OR AL, $12	
5	m8, i8	OR BVar, $34	
6	r16, r16	OR AX, BX	
7	r16, m16	OR CX, WVar	
8	m16, r16	OR WVar, DX	
9	r16, i16	OR AX, $4711	
10	m16, i16	OR WVar, $0815	
11	r16, i8	OR DX, $12	ab 80386
12	m16, i8	OR WVar, $12	ab 80386
13	r32, r32	OR EAX, EBX	ab 80386
14	r32, m32	OR ECX, DVar	ab 80386

15	m32, r32	OR DVar, EDX	ab 80386
16	r32, i32	OR EAX, $1234567	ab 80386
17	m32, i32	OR DVar, $987654	ab 80386
18	r32, i8	OR ESI, $12	ab 80386
19	m32, i8	OR DVar, $34	ab 80386
20	AL,i8	OR AL, 012h	nur AL!
21	AX, i16	OR AX, 01234h	nur AX!
22	EAX, i32	OR EAX, 0123456h	nur EAX!

Arbeitsweise

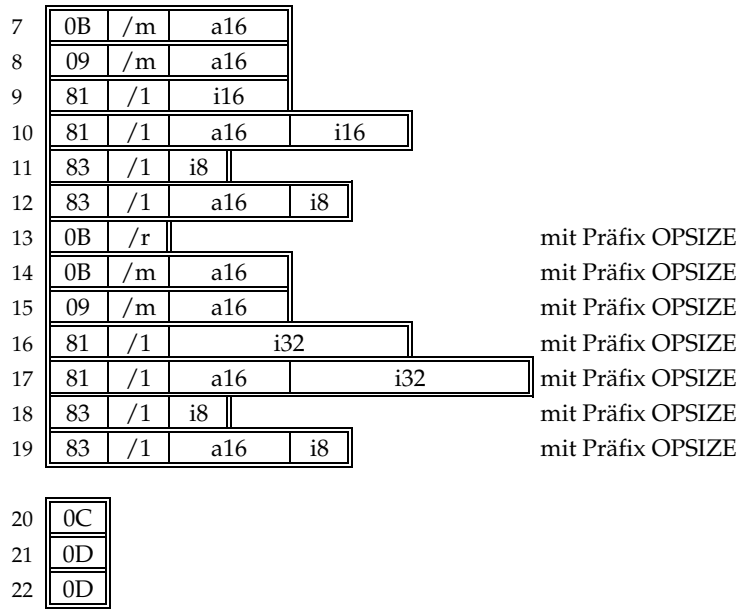
OR führt eine logische ODER-Verknüpfung durch. Hierbei wird bitweise der erste Operand mit dem zweiten Operanden ODER-verknüpft, das Resultat wird dann in den ersten Operanden eingetragen. Der zweite Operand bleibt unverändert.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	9+EA	7	6	2	2
3	16+EA	7	7	3	3
4	4	3	2	1	1
5	17+EA	7	7	3	3
6	3	2	2	1	1
7	9+EA	7	6	2	2
8	16+EA	7	7	3	3
9	4	3	2	1	1
10	17+EA	7	7	3	3
11	-	-	2	1	1
12	-	-	7	3	3
13	-	-	2	1	1
14	-	-	6	2	2
15	-	-	7	3	3
16	-	-	2	1	1
17	-	-	7	3	3
18	-	-	2	1	1
19	-	-	7	3	3
20	4	3	2	1	1
21	4	3	2	1	1
22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	0A	/r							
2	0A	/m	a16						
3	08	/m	a16						
4	80	/1	i8						
5	80	/1	a16					i8	
6	0B	/r							



Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 46

OUT *8086*

Funktion Ausgabe eines Datums auf einen Port.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	i8, AL	IN 034h, AL,	
2	i8, AX	IN 034h, AX,	
3	i8, EAX	IN 034h, EAX	ab 80386

4	DX, AL	IN DX, AL	
5	DX, AX	IN DX, AX	
6	DXX, EAX	IN DX, EAX	ab 80386

Arbeitsweise

OUT gibt einen Wert auf einen spezifizierbaren Port aus. Ob ein Byte, ein Wort oder – ab dem 80386 – ein Doppelwort ausgelesen wird, wird durch die Angabe des Akkumulators als zweitem Operanden gesteuert. So wird bei AL als Operanden ein Byte ausgegeben.

Die Angabe des Ports, auf den geschrieben werden soll, kann auf zwei Arten erfolgen. Liegt die Portadresse im Bereich $00 bis $FF, so kann sie als Konstante direkt als erster Operand angegeben werden. Andernfalls muß sie im Register DX abgelegt werden. In diesem Fall ist der erweiterte Befehl von IN zu verwenden, bei dem DX als erster Operand fungiert.

Takte

#	8086	80286	80386	80486	Pentium
1	10	3	10	16	12
2	10	3	10	16	12
3	-	-	10	16	12
4	8	3	11	16	12
5	8	3	11	16	12
6	-	-	11	16	12

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	E6	i8			
2	E7	i8			
3	E7	i8			mit Präfix OPSIZE
4	EE				
5	EF				
6	EF				mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	34	0	48	-

Beschreibung Seite 24

OUTS *80186*
OUTSB *80186*
OUTSW *80186*
OUTSD *80386*

Funktion Kopieren eines Datums aus einem String auf einen Port.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	keine	OUTSB	
2a	keine	OUTSW	
3a	keine	OUTSD	ab 80386
1b	DX, m8	OUTS DX, BString	*
2b	DX, m16	OUTS DX, WString	*
3b	DX, m32	OUTS DX, DString	* ab 80386

*ACHTUNG! Bei der Verwendung der allgemeinen Form müssen mehrere Dinge beachtet werden:

- Durch die bloße Angabe der beiden Pseudooperanden wird die Registerkombination DS:SI (DS:EDI) noch nicht korrekt besetzt! *Dies hat unbedingt gesondert vor der Verwendung des Stringbefehls zu erfolgen!*
- Der Assembler achtet auf die korrekte Angabe der Operanden. So muß als erster Operand DX angegeben werden. Andere Register läßt der Assembler nicht zu. Als zweiter Operand wird dann der String erwartet, dessen Segmentanteil in ES steht! Stimmt dieser Segmentanteil der Adresse nicht mit dem in ES stehenden überein, so erfolgt eine Fehlermeldung!
- Die verwendeten Strings müssen korrekt definiert worden sein, da nur über ihre Definition der Assembler die benötigte Information über die zu verwendende Datumsgröße erhält! In Verwendung #1b muß daher eine Definition der Art

  ```
  BString DB 128 DUP (?)
  ```

  erfolgen. Durch die Anweisung DB kann der Assembler den Befehl *OUTS DX, BString* in den Befehl *OUTSB* übersetzen. Analoges gilt für Verwendung #2b und #3b!

Arbeitsweise Mit OUTS ist eine Ausgabe von Daten auf einen Port aus einem String möglich. Als String bezeichnet man beim Assembler eine Folge von Variablen gleicher Größe. So gibt es *Byte-Strings*, *Wortstrings* und *Doppelwortstrings*. Die Größe der Strings ist keinen Beschränkungen unterworfen, wird

aber durch das gewählte Speichermodell limitiert. So können im Real-Mode die Strings maximal 64 kByte entsprechend der Größe eines Segments sein.

OUTS ist der Überbegriff für die Befehle OUTSB, OUTSW und OUTSD und wird durch den Assembler in einen der drei Befehle übersetzt. OUTS hat deshalb zwei nur zur Assemblierung benötigte Pseudooperanden, die jedoch nur angeben, ob der Port byteweise (OUTSB), wortweise (OUTSW) oder doppelwortweise (OUTSD) beschrieben werden soll.

Die eigentlichen Operanden des Befehls OUTS werden nicht explizit angegeben. Vielmehr gibt OUTS in allen drei Variationen Daten auf den Port aus, dessen Adresse in DX angegeben wird. Die Daten werden aus einem String, dessen Adresse in DS:SI (DS:ESI) steht, entnommen.

OUTS liest pro Durchgang nur jeweils ein Byte, Wort oder Doppelwort aus dem String, der durch die Adresse in DS:SI adressiert wird. Stringweises Ausgeben wird daher erst in Verbindung mit einem der Präfixe REPc möglich. Diese Präfixe bewirken, daß OUTS so lange ausgeführt wird, bis ein Abbruchkriterium erfüllt ist (siehe dort).

Damit diese Präfixe korrekt arbeiten können, verändert OUTS auch den Inhalt des Registers SI. So wird nach dem Einlesen in Abhängigkeit vom Status des Direction-Flags sowie von der Art und Größe des Datums der Inhalt dieses Registers um ein Byte (OUTSB), zwei Bytes (OUTSW) oder vier Bytes (OUTSD) erhöht (DF = 0) oder verringert (DF = 1). Auf diese Weise zeigt das Indexregister SI nach dem Einlesen *auf die nächste auszulesende Speicherstelle!*

Takte

#	8086	80286	80386	80486	Pentium
1	-	5	14	17	13
2	-	5	14	17	13
3	-	-	14	17	13

Opcodes

#	B1	Bemerkungen
1	6E	
2	6F	
3	6F	mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8, 20, 34	0	8, 48	8
#PF	?	1	?	1	./.

Bemerkungen Vor der Verwendung von OUTS müssen die Register DX, DS und SI (ESI) mit der Adresse des Strings und der Portnummer geladen werden.

Beschreibung Seite 122

POP *8086*

Funktion Abholen eines Datums vom Stack.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r16	POP DX	
2	m16	POP WVar	
3	r32	POP EAX	ab 80386
4	m32	POP DVar	ab 80386
5	sreg	POP DS	

Arbeitsweise POP kopiert das Wort oder (ab 80386) Doppelwort, das an der Spitze des Stacks verzeichnet ist und dessen Adresse in SS:SP (SS:ESP) steht, in den Operanden. Anschließend wird der Inhalt von SP um zwei (bzw. 4 bei ESP) erhöht. POP ist somit der entgegengesetzte Befehl zu PUSH.

Takte

#	8086	80286	80386	80486	Pentium
1	8	5	4	4	1
2	17+EA	5	5	6	3
3	-	-	4	4	1
4	-	-	5	6	3
5	8	5	7	3	3

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	58+i				*
2	8F	/0	a16		
3	58+i				* mit Präfix OPSIZE
4	8F	/0	a16		mit Präfix OPSIZE
5a	1F				DS
5b	07				ES
5c	0F	A1			FS
5d	0F	A9			GS
5e	17				SS

* i kann Werte zwischen 0 und 7 annehmen und definiert das zu verwendende Register. Die Register werden hierbei wie folgt codiert:

	0	1	2	3	4	5	6	7
r16	AX	CX	DX	BX	SP	BP	SI	DI
r32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 6, 8, 20	0	8	8
	?	14, 46		-	./.
#NP	?	4	?	4	./.
#PF	?	1	?	1	./.
#SS	0	1,9	0	-	-
	?	7	?	-	./.

Bemerkungen POP CS ist verboten, da dies die sofortige Änderung der Adresse des Codesegments zur Folge hätte.

Beschreibung Seite 23

POPA *80186*
POPAD *80386*
POPAW *80386*

Funktion Abholen aller Prozessorregister vom Stack.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	keine	POPA	
2	keine	POPAD	ab 80386
3	keine	POPAW	ab 80386

Arbeitsweise POPA arbeitet vollständig analog zu POP. Während jedoch POP einen Operanden als Adresse benötigt, in die das abzuholende Wort zu kopieren ist, benötigt POPA keinen Operanden, da hier als Ziel die Register des Prozessors dienen. Darüber hinaus poppt POPA nicht nur ein Datum vom Stack, sondern 8, die in alle Register des Prozessors abgelegt werden. POPA macht somit die Funktion von PUSHA rückgängig.

Der Unterschied der POPAx-Varianten besteht darin, daß POPA lediglich in die 16-Bit-Register poppen kann, während POPAD ab dem 80386 auch die 32-Bit-Register berücksichtigt. POPAW kann dazu verwendet werden, im *.386*-Modus z.B. auch das Poppen von Worten in die 16-Bit-Anteile der 32-Bit-Register zu veranlassen, ist also lediglich ein POPA-Befehl für bestimmte Situationen und wird genau in diesen übersetzt.

Die Register werden in folgender Reihenfolge belegt:
(E)DI – (E)SI – (E)BP – (E)SP – (E)BX – (E)DX – (E)CX – (E)AX.

POPA simuliert somit folgende Befehlssequenz:

```
pop di
pop si
pop bp
pop sp
pop bx
pop dx
pop cx
pop ax
```

Takte

#	8086	80286	80386	80486	Pentium
1	-	19	24	9	5
2	-	-	24	9	5
3	-	-	24	9	5

Opcodes

#	B1	Bemerkungen
1	61	
2	61	mit Präfix OPSIZE
3	61	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#PF	?	1	?	1	./.
#SS	0	10	0	10	10

Beschreibung Seite 122

POPF *8086*
POPFD *80386*
POPFW *80386*

Funktion

Abholen eines Datums vom Stack und Laden des Flagregisters mit diesem Datum.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*	*	*	*	*	*		*		*		*

Die Flags werden in Abhängigkeit vom gepoppten Wert gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	keine	POPF	
2	keine	POPFD	ab 80386
3	keine	POPFW	ab 80386

Arbeitsweise

POPF arbeitet vollständig analog zu POP. Während jedoch POP einen Operanden als Adresse benötigt, in die das abzuholende Wort zu kopieren ist, benötigt POPF keinen Operanden, da hier als Ziel das Flagregister des Prozessors dient. POPF macht somit die Funktion von PUSHF rückgängig und lädt die gesicherten Flagzustände zurück. Der Unterschied der POPFx-Varianten besteht darin, daß POPF lediglich in das 16-Bit-Flagregister poppen kann, während POPFD ab dem 80386 auch das 32-Bit-Flagregister berücksichtigt. POPFW kann dazu verwendet werden, im *.386*-Modus z.B. auch das Poppen in den 16-Bit-Anteil des 32-Bit-Flagregisters zu veranlassen, ist also lediglich ein POPF-Befehl für bestimmte Situationen und wird genau in diesen übersetzt.

Takte

#	8086	80286	80386	80486	Pentium
1	8	5	5	9	6
2	-	-	5	9	6
3	-	-	5	9	6

Opcodes

#	B1	Bemerkungen
1	9D	
2	9D	mit Präfix OPSIZE
3	9D	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	31, 57	0	31, 57	-
#PF	?	1	?	1	./.
#SS	0	9	0	9	9

Beschreibung

Seite 40

prefix 8086

Funktion

Explizites Festlegen des Segmentregisters für Befehle, bei denen auf Speicherstellen zugegriffen wird, sowie der Größe von Operanden und Adressen.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	*	CS:	
2	*	DS:	
3	*	ES:	
4	*	FS:	ab 80386
5	*	GS:	ab 80386
6	*	SS:	
7	*	*OPSIZE:*	ab 80386
8	*	*ADRSIZE:*	ab 80386

* *prefix*-Befehle sind Befehlspräfixe. Das heißt, sie haben keinen eigenen Parameter, sondern stehen vor anderen Befehlen, denen sie dadurch ein anderes Verhalten aufprägen.

Arbeitsweise

Üblicherweise besitzen alle Befehle, bei denen auf Speicherstellen zugegriffen wird, Standardvorgaben für das zur Adressierung zu verwendende Segmentregister. So impliziert z.B. der Befehl MOV das Datenregister DS, Indizierungen verwenden üblicherweise BP als Segmentvorgabe und einige Stringbefehle ES. Durch die explizite Segmentvorgabe kann der Prozessor jedoch angewiesen werden, das genannte Segmentregister zur Berechnung der Adresse zu verwenden. Dies erfolgt durch Voranstellen des Segmentpräfixes vor die Variablenangabe, z.B. wie in *MOV ES:BVar, AX*. Der Assembler stellt damit dem MOV-Befehl das Präfix voran, so daß die Adresse aus dem Inhalt des ES-Registers und dem übergebenen Offset gebildet wird.

Die Präfixe OPSIZE und ADRSIZE sind keine direkten Segmentpräfixe! Sie können vom Programmierer nicht benutzt werden. Vielmehr setzt sie der Assembler automatisch selbständig unter bestimmten Voraussetzungen. So wird das Präfix OPSIZE immer dann einem Befehl vorangestellt, wenn ab den 80386-Prozessoren nicht die 16-Bit-Register, sondern die 32-Bit-Register angesprochen werden sollen.

So übersetzt der Assembler z.B. den Befehl *MOV AX, 04711h* in die Opcode-Sequenz B8-11-47. Durch Angabe des EAX-Registers in *MOV EAX, 04711h* wird dem Assembler mitgeteilt, daß das 32-Bit-Register EAX verwendet

werden soll. Dieser stellt also dem MOV-Befehl B8 das OPSIZE-Präfix voran: 66-B8! Doch damit nicht genug: Da nun ein 32-Bit-Register betroffen ist, wird durch das Präfix OPSIZE auch gesteuert, daß als Konstante ein 32-Bit-Wert zu verwenden ist. Der Assembler hängt somit zwei weitere (Daten)-Bytes an: MOV EAX, 04711h wird in die Sequenz 66-B8-11-47-00-00 übersetzt.

Analog wird ADRSIZE verwendet. Wenn dem Assembler durch die Anweisung *.386* mitgeteilt wird, daß das sogenannte Flat-Modell Verwendung finden soll, so setzt er jedem Befehl, der auf den Speicher zugreift und somit mit Adressen umzugehen hat, das Präfix ADRSIZE voran. Dieses bewirkt, daß die (in dieser Referenz durchgehend verwendete) Angabe von Adreßoffsets nicht mit zwei Bytes (16 Bits) erfolgt, sondern mit vier Bytes (32 Bits). *Bitte denken Sie daran, wenn Sie die Opcodes der Befehle in dieser Referenz im .386-Modus vergleichen!* Ist das Präfix ADRSIZE vorhanden, sind alle *a16*-Angaben durch *a32* zu ersetzen!

Die unterschiedliche Kombination von OPSIZE und ADRSIZE erlaubt somit alle möglichen Größenkombinationen in Befehlen, wie die nachfolgende Tabelle zeigt.

Operandengröße	Adreßlänge	OPSIZE	ADRSIZE
16	16	nein	nein
16	32	nein	ja
32	16	ja	nein
32	32	ja	ja

Takte

#	8086	80286	80386	80486	Pentium
	2	2	1	1	1

Opcodes

#	B1
1	2E
2	3E
3	26
4	64
5	65
6	36
7	66
8	67

Bemerkungen

Es gibt drei Ausnahmen, bei denen eine explizite Segmentvorgabe wirkungslos bleibt: Bei INS, SCAS und STOS bleibt ES das Zielsegment!

PUSH *8086*

Funktion Kopieren eines Datums auf den Stack.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	i8 *	PUSH 012h	ab 80186
2	r16	PUSH DX	
3	m16	PUSH WVar	
4	i16	PUSH 01234h	ab 80186
5	r32	PUSH EAX	ab 80386
6	m32	PUSH DVar	ab 80386
7	i32	PUSH 012345678h	ab 80386
8	sreg	PUSH DS	

* PUSH i8 pusht zwar einen Byte-Wert auf den Stack, jedoch wird dieser Wert vorher (vorzeichenlos) auf Wortgröße erweitert, so daß de facto ein Wort auf den Stack gebracht wird.

Arbeitsweise PUSH kopiert das *Wort* oder (ab 80386) *Doppelwort*, das im Operanden verzeichnet ist, auf den Stack an die Adresse, die in SS:SP (SS:ESP) steht. Anschließend wird der Inhalt von SP um zwei (bzw. 4 bei ESP) verringert. PUSH ähnelt somit dem Stringbefehl MOVSW, nur daß die Adreßregister SS und SP verwendet werden und diese nicht vor der Ausführung explizit geladen werden müssen.

Takte

#	8086	80286	80386	80486	Pentium
1	-	3	2	1	1
2	11	3	2	1	1
3	16+EA	5	5	4	2
4	-	3	2	1	1
5	-	-	2	1	1
6	-	-	5	4	2
7	-	-	2	1	1
8	10	3	2	3	1

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	6A	i8			
2	50+i				
3	FF	/6	a16		
4	68	i16			
5	50+i				mit Präfix OPSIZE
6	FF	/6	a16		mit Präfix OPSIZE

7	68	i32	mit Präfix OPSIZE
8a	0E		CS
8b	1E		DS
8c	06		ES
8d	0F	A0	FS
8e	0F	A8	GS
8f	16		SS

* i kann Werte zwischen 0 und 7 annehmen und definiert das zu verwendende Register. Die Register werden hierbei wie folgt codiert:

	0	1	2	3	4	5	6	7
r16	AX	CX	DX	BX	SP	BP	SI	DI
r32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1, 2

Beschreibung Seite 23

PUSHA *80286*
PUSHAD *80386*
PUSHAW *80286*

Funktion Kopieren aller Prozessorregister auf den Stack.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	keine	PUSHA	
2	keine	PUSHAD	ab 80386
3	keine	PUSHAW	ab 80386

Arbeitsweise

PUSHA arbeitet vollständig analog zu PUSH. Während jedoch PUSH einen Operanden als Adresse benötigt, aus dem das zu pushende Wort zu kopieren ist, benötigt PUSHA keinen Operanden, da hier als Quelle die Register des Prozessors dienen. Darüber hinaus pusht PUSHA nicht nur ein Datum auf den Stack, sondern den Inhalt aller acht Allzweckregister.

Der Unterschied der PUSHAx-Varianten besteht darin, daß PUSHA lediglich in die 16-Bit-Register pushen kann, während PUSHAD ab dem 80386 auch die 32-Bit-Register berücksichtigt. PUSHAW kann dazu verwendet werden, im *.386*-Modus z.B. auch das Pushen von Worten aus den 16-Bit-Anteilen der 32-Bit-Register zu veranlassen, ist also lediglich ein PUSHA-Befehl für bestimmte Situationen und wird genau in diesen übersetzt.

Die Register werden in folgender Reihenfolge belegt:
(E)AX – (E)CX – (E)DX – (E)BX – (E)SP – (E)BP – (E)SI – (E)DI.

PUSHA simuliert somit folgende Befehlssequenz:

```
push ax
push cx
push dx
push bx
push sp
push bp
push si
push di
```

Takte

#	8086	80286	80386	80486	Pentium
1	-	17	18	11	5
2	-	-	18	11	5
3	-	-	18	11	5

Opcodes

#	B1	Bemerkungen
1	60	
2	60	mit Präfix OPSIZE
3	60	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	-	0	58	58
#PF	?	1	?	1	./.
#SS	0	1	0	-	-

Beschreibung Seite 122

PUSHF *8086*
PUSHFD *80386*
PUSHFW *8086*

Funktion Kopieren des Inhalts des Flagregisters auf den Stack.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	keine	PUSHF	
2	keine	PUSHFD	ab 80386
3	keine	PUSHFW	ab 80386

Arbeitsweise PUSHF arbeitet analog zu PUSH. Während jedoch PUSH einen Operanden als Adresse benötigt, aus dem das zu pushende Wort zu kopieren ist, benötigt PUSHF keinen Operanden, da hier als Quelle das Flagregister des Prozessors dient.

Der Unterschied der PUSHFx-Varianten besteht darin, daß PUSHF lediglich das 16-Bit-Flagregister pushen kann, während PUSHFD ab dem 80386 auch das 32-Bit-Flagregister berücksichtigt. PUSHFW kann dazu verwendet werden, im *.386*-Modus z.B. auch das Pushen des niederwertigen 16-Bit-Anteils des 32-Bit-Flagregisters zu veranlassen, ist also lediglich ein PUSHF-Befehl für bestimmte Situationen und wird genau in diesen übersetzt.

Takte

#	8086	80286	80386	80486	Pentium
1	10	3	4	4	4
2	-	-	4	4	4
3	-	-	4	4	4

Opcodes

#	B1	Bemerkungen
1	9C	
2	9C	mit Präfix OPSIZE
3	9C	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	-	0	31	-
#PF	?	1	?	1	./.
#SS	0	10	0	-	-

Beschreibung Seite 40

RCL *8086*

Funktion Bitrotation nach links unter Berücksichtigung des Carry-Flags.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															*

Das Carry-Flag wird in Abhängigkeit vom Ergebnis der Rotation gesetzt, alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	RCL AL,1	
2	m8, 1	RCL BVar, 1	
3	r8, CL	RCL DH, CL	
4	m8, CL	RCL BVar, CL	
5	r8, i8	RCL BL, 008h	ab 80186
6	m8, i8	RCL BVar, 003h	ab 80186
7	r16, 1	RCL AX, 1	
8	m16, 1	RCL WVar, 1	
9	r16, CL	RCL DX, CL	
10	m16, CL	RCL WVar, CL	
11	r16, i8	RCL BX, 012h	ab 80186
12	m16, i8	RCL WVar, 034h	ab 80186
13	r32, 1	RCL EAX, 1	ab 80386
14	m32, 1	RCL DVar, 1	ab 80386
15	r32, CL	RCL EBX, CL	ab 80386
16	m32, CL	RCL DVar, CL	ab 80386
17	r32, i8	RCL ECX, 034h	ab 80386
18	m32, i8	RCL DVar, 012h	ab 80386

Arbeitsweise RCL rotiert den Inhalt des ersten Operanden um die Anzahl von Stellen nach links, die im zweiten Operanden angegeben ist. Möglich ist hierbei die direkte Angabe der Anzahl, jedoch kann der Wert auch indirekt über das CL-Register angegeben werden.

RCL benutzt bei der Bitverschiebung das *Carry-Flag*. Zur detaillierten Darstellung das Ablaufs siehe Teil 1 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	9	3	1
2	15+EA	7	10	4	3
3	8+4·b	5	9	8-30	7-24
4	20+4·b+EA	8	10	9-31	9-26
5	-	5	9	8-30	8-25
6	-	8	10	9-31	10-27
7	2	2	9	3	1
8	15+EA	7	10	4	3

9	8+4·b	5	9	8-30	7-24
10	20+4·b+EA	8	10	9-31	9-26
11	-	5	9	8-30	8-25
12	-	8	10	9-31	10-27
13	-	-	9	3	1
14	-	-	10	4	3
15	-	-	9	8-30	7-24
16	-	-	10	9-31	9-26
17	-	-	9	8-30	8-25
18	-	-	10	9-31	10-27

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/2				
2	D0	/2	a16			
3	D2	/2				
4	D2	/2	a16			
5	C0	/2	i8			
6	C0	/2	a16		i8	
7	D1	/2				
8	D1	/2	a16			
9	D3	/2				
10	D3	/2	a16			
11	C1	/2	i8			
12	C1	/2	a16		i8	
13	D1	/2				mit Präfix OPSIZE
14	D1	/2	a16			mit Präfix OPSIZE
15	D3	/2				mit Präfix OPSIZE
16	D3	/2	a16			mit Präfix OPSIZE
17	C1	/2	i8			mit Präfix OPSIZE
18	C1	/2	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch RCR, ROL, ROR, SAL, SAR, SHL, SHR.

Beschreibung Seite 45

RCR *8086*

Funktion Bitrotation nach rechts unter Berücksichtigung des Carry-Flags.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															*

Das Carry-Flag wird in Abhängigkeit vom Ergebnis der Rotation gesetzt, alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	RCR AL,1	
2	m8, 1	RCR BVar, 1	
3	r8, CL	RCR DH, CL	
4	m8, CL	RCR BVar, CL	
5	r8, i8	RCR BL, 008h	ab 80186
6	m8, i8	RCR BVar, 003h	ab 80186
7	r16, 1	RCR AX, 1	
8	m16, 1	RCR WVar, 1	
9	r16, CL	RCR DX, CL	
10	m16, CL	RCR WVar, CL	
11	r16, i8	RCR BX, 012h	ab 80186
12	m16, i8	RCR WVar, 034h	ab 80186
13	r32, 1	RCR EAX, 1	ab 80386
14	m32, 1	RCR DVar, 1	ab 80386
15	r32, CL	RCR EBX, CL	ab 80386
16	m32, CL	RCR DVar, CL	ab 80386
17	r32, i8	RCR ECX, 034h	ab 80386
18	m32, i8	RCR DVar, 012h	ab 80386

Arbeitsweise RCR rotiert den Inhalt des ersten Operanden um die Anzahl von Stellen nach rechts, die im zweiten Operanden angegeben ist. Möglich ist hierbei die direkte Angabe der Anzahl, jedoch kann der Wert auch indirekt über das CL-Register angegeben werden.

RCR benutzt bei der Bitverschiebung das Carry-Flag. Zur detaillierten Darstellung das Ablaufs siehe Teil 1 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	9	3	1
2	15+EA	7	10	4	3
3	8+4·b	5	9	8-30	7-24
4	20+4·b+EA	8	10	9-31	9-26
5	-	5	9	8-30	8-25
6	-	8	10	9-31	10-27
7	2	2	9	3	1
8	15+EA	7	10	4	3

9	8+4·b	5	9	8-30	7-24
10	20+4·b+EA	8	10	9-31	9-26
11	-	5	9	8-30	8-25
12	-	8	10	9-31	10-27
13	-	-	9	3	1
14	-	-	10	4	3
15	-	-	9	8-30	7-24
16	-	-	10	9-31	9-26
17	-	-	9	8-30	8-25
18	-	-	10	9-31	10-27

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/3				
2	D0	/3	a16			
3	D2	/3				
4	D2	/3	a16			
5	C0	/3	i8			
6	C0	/3	a16		i8	
7	D1	/3				
8	D1	/3	a16			
9	D3	/3				
10	D3	/3	a16			
11	C1	/3	i8			
12	C1	/3	a16		i8	
13	D1	/3				mit Präfix OPSIZE
14	D1	/3	a16			mit Präfix OPSIZE
15	D3	/3				mit Präfix OPSIZE
16	D3	/3	a16			mit Präfix OPSIZE
17	C1	/3	i8			mit Präfix OPSIZE
18	C1	/3	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch RCL, ROL, ROR, SAL, SAR, SHL, SHR.

Beschreibung Seite 46

RDMSR *Pentium*

Funktion Liest ein modellspezifisches Register (MSR) des Prozessors aus.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	RDMSR

Arbeitsweise RDMSR liest das *modellspezifische Register* des Prozessors aus, dessen Nummer in ECX übergeben wird. Dieses Register umfaßt 64 Bits, so daß die Information in die Registerkombination EDX:EAX eingetragen wird, wobei die oberen 32 Bit in das EDX-, die unteren in das EAX-Register kopiert werden.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	20-24

Opcodes

#	B1	B2
	0F	32

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	32, 59	0	2	59

Bemerkungen Falls das spezifizierte Register weniger als 64 Bits an Informationen enthält, so sind die nicht implementierten Bits in EDX:EAX undefiniert.

RDMSR muß mit höchster Privilegstufe (CPL = 0) ausgeführt werden, andernfalls wird eine General-Protection-Exception ausgeführt. Dies erfolgt auch, wenn eine reservierte oder undefinierte Registeradresse in ECX übergeben wird.

Beschreibung Seite 148

RDPMC — Pentium Pro

Funktion: Diese Funktion liest den *Performance-Monitoring-Counter* aus.

Flags:

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung:

#	Parameter	Beispiel
	keine	RDPMC

Arbeitsweise: Der Befehl liest das sogenannte *Performance-Monitoring-Register* aus, dessen Nummer in ECX übergeben wird. Die unteren 32 Bit des 40-Bit-Counters werden in EAX, die oberen 8 Bits in EDX abgelegt.

Takte:

#	8086	80286	80386	80486	Pentium
	-	-	-	-	-

Opcodes:

#	B1	B2
	0F	33

Exceptions:

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	52, 60	0	53, 60	53, 60

Bemerkungen: Der Pentium-Pro-Prozessor hat zwei *Performance-Monitoring-Register* mit den Nummern 0 und 1.

Falls das PCE-Flag im Kontrollregister CR4 des Prozessors gesetzt ist, können die *Performance-Monitoring-Register* in den Privilegstufen 1 bis 3 ausgelesen werden. Andernfalls ist das nur in der höchsten Privilegstufe 0 im Rahmen des Betriebssystems möglich.

RDPMC ist keine serialisierende Funktion. Das heißt, daß sie nicht darauf wartet, daß alle vorangegangenen Befehle tatsächlich abgeschlossen sind, bevor das *Performance-Monitoring-Register* ausgelesen wird. Analog können bereits folgende Befehle ausgeführt werden, bevor RDPMC das Register ausgelesen hat. Es ist somit sinnvoll, RDPMC von serialisierenden Funktionen »einzurahmen«, wie z.B. CPUID eine ist. Dies stellt sicher, daß RDPMC erst nach Abschluß aller vorangegangenen Befehle aktiv wird und die weitere Programmausführung erst nach Abschluß von RDPMC erfolgt.

RDTSC *Pentium Pro*

Funktion Diese Funktion liest den *Time-Stamp-Counter* aus.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	RDTSC

Arbeitsweise RDTSC liest ein modellspezifisches Register aus, das den sogenannten *Time-Stamp* beinhaltet. Dieser 64-Bit-Wert wird in den Registern EDX:EAX abgelegt, wobei die oberen 32 Bit in EDX, die unteren in EAX kopiert werden.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2
	0F	31

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	54	0	55	55

Bemerkungen Der Prozessor inkrementiert den *Time-Stamp-Counter* bei jedem *Clock-Zyklus* und setzt ihn auf 0 zurück, wenn ein *Processor-Reset* ausgeführt wird.

Falls das TSD-Flag im Kontrollregister CR4 des Prozessors gesetzt ist, kann das *Time-Stamp-Register* in den Privilegstufen 1 bis 3 ausgelesen werden. Andernfalls ist das nur in der höchsten Privilegstufe 0 im Rahmen des Betriebssystems möglich.

Der *Time-Stamp-Counter* kann auch mittels RDMSR ausgelesen werden!

RDTSC ist keine serialisierende Funktion. Das heißt, daß sie nicht darauf wartet, daß alle vorangegangenen Befehle tatsächlich abgeschlossen sind, bevor der Counter ausgelesen wird. Analog können bereits folgende Befehle ausgeführt werden, bevor RDTSC den Counter ausgelesen hat. Es ist somit sinnvoll, RDTSC von serialisierenden Funktionen »einzurahmen«, wie z.B. CPUID eine ist. Dies stellt sicher, daß RDTSC erst nach Abschluß aller vorangegangenen Befehle aktiv wird und die weitere Programmausführung erst nach Abschluß von RDTSC erfolgt.

Beschreibung Seite 148.

REP *8086*
REPcc *8086*

Funktion Wiederholungsanweisung für einen nachfolgenden Stringbefehl.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									(*)						

Das Zero-Flag wird verändert, jedoch nicht durch den REP-Befehl selbst, sondern durch einen eventuell nachfolgenden, zu repetierenden Befehl CMPS oder SACS! Alle anderen Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	CMPS	REPE CMPSB	≡ REP CMPS
2	CMPS	REPNE CMPSW	
3	INS	REP INSW	≡ REPE/REPNE
4	LODS	REP LODSW	nicht sinnvoll!
5	MOVS	REP MOVSD	≡ REPE/REPNE
6	OUTS	REP	≡ REPE/REPNE
7	SCAS	REPE SCASB	≡ REP SCAS
8	SCAS	REPNE SCASW	
9	STOS	REP STOSD	≡ REPE/REPNE

Arbeitsweise Das Wiederholungspräfix gibt es in drei Ausführungen:

- REP (*REP*eat). Dieser Befehl wird vom Assembler in den gleichen Opcode übersetzt wie REPE (s.u.) und ist somit ein Pseudobefehl. Dementsprechend stellen einige Debugger REP/REPE in Verbindung mit CMPS und SCAS als REPE dar, um von der ebenfalls möglichen Kombination mit REPNE zu unterscheiden. Mit allen anderen Stringbefehlen wird REP/REPE als REP dargestellt.

  REP hat daher je nach folgendem Stringbefehl unterschiedliche Wirkungen! Folgen CMPS oder SCAS, so arbeitet er absolut gleich wie REPE (siehe dort!), da er in die identische Opcode-Sequenz übersetzt wird, prüft also den Zustand des Zero-Flags ab. Mit allen anderen Stringbefehlen dagegen bewirkt REP lediglich eine Veränderung des Inhalts von CX, ohne daß das Zero-Flag geprüft wird!

- REPE (*REP*eat while *E*qual); Synonym: REPZ (*REP*eat while *Z*ero). Diese Version des REP-Befehls arbeitet vollständig analog zu REP, nur prüft sie nach jeder Dekrementierung von CX, ob das *Zero-Flag* gesetzt ist (daher REPZ; das *Zero-Flag* ist z.B. immer dann gesetzt, wenn zwei verglichene Werte gleich sind, daher auch REPE!). Ist dies nicht der Fall, so wird der folgende Befehl nicht mehr ausgeführt.

REPE führt also den folgenden Stringbefehl so lange aus, wie das *Zero-Flag* gesetzt ist, maximal jedoch die in CX angegebene Anzahl von Durchgängen. Dies erfolgt jedoch nur in Verbindung mit CMPS und SCAS. Mit den anderen Befehlen wird REPE wie REP behandelt, es erfolgt also keine Flagprüfung.

- REPNE (*REP*eat while *N*ot *E*qual); Synonym: REPNZ (*REP*eat while *N*ot Zero). Für diese Variante des REP-Befehls gilt das unter REPE Gesagte – mit umgekehrtem Vorzeichen! Nun wird der folgende Befehl so lange wiederholt, bis das Zero-Flag gesetzt wird, maximal jedoch CX-mal! Auch hier erfolgt die Flagprüfung nur in Verbindung mit CMPS und SCAS! Bei den restlichen Befehlen wird zwar eine andere *Opcode*-Sequenz als bei REP/REPE erzeugt, jedoch verhält sich diese Sequenz exakt wie die erstere (alles andere macht ja auch keinen Sinn!).

Allgemein erfolgt die Iteration nach folgendem Schema:

- Prüfung des Inhalts von CX (bei 32-Bit: ECX). Ist der dort enthaltene Wert 0, so wird die Iteration abgebrochen und die Programmausführung mit dem Befehl fortgesetzt, der auf den zu wiederholenden Befehl folgt.
- Eventuelle Ausführung von Interrupts, falls der Interrupt-Controller Interrupts anfordert.
- Einmalige (!) Ausführung des folgenden Befehls.
- Dekrementieren von CX (ECX) um 1. Hierbei werden *keine* Flags verändert – es handelt sich also *nicht* um ein verkapptes DEC CX!
- Prüfung des Zero-Flags bei REPE (REPZ) und REPNE (REPNZ). Ist es bei REPE nicht gesetzt oder bei REPNE gesetzt, so wird die Iteration abgebrochen und die Programmausführung mit dem auf den zu repetierenden Befehl folgenden Befehl fortgesetzt. Andernfalls wird ein neuer Zyklus begonnen.

Takte

#	8086*	80286*	80386*	80486*	Pentium
1	9+22·n	5+9·n	5+9·n	$5/7+7\cdot n^1$	$7/9+4\cdot n^1$
2	9+22·n	5+9·n	5+9·n	$5/7+7\cdot n^1$	$7/9+4\cdot n^1$
3	-	5+4·n	13+6·n	16+8·n	11+3·n
4	9+11·n	5+4·n	5+4·n	$5/4+4\cdot n^1$	$7/7+3\cdot n^1$
5	9+17·n	5+4·n	5+4·n	$5/13/12\cdot n^2$	$6/13/13\cdot n^2$
6	-	5+4·n	5+12·n	17+5·n	13+4·n
7	9+15·n	5+8·n	5+8·n	$5/7+5\cdot n^1$	$7/9+4\cdot n^1$
8	9+15·n	5+8·n	5+8·n	$5/7+5\cdot n^1$	$7/9+4\cdot n^1$
9	9+10·n	5+3·n	5+5·n	$5/7+4\cdot n^1$	$6/9\cdot n^1$

* n bedeutet bei allen Befehlen die in CX (ECX) enthaltene Anzahl von Iterationen. Beachten Sie bitte, daß die Konstante vor dem »+«-Zeichen die Ausführungszeit des REP/REPc-Befehls darstellt und unabhängig von der Anzahl der Iterationen ist. Das Produkt nach dem »+«-Zeichen gibt die Ausführungszeit des folgenden Stringbefehls an. Dieser ist im allgemeinen in Verbindung mit REP/REPc um einen Takt kürzer als die Ausführungszeit eines Durchgangs des Stringbefehls selbst!

[1] Der erste Wert gibt die Ausführungszeit unter der Bedingung an, daß (E)CX den Wert 0 enthält, beim zweiten Wert enthält (E)CX einen anderen Inhalt.

[2] Der erste Wert gibt die Ausführungszeit unter der Bedingung an, daß (E)CX den Wert 0 enthält. Der zweite Wert gilt, wenn CX den Wert 1 enthält, bei allen anderen Inhalten gilt die dritte Angabe.

Opcodes

#	B1	Bemerkungen
*	F3	REP / REPE
*	F2	REPNE

* Das Präfix $F3 wird in der Verwendung #1, #3, #4, #5, #6, #7 und #9 verwendet (REP = REPE!), das Präfix $F2 bei #2 und #8.

Exceptions Keine

Bemerkungen REP / REPE / REPNE ist ein Befehlspräfix und steht damit unmittelbar vor dem eigentlichen Stringbefehl!

Im Gegensatz zu LOOP / LOOPc kann mit REP / REPcc keine Befehlssequenz repetiert werden!

Beschreibung Seite 69

RET *8086*

Funktion Rücksprung aus einer Routine (»Unterprogramm«) in den aufrufenden Programmteil.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	keine	RET
2	keine	RETF
3	i16	RET 00012h
4	i16	RETF 00034h

Arbeitsweise RET entnimmt dem Stack die Adresse, die der CALL-Befehl dort abgelegt hat und legt sie in CS:IP ab. Somit wird der Programmablauf unmittelbar hinter dem CALL-Befehl fortgesetzt, der für den Aufruf der Routine verantwortlich war.

RET gibt es in zwei Variationen: Der »normale« RET-Befehl führt einen Rücksprung aus Routinen durch, die mittels eines *Near Calls* angesprungen wurden. Diese CALL-Version hat lediglich den Offset der anzuspringenden Routine auf den Stack gelegt, so daß RET in diesem Fall auch nur den Offset vom Stack in das IP-Register kopiert.

Wurde dagegen ein *Far Call* ausgeführt, so hat dieser die vollständige Adresse mit Segment- und Offsetanteil auf den Stack gelegt. In diesem Fall kopiert der RETF-Befehl beide Komponenten vom Stack in CS:IP.

RET/RETF kann auch einen Operanden besitzen. In diesem Operanden kann spezifiziert werden, wie viele Bytes nach dem Rücksprung vom Stack entfernt werden sollen. Einige Hochsprachen wie PASCAL verwenden diesen Befehl, um übergebene Parameter wieder vom Stack zu entfernen. Zur detaillierteren Besprechung siehe Teil 2 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	16	11	10+m	5	2
2	26	15	18+m	13	4
3	20	11	10+m	5	3
4	25	15	18+m	14	4

Opcodes

#	B1	B2	B3
1	C3		
2	CB		
3	C2	i16	
4	CA	i16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	7, 10	0	10	10
	?	13, 14, 27, 28, 29, 39, 44, 45, 66, 67	?	-	./.
#NP	?	5	?	-	./.
#PF	?	1	?	1	./.
#SS	0	9	0	9	9

Bemerkungen

Der RET-Befehl hat im Protected-Mode des 80286 und im Virtual-Mode der 80386-Prozessoren sowie deren Nachfolgern eine Erweiterung erfahren, die die Privilegstufen und Taskwechsel berücksichtigt. Dies soll jedoch in diesem Zusammenhang nicht beschrieben werden.

ROL 8086

Funktion Bitrotation nach links unter Umgehung des Carry-Flags.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	ROL AL,1	
2	m8, 1	ROL BVar, 1	
3	r8, CL	ROL DH, CL	
4	m8, CL	ROL BVar, CL	
5	r8, i8	ROL BL, 008h	ab 80186
6	m8, i8	ROL BVar, 003h	ab 80186
7	r16, 1	ROL AX, 1	
8	m16, 1	ROL WVar, 1	
9	r16, CL	ROL DX, CL	
10	m16, CL	ROL WVar, CL	
11	r16, i8	ROL BX, 012h	ab 80186
12	m16, i8	ROL WVar, 034h	ab 80186
13	r32, 1	ROL EAX, 1	ab 80386
14	m32, 1	ROL DVar, 1	ab 80386
15	r32, CL	ROL EBX, CL	ab 80386
16	m32, CL	ROL DVar, CL	ab 80386
17	r32, i8	ROL ECX, 034h	ab 80386
18	m32, i8	ROL DVar, 012h	ab 80386

Arbeitsweise ROL rotiert den Inhalt des ersten Operanden um die Anzahl von Stellen nach links, die im zweiten Operanden angegeben ist. Möglich ist hierbei die direkte Angabe der Anzahl, jedoch kann der Wert auch indirekt über das CL-Register angegeben werden.

RCL benutzt bei der Bitverschiebung das Carry-Flag *nicht*! Zur detaillierten Darstellung das Ablaufs siehe Teil 1 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	3	3	1
2	15+EA	7	7	4	3
3	8+4·b	5	3	3	4
4	20+4·b+EA	8	7	4	4
5	-	5	3	2	1
6	-	8	7	4	3
7	2	2	3	3	1
8	15+EA	7	7	4	3

9	8+4·b	5	3	3	4
10	20+4·b+EA	8	7	4	4
11	-	5	3	2	1
12	-	8	7	4	3
13	-	-	3	3	1
14	-	-	7	4	3
15	-	-	3	3	4
16	-	-	7	4	4
17	-	-	3	2	1
18	-	-	7	4	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/0				
2	D0	/0	a16			
3	D2	/0				
4	D2	/0	a16			
5	C0	/0	i8			
6	C0	/0	a16		i8	
7	D1	/0				
8	D1	/0	a16			
9	D3	/0				
10	D3	/0	a16			
11	C1	/0	i8			
12	C1	/0	a16		i8	
13	D1	/0				mit Präfix OPSIZE
14	D1	/0	a16			mit Präfix OPSIZE
15	D3	/0				mit Präfix OPSIZE
16	D3	/0	a16			mit Präfix OPSIZE
17	C1	/0	i8			mit Präfix OPSIZE
18	C1	/0	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch RCL, RCR, ROR, SAL, SAR, SHL, SHR.

Beschreibung Seite 43

ROR 8086

Funktion Bitrotation nach rechts unter Umgehung des Carry-Flags.

Flags X X X X O D I T S Z X A X P X C

Die Flags bleiben unverändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	ROR AL,1	
2	m8, 1	ROR BVar, 1	
3	r8, CL	ROR DH, CL	
4	m8, CL	ROR BVar, CL	
5	r8, i8	ROR BL, 008h	ab 80186
6	m8, i8	ROR BVar, 003h	ab 80186
7	r16, 1	ROR AX, 1	
8	m16, 1	ROR WVar, 1	
9	r16, CL	ROR DX, CL	
10	m16, CL	ROR WVar, CL	
11	r16, i8	ROR BX, 012h	ab 80186
12	m16, i8	ROR WVar, 034h	ab 80186
13	r32, 1	ROR EAX, 1	ab 80386
14	m32, 1	ROR DVar, 1	ab 80386
15	r32, CL	ROR EBX, CL	ab 80386
16	m32, CL	ROR DVar, CL	ab 80386
17	r32, i8	ROR ECX, 034h	ab 80386
18	m32, i8	ROR DVar, 012h	ab 80386

Arbeitsweise ROR rotiert den Inhalt des ersten Operanden um die Anzahl von Stellen nach rechts, die im zweiten Operanden angegeben ist. Möglich ist hierbei die direkte Angabe der Anzahl, jedoch kann der Wert auch indirekt über das CL-Register angegeben werden.

ROR benutzt bei der Bitverschiebung das Carry-Flag *nicht*! Zur detaillierten Darstellung das Ablaufs siehe Teil 1 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	3	3	1
2	15+EA	7	7	4	3
3	8+4·b	5	3	3	4
4	20+4·b+EA	8	7	4	4
5	-	5	3	2	1
6	-	8	7	4	3
7	2	2	3	3	1
8	15+EA	7	7	4	3
9	8+4·b	5	3	3	4
10	20+4·b+EA	8	7	4	4
11	-	5	3	2	1

12	-	8	7	4	3
13	-	-	3	3	1
14	-	-	7	4	3
15	-	-	3	3	4
16	-	-	7	4	4
17	-	-	3	2	1
18	-	-	7	4	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/1				
2	D0	/1	a16			
3	D2	/1				
4	D2	/1	a16			
5	C0	/1	i8			
6	C0	/1	a16		i8	
7	D1	/1				
8	D1	/1	a16			
9	D3	/1				
10	D3	/1	a16			
11	C1	/1	i8			
12	C1	/1	a16		i8	
13	D1	/1				mit Präfix OPSIZE
14	D1	/1	a16			mit Präfix OPSIZE
15	D3	/1				mit Präfix OPSIZE
16	D3	/1	a16			mit Präfix OPSIZE
17	C1	/1	i8			mit Präfix OPSIZE
18	C1	/1	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch RCL, RCR, ROL, SAL, SAR, SHL, SHR.

Beschreibung Seite 43

RSM — Pentium

Funktion: Zurückschalten aus dem *System-Management-Mode.*

Flags:

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?	?	?	?	?	?		?		?		?

Die Flags werden in den Zustand zurückgesetzt, der vor dem Eintreten in den *System-Management-Mode* herrschte.

Verwendung:

#	Parameter	Beispiel
	keine	RSM

Arbeitsweise: RSM gibt die Programmausführung an das Programm oder die Betriebssystemroutine zurück, die unterbrochen wurde, als der Prozessor einen *System-Managemen*-Interrupt-*Request* erhielt. Alle Prozessorregister mit Ausnahme der modellspezifischen Register werden mit den Werten beladen, die bei Eintritt in den *System-Management-Mode* gesichert wurden.

Takte:

#	8086	80286	80386	80486	Pentium
	-	-	-	-	83

Opcodes:

#	B1	B2
	0F	AA

Exceptions:

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#UD	-	10	-	10	10

Bemerkungen: Falls bei der Restaurierung der Daten Inkohärenzen entstehen sollten, schaltet der Prozessor in den Shut-Down-State. Dies kann z.B. durch das Setzen von reservierten Bits im Kontrollregister CR4 auf 1 ausgelöst werden oder durch unerlaubte Bitkombinationen im Kontrollregister CR0 (z.B. PG = 1 und PE = 0 oder NW = 1 und CD = 0).

Beschreibung: Seite 148.

SAHF *8086*

Funktion Kopieren des Inhalts von AH in das Flagregister.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
								*	*		*		*		*

Die Flags werden in Abhängigkeit vom Inhalt von AH gesetzt. ACHTUNG: Das Overflow-Flag wird nicht beeinflußt, da es nicht im niederwertigen Byte vom Flagregister codiert wird und daher im Byte in AH auch nicht repräsentiert werden kann!

Verwendung

#	Parameter	Beispiel
	keine	SAHF

Arbeitsweise Dieser Befehl kopiert das Byte im AH-Register in das untere Byte des Flagregisters. Somit werden die Flagzustände von *sign*, *zero*, *auxiliary*, *parity* und *carry* anhand eines Byte-Werts neu gesetzt. Das »Status-Byte« in AH muß bitweise wie folgt codiert sein:

7	6	5	4	3	2	1	0
S	**Z**		**A**		**P**		**C**

Takte

#	8086	80286	80386	80486	Pentium
	4	2	3	2	2

Opcodes

#	B1
	9E

Exception: Keine

Bemerkungen SAHF kann dazu verwendet werden, den Condition Code des Coprozessors in das Flagregister zu laden (siehe FSTSW).

Beschreibung Seite 40

SAL 8086

Funktion (Arithmetische) Bitverschiebung nach links.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		?		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Bitverschiebung gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	SAL AL,1	
2	m8, 1	SAL BVar, 1	
3	r8, CL	SAL DH, CL	
4	m8, CL	SAL BVar, CL	
5	r8, i8	SAL BL, 008h	ab 80186
6	m8, i8	SAL BVar, 003h	ab 80186
7	r16, 1	SAL AX, 1	
8	m16, 1	SAL WVar, 1	
9	r16, CL	SAL DX, CL	
10	m16, CL	SAL WVar, CL	
11	r16, i8	SAL BX, 012h	ab 80186
12	m16, i8	SAL WVar, 034h	ab 80186
13	r32, 1	SAL EAX, 1	ab 80386
14	m32, 1	SAL DVar, 1	ab 80386
15	r32, CL	SAL EBX, CL	ab 80386
16	m32, CL	SAL DVar, CL	ab 80386
17	r32, i8	SAL ECX, 034h	ab 80386
18	m32, i8	SAL DVar, 012h	ab 80386

Arbeitsweise SAL verschiebt den Inhalt des ersten Operanden um die Anzahl von Stellen nach links, die im zweiten Operanden angegeben ist. Möglich ist hierbei die direkte Angabe der Anzahl, jedoch kann der Wert auch indirekt über das CL-Register angegeben werden. SAL benutzt bei der Bitverschiebung das Carry-Flag. Zur detaillierten Darstellung das Ablaufs siehe Teil 1 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	3	3	1
2	15+EA	7	7	4	3
3	8+4·b	5	3	3	4
4	20+4·b+EA	8	7	4	4
5	-	5	3	2	1
6	-	8	7	4	3
7	2	2	3	3	1
8	15+EA	7	7	4	3
9	8+4·b	5	3	3	4
10	20+4·b+EA	8	7	4	4

11	-	5	3	2	1
12	-	8	7	4	3
13	-	-	3	3	1
14	-	-	7	4	3
15	-	-	3	3	4
16	-	-	7	4	4
17	-	-	3	2	1
18	-	-	7	4	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/4				
2	D0	/4	a16			
3	D2	/4				
4	D2	/4	a16			
5	C0	/4	i8			
6	C0	/4	a16		i8	
7	D1	/4				
8	D1	/4	a16			
9	D3	/4				
10	D3	/4	a16			
11	C1	/4	i8			
12	C1	/4	a16		i8	
13	D1	/4				mit Präfix OPSIZE
14	D1	/4	a16			mit Präfix OPSIZE
15	D3	/4				mit Präfix OPSIZE
16	D3	/4	a16			mit Präfix OPSIZE
17	C1	/4	i8			mit Präfix OPSIZE
18	C1	/4	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch RCL, RCR, ROL, ROR, SAR, SHL, SHR.

Beschreibung Seite 45

SAR 8086

Funktion (Arithmetische) Bitverschiebung nach rechts.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		?		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Bitverschiebung gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	SAR AL,1	
2	m8, 1	SAR BVar, 1	
3	r8, CL	SAR DH, CL	
4	m8, CL	SAR BVar, CL	
5	r8, i8	SAR BL, 008h	ab 80186
6	m8, i8	SAR BVar, 003h	ab 80186
7	r16, 1	SAR AX, 1	
8	m16, 1	SAR WVar, 1	
9	r16, CL	SAR DX, CL	
10	m16, CL	SAR WVar, CL	
11	r16, i8	SAR BX, 012h	ab 80186
12	m16, i8	SAR WVar, 034h	ab 80186
13	r32, 1	SAR EAX, 1	ab 80386
14	m32, 1	SAR DVar, 1	ab 80386
15	r32, CL	SAR EBX, CL	ab 80386
16	m32, CL	SAR DVar, CL	ab 80386
17	r32, i8	SAR ECX, 034h	ab 80386
18	m32, i8	SAR DVar, 012h	ab 80386

Arbeitsweise SAR verschiebt den Inhalt des ersten Operanden um die Anzahl von Stellen nach rechts, die im zweiten Operanden angegeben ist. Möglich ist hierbei die direkte Angabe der Anzahl, jedoch kann der Wert auch indirekt über das CL-Register angegeben werden.

SAR benutzt bei der Bitverschiebung das Carry-Flag. Zur detaillierten Darstellung das Ablaufs siehe Teil 1 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	3	3	1
2	15+EA	7	7	4	3
3	8+4·b	5	3	3	4
4	20+4·b+EA	8	7	4	4
5	-	5	3	2	1
6	-	8	7	4	3
7	2	2	3	3	1
8	15+EA	7	7	4	3

9	8+4·b	5	3	3	4
10	20+4·b+EA	8	7	4	4
11	-	5	3	2	1
12	-	8	7	4	3
13	-	-	3	3	1
14	-	-	7	4	3
15	-	-	3	3	4
16	-	-	7	4	4
17	-	-	3	2	1
18	-	-	7	4	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/7				
2	D0	/7	a16			
3	D2	/7				
4	D2	/7	a16			
5	C0	/7	i8			
6	C0	/7	a16		i8	
7	D1	/7				
8	D1	/7	a16			
9	D3	/7				
10	D3	/7	a16			
11	C1	/7	i8			
12	C1	/7	a16		i8	
13	D1	/7				mit Präfix OPSIZE
14	D1	/7	a16			mit Präfix OPSIZE
15	D3	/7				mit Präfix OPSIZE
16	D3	/7	a16			mit Präfix OPSIZE
17	C1	/7	i8			mit Präfix OPSIZE
18	C1	/7	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch RCL, RCR, ROL, ROR, SAL, SHL, SHR.

Beschreibung Seite 45

SBB 8086

Funktion: Subtraktion unter Berücksichtigung des Carry-Flags (*Subtraktion with Borrow*).

Flags:

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Subtraktion gesetzt.

Verwendung:

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	SBB AH, BL	
2	r8, m8	SBB CL, BVar	
3	m8, r8	SBB BVar, CH	
4	r8, i8	SBB AL, $12	
5	m8, i8	SBB BVar, $34	
6	r16, r16	SBB AX, BX	
7	r16, m16	SBB CX, WVar	
8	m16, r16	SBB WVar, DX	
9	r16, i16	SBB AX, $4711	
10	m16, i16	SBB WVar, $0815	
11	r16, i8	SBB DX, $12	
12	m16, i8	SBB WVar, $12	
13	r32, r32	SBB EAX, EBX	ab 80386
14	r32, m32	SBB ECX, DVar	ab 80386
15	m32, r32	SBB DVar, EDX	ab 80386
16	r32, i32	SBB EAX, $1234567	ab 80386
17	m32, i32	SBB DVar, $987654	ab 80386
18	r32, i8	SBB ESI, $12	ab 80386
19	m32, i8	SBB DVar, $34	ab 80386
20	AL,i8	SBB AL, 012h	nur AL!
21	AX, i16	SBB AX, 01234h	nur AX!
22	EAX, i32	SBB EAX, 0123456h	nur EAX!

Arbeitsweise: SBB subtrahiert den Wert des zweiten Operanden vom Wert des ersten Operanden. Anschließend wird, falls das Carry-Flag gesetzt ist, vom Ergebnis 1 abgezogen, andernfalls bleibt die Differenz unverändert. Das Ergebnis dieser Berechnung wird im ersten Operanden abgelegt.

Takte:

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	9+EA	7	6	2	2
3	16+EA	7	7	3	3
4	4	3	2	1	1
5	17+EA	7	7	3	3
6	3	2	2	1	1
7	9+EA	7	6	2	2

8	16+EA	7	7	3	3
9	4	3	2	1	1
10	17+EA	7	7	3	3
11	4	3	2	1	1
12	17+EA	7	7	3	3
13	-	-	2	1	1
14	-	-	6	2	2
15	-	-	7	3	3
16	-	-	2	1	1
17	-	-	7	3	3
18	-	-	2	1	1
19	-	-	7	3	3
20	4	3	2	1	1
21	4	3	2	1	1
22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	1A	/r							
2	1A	/m	a16						
3	18	/m	a16						
4	80	/3	i8						
5	80	/3	a16		i8				
6	1B	/r							
7	1B	/m	a16						
8	19	/m	a16						
9	81	/3	i16						
10	81	/3	a16		i16				
11	83	/3	i8						
12	83	/3	a16		i8				
13	1B	/r							mit Präfix OPSIZE
14	1B	/m	a16						mit Präfix OPSIZE
15	19	/m	a16						mit Präfix OPSIZE
16	81	/3	i32						mit Präfix OPSIZE
17	81	/3	a16		i32				mit Präfix OPSIZE
18	83	/3	i8						mit Präfix OPSIZE
19	83	/3	a16		i8				mit Präfix OPSIZE
20	1C								
21	1D								
22	1D								

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen SBB kann wirksam zur Berechnung von Differenzen benutzt werden, die die Registergröße des Prozessors überschreiten (z.B. zur Subtraktion von Doppelworten bei 8086-Prozessoren oder für Quadworte bei 80386ern). Hierzu wird die erste (»untere«) Hälfte der beiden Summanden mit SUB subtrahiert. Falls ein Überlauf stattfindet, so wird das Carry-Flag gesetzt. Dieser Überlauf kann dann bei der Subtraktion der zweiten (»oberen«) Hälfte der Operanden mittels SBB berücksichtigt werden, ohne Fallunterscheidungen programmieren zu müssen!

Beschreibung Seite 55

SCAS *8086*
SCASB *8086*
SCASW *8086*
SCASD *80386*

Funktion Suche eines Werts in einem String.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis des Vergleichs gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	keine	SCASB	
2a	keine	SCASW	
3a	keine	SCASD	ab 80386
1b	m8	SACS BString	*
2b	m16	SCAS WString	*
3b	m32	SCAS DString	* ab 80386

*ACHTUNG! Bei der Verwendung der allgemeinen Form müssen mehrere Dinge beachtet werden:

- Durch die bloße Angabe der beiden Pseudooperanden wird die Registerkombination ES:DI (ES:EDI) noch nicht korrekt besetzt! Dies hat unbedingt gesondert vor der Verwendung des Stringbefehls zu erfolgen!

- Der Assembler achtet auf die korrekte Angabe des Operanden. So muß als Operand ein String angegeben werden, dessen Segmentanteil in ES steht. Stimmt dieser Segmentanteil der Adresse nicht mit dem in ES stehenden überein, so erfolgt eine Fehlermeldung!
- Der verwendete String muß korrekt definiert worden sein, da nur über dessen Definition der Assembler die benötigte Information über die zu verwendende Datumsgröße erhält! In Verwendung #1b muß daher eine Definition der Art

  ```
  BString DB 128 DUP (?)
  ```

  erfolgen. Durch die Anweisung DB kann der Assembler den Befehl *SCAS BString* in den Befehl *SCASB* übersetzen. Analoges gilt für Verwendung #2b und #3b.

Arbeitsweise

Mit SCAS ist ein Durchsuchen eines Strings möglich. Als String bezeichnet man beim Assembler eine Folge von Variablen gleicher Größe. So gibt es *Byte-Strings*, *Wortstrings* und *Doppelwortstrings*. Die Größe der Strings ist keinen Beschränkungen unterworfen, wird aber durch das gewählte Speichermodell limitiert. So können im Real-Mode die Strings maximal 64 kByte entsprechend der Größe eines Segments sein.

SCAS (*SCA*n *S*tring) ist der Überbegriff für die Befehle SCASB, SCASW und SCASD und wird durch den Assembler in einen der drei Befehle übersetzt. SCAS hat deshalb zwei nur zur Assemblierung benötigte Pseudooperanden, die jedoch nur angeben, ob byteweise (SCASB), wortweise (SCASW) oder doppelwortweise (SCASD) durchsucht werden soll.

Die eigentlichen Operanden des Befehls SCAS werden nicht explizit angegeben. Vielmehr durchsucht SCAS in allen drei Variationen einen String, dessen Adresse in ES:DI (ES:EDI bei SCASD) verzeichnet ist, nach einem Datum, das im Akkumulator (AL, AX oder EAX) steht. Das Durchsuchen erfolgt durch Vergleich des aktuellen Datums im String mit dem Datum im Akkumulator. Wie bei CMP geschieht dies durch eine Subtraktion des Bytes, Worts oder Doppelworts aus ES:DI (ES:EDI) von dem im Akkumulator. Das Ergebnis der Subtraktionen wird auch hier verworfen – wie bei CMP werden nur die Flags anhand des Ergebnisses gesetzt.

SCAS vergleicht pro Durchgang nur jeweils ein Datum. Stringweises Vergleichen und somit Durchsuchen wird daher erst in Verbindung mit einem der Präfixe REPc möglich. Diese Präfixe bewirken, daß SCAS so lange ausgeführt wird, bis ein Abbruchkriterium erfüllt ist (siehe dort).

Damit diese Präfixe korrekt arbeiten können, verändert SACS auch den Inhalt des Registers DI. So wird nach dem Vergleich in Abhängigkeit vom Status des Direction-Flags sowie der Art des Vergleichs der Inhalt dieses Registes um ein Byte (SCASB), zwei Bytes (SCASW) oder vier Bytes (SCASD) erhöht (DF = 0) oder verringert (DF = 1). Auf diese Weise zeigt das Indexregister DI nach dem Vergleich *auf das nächste zu prüfende Datum*!

Takte

#	8086	80286	80386	80486	Pentium
1	15	7	7	6	4
2	15	7	7	6	4
3	-	-	7	6	4

Opcodes

#	B1	Bemerkungen
1	AE	
2	AF	
3	AF	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Vor der Verwendung von SCAS müssen die Register AL (AX; EAX) ES und DI (EDI) mit dem Suchwert und der Adresse des Strings geladen werden.

Beschreibung Seite 70

SETcc *80386*

Funktion Bedingtes Setzen eines Bytes.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r8	SETA AH
2	m8	SETNZ BVar

Arbeitsweise Bei dem bedingten Setzen eines Bytes handelt es sich im Prinzip um einen Vorgang, der wie ein bedingter Sprung abläuft. Der Unterschied besteht jedoch darin, daß in Abhängigkeit von der Bedingung nicht der Inhalt von IP verändert wird, um eine neue Zieladresse für die Programmfortsetzung zu erhalten, sondern ein beliebiges Byte, um die Bedingung zu codieren. So speichert SETcc an dem zu spezifizierenden Byte immer dann eine 1, wenn

die Bedingung erfüllt ist. Andernfalls wird mit einer 0 signalisiert, daß die Bedingung nicht zutrifft. Analog zu den bedingten Sprungbefehlen gibt es die gleichen SETcc-Befehle:

- SETA, SETAE, SETB, SETBE, SETNA, SETNAE, SETNB, SETNBE; Bedingungen, die bei einem Vergleich vorzeichenloser Zahlen bestehen. Ausgewertet werden das Carry- und/oder das Zero-Flag.
- SETG, SETGE, SETL, SETLE, SETNG, SETNGE, SETNL, SETNLE; Bedingungen, die bei einem Vergleich vorzeichenbehafteter Zahlen bestehen. Ausgewertet werden das Overflow-, das Zero- und/oder das Sign-Flag.
- SETE, SETNE, SETNZ, SETZ; Bedingungen, die bei einem Vergleich von Zahlen allgemein bestehen. Ausgewertet wird das Zero-Flag.
- SETC, SETNC, SETNO, SETNP, SETNS, SETO, SETP, SETPE, SETPO, SETS; Bedingungen, die einzelne Flags betreffen und von verschiedenen Befehlen erzeugt werden.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	7/5*	4/3*	2/1*

* Die jeweils ersten Werte gelten für den Fall, daß die Bedingung erfüllt ist. Der zweite Wert gilt für den Fall, daß die Bedingung nicht erfüllt ist.
Alle Taktangaben gelten für alle SET-Befehle, unabhängig von der Bedingung.

Opcodes

#	B1	B2	B3	B4	B5		Bemerkungen
	0F	97	/r	*a16*		*m*	SETA*
	0F	93	/r	*a16*		*m*	SETAE*
	0F	92	/r	*a16*		*m*	SETB*
	0F	96	/r	*a16*		*m*	SETBE*
	0F	92	/r	*a16*		*m*	SETC*
	0F	94	/r	*a16*		*m*	SETE*
	0F	9F	/r	*a16*		*m*	SETG*
	0F	9D	/r	*a16*		*m*	SETGE*
	0F	9C	/r	*a16*		*m*	SETL*
	0F	9E	/r	*a16*		*m*	SETLE*

0F	96	/r	*a16*	*m*	SETNA*
0F	92	/r	*a16*	*m*	SETNAE*
0F	93	/r	*a16*	*m*	SETNB*
0F	97	/r	*a16*	*m*	SETNBE*
0F	93	/r	*a16*	*m*	SETNC*
0F	95	/r	*a16*	*m*	SETNE*
0F	9E	/r	*a16*	*m*	SETNG*
0F	9C	/r	*a16*	*m*	SETNGE*
0F	9D	/r	*a16*	*m*	SETNL*
0F	9F	/r	*a16*	*m*	SETNLE*
0F	91	/r	*a16*	*m*	SETNO*
0F	9B	/r	*a16*	*m*	SETNP*
0F	99	/r	*a16*	*m*	SETNS*
0F	95	/r	*a16*	*m*	SETNZ*
0F	90	/r	*a16*	*m*	SETO*
0F	9A	/r	*a16*	*m*	SETP*
0F	9A	/r	*a16*	*m*	SETPE*
0F	9B	/r	*a16*	*m*	SETPO*
0F	98	/r	*a16*	*m*	SETS*
0F	94	/r	*a16*	*m*	SETZ*

* Die kursiv gedruckten Bytefolgen beziehen sich auf die Versionen des Befehls, die als Operanden eine Speicherstelle angeben.

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 133

SGDT *80286*

Funktion Speichern einer *globalen Deskriptortabelle*.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	Pointer	SGDT StrukturVar.

Arbeitsweise Der Operand ist ein Zeiger auf eine Struktur, die 6 Bytes Daten umfaßt. Diese 6 Bytes Daten sind eine 32-Bit-Basisadresse der zu speichernden *Globalen-Deskriptortabelle* sowie ein 16-Bit-Limit (Tabellengröße in Bytes). Falls die Operandengröße 32 Bit ist, werden die gesamten 32 Bit der Basisadresse und die 16 Bit des Limits (die unteren beiden Bytes) aus dem *Globalen Deskriptortabellen-Register* (GDTR) des Prozessors ausgelesen und in die Struktur Bytes 3, 4, 5 und 6 (Basisadresse) bzw. 1 und 2 (Limit) eingetragen. Ist die Operandengröße dagegen 16 Bit, so werden lediglich die Bytes 3, 4 und 5 der Struktur mit der (24-Bit-) Basisadresse belegt, und Byte 6 wird gelöscht. Für das Limit werden die gesamten beiden unteren Bytes der Struktur (16 Bit) verwendet.

Takte

#	8086	80286	80386	80486	Pentium
1	-	11	9	10	4
2	-	11	9	10	4

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	
1	0F	01	/0	a16				
2	0F	01	/0	a32				mit OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
#UD	-	4	-	4	4

Bemerkungen

Die Unterscheidung 16-Bit-/32-Bit-Operanden ist traditionell bedingt. So besitzt bereits der 80286 einen Protected-Mode mit allen wichtigen Registern und Konzepten. Er hatte jedoch lediglich 24 Adreßleitungen, so daß nur 24 Bits für die Adressen zur Verfügung standen. Über die Operandengröße kann man daher auf diese Situation Rücksicht nehmen.

SGDT ist ein Befehl, der für Betriebssysteme reserviert ist! Er macht nämlich nur dann Sinn, wenn mit LGDT eine neue *globale Deskriptortabelle* eingelesen werden soll, die alte aber weiterhin zur Verfügung stehen soll. Dies sollte aber nicht durch Anwendungen unter Umgehung des Betriebssystems erfolgen!

Beschreibung

Seite 185.

SHL *8086*

Funktion

(Logische) Bitverschiebung nach links.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		?		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Bitverschiebung gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	SHL AL,1	
2	m8, 1	SHL BVar, 1	
3	r8, CL	SHL DH, CL	
4	m8, CL	SHL BVar, CL	
5	r8, i8	SHL BL, 008h	ab 80186
6	m8, i8	SHL BVar, 003h	ab 80186
7	r16, 1	SHL AX, 1	
8	m16, 1	SHL WVar, 1	
9	r16, CL	SHL DX, CL	
10	m16, CL	SHL WVar, CL	

11	r16, i8	SHL BX, 012h	ab 80186
12	m16, i8	SHL WVar, 034h	ab 80186
13	r32, 1	SHL EAX, 1	ab 80386
14	m32, 1	SHL DVar, 1	ab 80386
15	r32, CL	SHL EBX, CL	ab 80386
16	m32, CL	SHL DVar, CL	ab 80386
17	r32, i8	SHL ECX, 034h	ab 80386
18	m32, i8	SHL DVar, 012h	ab 80386

Arbeitsweise SHL arbeitet absolut gleich wie SAL (siehe dort).

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	3	3	1
2	15+EA	7	7	4	3
3	8+4·b	5	3	3	4
4	20+4·b+EA	8	7	4	4
5	-	5	3	2	1
6	-	8	7	4	3
7	2	2	3	3	1
8	15+EA	7	7	4	3
9	8+4·b	5	3	3	4
10	20+4·b+EA	8	7	4	4
11	-	5	3	2	1
12	-	8	7	4	3
13	-	-	3	3	1
14	-	-	7	4	3
15	-	-	3	3	4
16	-	-	7	4	4
17	-	-	3	2	1
18	-	-	7	4	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/4				
2	D0	/4	a16			
3	D2	/4				
4	D2	/4	a16			
5	C0	/4	i8			
6	C0	/4	a16	i8		
7	D1	/4				
8	D1	/4	a16			
9	D3	/4				
10	D3	/4	a16			
11	C1	/4	i8			

12	C1	/4	a16	i8	
13	D1	/4			mit Präfix OPSIZE
14	D1	/4	a16		mit Präfix OPSIZE
15	D3	/4			mit Präfix OPSIZE
16	D3	/4	a16		mit Präfix OPSIZE
17	C1	/4	i8		mit Präfix OPSIZE
18	C1	/4	a16	i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen SHL und SAL sind identische Befehle.

Siehe auch RCL, RCR, ROL, ROR, SAL, SAR, SHR.

Beschreibung Seite 44

SHLD *80386*

Funktion (Logische) Bitverschiebung nach links mit »doppelter Genauigkeit«.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				*	*		?		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Bitverschiebung gesetzt.

Verwendung

#	Parameter	Beispiel
1	r16, r16, i8	SHLD AX, BX, 008h
2	m16, r16, i8	SHLD WVar, CX, 012h
3	r16, r16, CL	SHLD CX, DX, CL
4	m16, r16, CL	SHLD WVar, DI, CL
5	r32, r32, i8	SHLD EAX, EBX, 016h
6	m32, r32, i8	SHLD DVar, ECX, 020h
7	r32, r32, CL	SHLD ECX, EDX, CL
8	m32, r32, CL	SHLD DVar, EDI, CL

Arbeitsweise

SHLD ist ein »ganz normaler« SHL-Befehl, der lediglich etwas erweitert wurde. Er besitzt drei Operanden, wobei der dritte Operand angibt, um wie viele Positionen die Inhalte des ersten Operanden verschoben werden sollen. Dies kann direkt über die Angabe einer Konstanten oder indirekt über das CL-Register erfolgen.

SHLD verschiebt nun die entsprechende Anzahl von Bits aus dem ersten Operanden nach links heraus. Anschließend wird aus dem zweiten Operanden die gleiche Anzahl von Bits von links beginnend in die frei gewordenen Positionen verschoben. Der zweite Operand bleibt unverändert.

Zur detaillierteren Besprechung dieses Befehls siehe Teil 1 des Buches!

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	3	2	4
2	-	-	7	3	4
3	-	-	3	2	4
4	-	-	7	3	5
5	-	-	3	2	4
6	-	-	7	3	4
7	-	-	3	2	4
8	-	-	7	3	5

Opcodes

#	B1	B2	B3	B4		Bemerkungen
1	0F	A4	/r	i8		
2	0F	A4	/m	a16	i8	
3	0F	A5	/r			
4	0F	A5	/m	a16		
5	0F	A4	/r	i8		mit Präfix OPSIZE
6	0F	A4	/m	a16	i8	mit Präfix OPSIZE
7	0F	A5	/r			mit Präfix OPSIZE
8	0F	A5	/m	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung

Seite 136

SHR 8086

Funktion (Logische) Bitverschiebung nach rechts.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		?		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Bitverschiebung gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, 1	SHR AL,1	
2	m8, 1	SHR BVar, 1	
3	r8, CL	SHR DH, CL	
4	m8, CL	SHR BVar, CL	
5	r8, i8	SHR BL, 008h	ab 80186
6	m8, i8	SHR BVar, 003h	ab 80186
7	r16, 1	SHR AX, 1	
8	m16, 1	SHR WVar, 1	
9	r16, CL	SHR DX, CL	
10	m16, CL	SHR WVar, CL	
11	r16, i8	SHR BX, 012h	ab 80186
12	m16, i8	SHR WVar, 034h	ab 80186
13	r32, 1	SHR EAX, 1	ab 80386
14	m32, 1	SHR DVar, 1	ab 80386
15	r32, CL	SHR EBX, CL	ab 80386
16	m32, CL	SHR DVar, CL	ab 80386
17	r32, i8	SHR ECX, 034h	ab 80386
18	m32, i8	SHR DVar, 012h	ab 80386

Arbeitsweise SHR verschiebt den Inhalt des ersten Operanden um die Anzahl von Stellen nach rechts, die im zweiten Operanden angegeben ist. Möglich ist hierbei die direkte Angabe der Anzahl, jedoch kann der Wert auch indirekt über das CL-Register angegeben werden.

SHR benutzt bei der Bitverschiebung das Carry-Flag *nicht*! Zur detaillierten Darstellung das Ablaufs siehe Teil 1 des Buches.

Takte

#	8086	80286	80386	80486	Pentium
1	2	2	3	3	1
2	15+EA	7	7	4	3
3	8+4·b	5	3	3	4
4	20+4·b+EA	8	7	4	4
5	-	5	3	2	1
6	-	8	7	4	3
7	2	2	3	3	1

8	15+EA	7	7	4	3
9	8+4·b	5	3	3	4
10	20+4·b+EA	8	7	4	4
11	-	5	3	2	1
12	-	8	7	4	3
13	-	-	3	3	1
14	-	-	7	4	3
15	-	-	3	3	4
16	-	-	7	4	4
17	-	-	3	2	1
18	-	-	7	4	3

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	D0	/5				
2	D0	/5	a16			
3	D2	/5				
4	D2	/5	a16			
5	C0	/5	i8			
6	C0	/5	a16		i8	
7	D1	/5				
8	D1	/5	a16			
9	D3	/5				
10	D3	/5	a16			
11	C1	/5	i8			
12	C1	/5	a16		i8	
13	D1	/5				mit Präfix OPSIZE
14	D1	/5	a16			mit Präfix OPSIZE
15	D3	/5				mit Präfix OPSIZE
16	D3	/5	a16			mit Präfix OPSIZE
17	C1	/5	i8			mit Präfix OPSIZE
18	C1	/5	a16		i8	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Siehe auch RCL, RCR, ROL, ROR, SAL, SAR, SHL.

Beschreibung Seite 44

SHRD 80386

Funktion (Logische) Bitverschiebung nach rechts mit »doppelter Genauigkeit«.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				?				*	*		?		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Bitverschiebung gesetzt.

Verwendung

#	Parameter	Beispiel
1	r16, r16, i8	SHRD AX, BX, 008h
2	m16, r16, i8	SHRD WVar, CX, 012h
3	r16, r16, CL	SHRD CX, DX, CL
4	m16, r16, CL	SHRD WVar, DI, CL
5	r32, r32, i8	SHRD EAX, EBX, 016h
6	m32, r32, i8	SHRD DVar, ECX, 020h
7	r32, r32, CL	SHRD ECX, EDX, CL
8	m32, r32, CL	SHRD DVar, EDI, CL

Arbeitsweise SHRD ist ein »ganz normaler« SHR-Befehl, der lediglich etwas erweitert wurde. Er besitzt drei Operanden, wobei der dritte Operand angibt, um wie viele Positionen die Inhalte des ersten Operanden verschoben werden sollen. Dies kann direkt über die Angabe einer Konstanten oder indirekt über das CL-Register erfolgen.

SHRD verschiebt nun die entsprechende Anzahl von Bits aus dem ersten Operanden nach rechts heraus. Anschließend wird aus dem zweiten Operanden die gleiche Anzahl von Bits von rechts beginnend in die frei gewordenen Positionen verschoben. Der zweite Operand bleibt unverändert.

Zur detaillierteren Besprechung dieses Befehls siehe Teil 1 des Buches!

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	3	2	4
2	-	-	7	3	4
3	-	-	3	2	4
4	-	-	7	3	5
5	-	-	3	2	4
6	-	-	7	3	4
7	-	-	3	2	4
8	-	-	7	3	5

Opcodes

#	B1	B2	B3	B4		Bemerkungen
1	0F	AC	/r	i8		
2	0F	AC	/m	a16	i8	
3	0F	AD	/r			
4	0F	AD	/m	a16		

5	0F	AC	/r	i8		mit Präfix OPSIZE
6	0F	AC	/m	a16	i8	mit Präfix OPSIZE
7	0F	AD	/r			mit Präfix OPSIZE
8	0F	AD	/m	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 136

SIDT *80286*

Funktion Speichern einer *Interrupt-Deskriptortabelle*.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	Pointer	SIDT StrukturVar.

Arbeitsweise Der Operand ist ein Zeiger auf eine Struktur, die 6 Bytes Daten umfaßt. Diese 6 Bytes Daten sind eine 32-Bit-Basisadresse der zu speichernden Interrupt *Deskriptor table* sowie ein 16-Bit-Limit (Tabellengröße in Bytes). Falls die Operandengröße 32 Bit ist, werden die gesamten 32 Bit der Basisadresse und die 16 Bit des Limits (die unteren beiden Bytes) aus dem Interrupt-Deskriptortabellen-*Register* (IDTR) des Prozessors ausgelesen und in die Struktur Bytes 3, 4, 5 und 6 (Basisadresse) bzw. 1 und 2 (Limit) eingetragen. Ist die Operandengröße dagegen 16 Bit, so werden lediglich die Bytes 3, 4 und 5 der Struktur mit der (24-Bit-) Basisadresse belegt, und Byte 6 wird gelöscht. Für das Limit werden die gesamten beiden unteren Bytes der Struktur (16 Bit) verwendet.

Takte

#	8086	80286	80386	80486	Pentium
1	-	12	9	10	4
2	-	12	9	10	4

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	
1	0F	01	/1	a16				
2	0F	01	/1	a32				mit OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
#UD	-	4	-	4	4

Bemerkungen

Die Unterscheidung 16-Bit-/32-Bit-Operanden ist traditionell bedingt. So besitzt bereits der 80286 einen Protected-Mode mit allen wichtigen Registern und Konzepten. Er hatte jedoch lediglich 24 Adreßleitungen, so daß nur 24 Bits für die Adressen zur Verfügung standen. Über die Operandengröße kann man daher auf diese Situation Rücksicht nehmen.

SIDT ist ein Befehl, der für Betriebssysteme reserviert ist! Er macht nämlich nur dann Sinn, wenn mit LIDT eine neue Interrupt-Deskriptortabelle eingelesen werden, die alte aber weiterhin zur Verfügung stehen soll. Dies sollte aber nicht durch Anwendungen unter Umgehung des Betriebssystems erfolgen!

Beschreibung

Seite 231.

SLDT *80286*

Funktion

Speichern einer *lokalen Deskriptortabelle*.

Flags

X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16	SLDT BX
2	m16	SLDT WVar

Arbeitsweise

Der Operand ist ein Register oder eine Variable, die einen Selektor aufnehmen soll, der auf eine *lokale Deskriptortabelle* zeigt. SLDT kopiert diesen Selektor aus dem *lokalen Deskriptortabellen-Register* (LDTR)

Takte

#	8086	80286	80386	80486	Pentium
1	-	2	2	2	2
2	-	3	2	3	2

Opcodes

#	B1	B2	B3	B4	B5
1	0F	00	/0		
2	0F	00	/0	a16	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	1
#GP	0	3, 8, 20	0	./.	./.
#PF	?	1	?	./.	./.
#SS	0	1	0	./.	./.
#UD	-	-	-	1	1

Bemerkungen Falls die Operandengröße 32 Bit beträgt, wird der 16-Bit-Selektor aus dem LDTR in die unteren 16 Bits des Operanden kopiert. Die oberen 16 Bit werden gelöscht.

SLDT ist ein Betriebssystembefehl und sollte daher nicht von Applikationen benutzt werden, da er nur in Kombination mit einem LLDT wirklich Sinn macht!

Beschreibung Seite 188.

SMSW *80286*

Funktion Speichern des Maschinen-Statuswortes.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16	SMSW BX
2	m16	SMSW WVar

Arbeitsweise SMSW kopiert die Bits 15 bis 0 des Kontrollregisters CR0 des Prozessors in den Operanden.

Takte

#	8086	80286	80386	80486	Pentium
1	-	2	2	2	4
2	-	3	2	3	4

Opcodes

#	B1	B2	B3	B4	B5
1	0F	01	/4		
2	0F	01	/4	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen

Falls der Operand ein 32-Bit-Register ist, bleiben die 16 oberen Bits nach dem Kopieren des Statusworts in die 16 unteren Bits des Registers undefiniert. Falls der Operand eine Speicherstelle angibt, werden die 16 Bits unabhängig von der Operandengröße als 16-Bit-Statuswort an die Speicheradresse geschrieben.

LMSW ist nur im Rahmen von Betriebssystemroutinen sinnvoll einsetzbar, obwohl es sich nicht um eine privilegierte Funktion handelt. Sie dient nur der Abwärtskompatibilität. Ab dem 80386 sollte anstelle dieser Funktion der MOV-Befehl mit seinen Erweiterungen auf die Kontrollregister verwendet werden!

Beschreibung

Seite 946.

STC *8086*

Funktion

Explizites Setzen des Carry-Flags.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
															1

Das Carry-Flag wird explizit gesetzt.

Takte

#	8086	80286	80386	80486	Pentium
	2	2	2	2	2

Opcodes

#	B1
	F9

Exceptions

Keine

Beschreibung

Seite 40

STD 8086

Funktion Explizites Setzen des Direction-Flags.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
					1										

Das Direction-Flag wird explizit gesetzt.

Takte

#	8086	80286	80386	80486	Pentium
	2	2	2	2	2

Opcodes

#	B1
	FD

Exception: Keine

Beschreibung Seite 41

STI 8086

Funktion Freischaltung der Interrupt-Möglichkeit.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
						1									

Das Interrupt-Enable-Flag wird explizit gesetzt.

Takte

#	8086	80286	80386	80486	Pentium
	2	2	3	5	7

Opcodes

#	B1
	FB

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	33	0	33	-

Beschreibung Seite 41

STOS 8086
STOSB 8086
STOSW 8086
STOSD 80386

Funktion Speichern eines Datums in einen String.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	keine	STOSB	
2a	keine	STOSW	
3a	keine	STOSD	ab 80386
1b	m8	STOS BString	*
2b	m16	STOS WString	*
3b	m32	STOS DString	* ab 80386

*ACHTUNG! Bei der Verwendung der allgemeinen Form müssen mehrere Dinge beachtet werden:

- Durch die bloße Angabe des Pseudooperanden wird die Registerkombination ES:DI (ES:EDI) noch nicht korrekt besetzt! Dies hat unbedingt gesondert vor der Verwendung des Stringbefehls zu erfolgen!
- Der Assembler achtet auf die korrekte Angabe des Operanden. So muß als Operand ein String angegeben werden, dessen Segmentanteil in ES steht. Stimmt dieser Segmentanteil der Adresse nicht mit dem in ES stehenden überein, so erfolgt eine Fehlermeldung.
- Der verwendete String muß korrekt definiert worden sein, da nur über seine Definition der Assembler die benötigte Information über die zu verwendende Datumsgröße erhält! In Verwendung #1b muß daher eine Definition der Art

  ```
  BString DB 128 DUP (?)
  ```

  erfolgen. Durch die Anweisung DB kann der Assembler den Befehl *STOS BString* in den Befehl *STOSB* übersetzen! Analoges gilt für Verwendung #2b und #3b!

Arbeitsweise Mit STOS ist ein Beschreiben eines Strings möglich. Als String bezeichnet man beim Assembler eine Folge von Variablen gleicher Größe. So gibt es *Byte-Strings*, *Wortstrings* und *Doppelwortstrings*. Die Größe der Strings ist keinen Beschränkungen unterworfen, wird aber durch das gewählte Speichermodell limitiert. So können im Real-Mode die Strings maximal 64 kByte entsprechend der Größe eines Segments sein.

STOS (*STO*re into *S*tring) ist der Überbegriff für die Befehle STOSB, STOSW und STOSD und wird durch den Assembler in einen der drei Befehle übersetzt. STOS hat deshalb einen nur zur Assemblierung benötigten Pseudooperanden, der jedoch nur angibt, ob byteweise (STOSB), wortweise (STOSW) oder doppelwortweise (STOSD) gespeichert werden soll.

Die eigentlichen Operanden des Befehls STOS werden nicht explizit angegeben. Vielmehr belädt STOS in allen drei Variationen einen String, dessen Adresse in ES:DI (ES:EDI bei STOSD) verzeichnet ist, mit einem Byte aus AL (STOSB), einem Wort aus AX (STOSW) oder einem Doppelwort aus EAX (STOSD).

STOS speichert pro Durchgang nur jeweils ein Byte, Wort oder Doppelwort an die Stelle, die durch die Adresse in ES:DI adressiert wird. Stringweises Speichern wird daher erst in einer Schleife möglich. Der Präfix REPc, der bei CMPS, INS, OUTS und SCAS sinnvoll angewendet werden kann, ist hier nur eingeschränkt zu gebrauchen, da mit jedem Durchgang das im Akkumulator stehende Datum geschrieben wird. REPc macht also als Präfix nur dann Sinn, wenn ein String mit einem konstanten Datum vorbelegt werden soll.

Damit Schleifen einfach programmierbar sind, verändert STOS auch den Inhalt des Registers DI (EDI). So wird nach dem Auslesen in Abhängigkeit vom Status des Direction-Flags sowie von der Art des Datums der Inhalt dieser Register um ein Byte (STOSB), zwei Bytes (STOSW) oder vier Bytes (STOSD) erhöht (DF = 0) oder verringert (DF = 1). Auf diese Weise zeigt das Indexregister DI nach dem Beschreiben *auf die nächste zu beschreibende Speicherstelle*!

Takte

#	8086	80286	80386	80486	Pentium
1	11	3	4	5	3
2	11	3	4	5	3
3	-	-	4	5	3

Opcodes

#	B1	Bemerkungen
1	AA	
2	AB	
3	AB	mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Vor der Verwendung von STOS müssen der Akkumulator (AL; AX bzw. EAX) mit dem Datum und die Register ES und DI (EDI) mit der Adresse des Strings beladen werden.

Beschreibung Seite 65

STR 80286

Funktion Speichern des *Task-Registers*.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	r16	STR BX
2	m16	STR WVar

Arbeitsweise STR kopiert den im *Task-Register* stehenden Selektor in den Operanden.

Takte

#	8086	80286	80386	80486	Pentium
1	-	2	23	2	2
2	-	3	27	3	2

Opcodes

#	B1	B2	B3	B4	B5
1	0F	00	/1		
2	0F	00	/1	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	./.	./.
#PF	?	1	?	./.	./.
#SS	0	1	0	./.	./.
#UD	-	-	-	1	1

Bemerkungen Der in den Operanden kopierte Selektor zeigt auf das Task-Segment des augenblicklich aktiven Task!

Falls die Operandengröße 32 Bit beträgt, wird der 16-Bit-Selektor aus dem *Task-Register* in die unteren 16 Bits des Operanden kopiert, die oberen 16 Bits werden gelöscht.

Die Operandengröße hat keinen Einfluß auf die Funktion. Es werden in jedem Fall 16 Bits an Informationen kopiert.

Beschreibung Seite 946.

SUB 8086

Funktion Subtraktion zweier Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden in Abhängigkeit vom Ergebnis der Subtraktion gesetzt.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	SUB AH, BL	
2	r8, m8	SUB CL, BVar	
3	m8, r8	SUB BVar, CH	
4	r8, i8	SUB AL, 012h	
5	m8, i8	SUB BVar, 034h	
6	r16, r16	SUB AX, BX	
7	r16, m16	SUB CX, WVar	
8	m16, r16	SUB WVar, DX	
9	r16, i16	SUB AX, 04711h	
10	m16, i16	SUB WVar, 00815h	
11	r16, i8	SUB DX, 012h	
12	m16, i8	SUB WVar, 012h	
13	r32, r32	SUB EAX, EBX	ab 80386
14	r32, m32	SUB ECX, DVar	ab 80386
15	m32, r32	SUB DVar, EDX	ab 80386
16	r32, i32	SUB EAX, 012345678h	ab 80386
17	m32, i32	SUB DVar, 098765432h	ab 80386
18	r32, i8	SUB ESI, 012h	ab 80386
19	m32, i8	SUB DVar, 034h	ab 80386
20	AL, i8	SUB AL, 012h	nur AL!
21	AX, i16	SUB AX, 01234h	nur AX!
22	EAX, i32	SUB EAX, 012345678h	nur EAX!

Arbeitsweise SUB subtrahiert den Wert des zweiten Operanden vom Wert des ersten Operanden und legt das Ergebnis im ersten Operanden ab.

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	9+EA	7	6	2	2
3	16+EA	7	7	3	3
4	4	3	2	1	1
5	17+EA	7	7	3	3
6	3	2	2	1	1
7	9+EA	7	6	2	2
8	16+EA	7	7	3	3
9	4	3	2	1	1
10	17+EA	7	7	3	3
11	4	3	2	1	1

12	17+EA	7	7	3	3
13	-	-	2	1	1
14	-	-	6	2	2
15	-	-	7	3	3
16	-	-	2	1	1
17	-	-	7	3	3
18	-	-	2	1	1
19	-	-	7	3	3
20	4	3	2	1	1
21	4	3	2	1	1
22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	2A	/r							
2	2A	/m	a16						
3	28	/m	a16						
4	80	/5	i8						
5	80	/5	a16				i8		
6	2B	/r							
7	2B	/m	a16						
8	29	/m	a16						
9	81	/5	i16						
10	81	/5	a16			i16			
11	83	/5	i8						
12	83	/5	a16				i8		
13	2B	/r							mit Präfix OPSIZE
14	2B	/m	a16						mit Präfix OPSIZE
15	29	/m	a16						mit Präfix OPSIZE
16	81	/5	i32						mit Präfix OPSIZE
17	81	/5	a16			i32			mit Präfix OPSIZE
18	83	/5	i8						mit Präfix OPSIZE
19	83	/5	a16				i8		mit Präfix OPSIZE
20	2C								
21	2D								
22	2D								

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Bemerkungen Im Unterschied zu SBB wird das Carry-Flag bei der Subtraktion nicht berücksichtigt, wohl aber anhand des Ergebnisses der Subtraktion verändert.

Beschreibung Seite 55

SYSENTER *Pentium II*

Funktion Teil der »Fast System Call«-Fähigkeiten, die mit dem Pentium II eingeführt wurden.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	SYSENTER

Arbeitsweise SYSENTER belädt CS:EIP und SS:ESP mit bestimmten Werten aus bestimmten modellspezifischen Registern des Prozessors.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2
	0F	34

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	61	0	61	1

Bemerkungen SYSENTER und SYSEXIT sind Teile der mit dem Pentium II-Prozessor eingeführten »Fast System Call«-Fähigkeit, mit der hardwareseitig das schnelle Umschalten in Ring 0 (CPL = 0) der Privilegstufen unterstützt wird. SYSENTER und SYSEXIT entnehmen oder versorgen sog. modellspezifische Register (MSR) des Prozessors, die mit WRMSR und RDMSR angesprochen werden können, mit bestimmten Werten für CS:EIP und SS:ESP. Zu Einzelheiten dieser sehr spezifischen Register siehe die Dokumentationen von Intel.

Beschreibung Seite 150.

SYSEXIT — Pentium II

Funktion: Teil der »Fast System Call«-Fähigkeiten, die mit dem Pentium II eingeführt wurden.

Flags:

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung:

#	Parameter	Beispiel
	keine	SYSENTER

Arbeitsweise: SYSEXIT belädt CS:EIP und SS:ESP mit bestimmten Werten aus bestimmten modellspezifischen Registern des Prozessors.

Takte:

#	8086	80286	80386	80486	Pentium
	-	-	-	-	-

Opcodes:

#	B1	B2
	0F	35

Exceptions:

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	32, 61	0	32, 61	1

Bemerkungen: SYSENTER und SYSEXIT sind Teile der mit dem Pentium II-Prozessor eingeführten »Fast System Call«-Fähigkeit, mit der hardwareseitig das schnelle Umschalten in Ring 0 (CPL = 0) der Privilegstufen unterstützt wird. SYSENTER und SYSEXIT entnehmen oder versorgen sogenannte modellspezifische Register (MSR) des Prozessors, die mit WRMSR und RDMSR angesprochen werden können, mit bestimmten Werten für CS:EIP und SS:ESP. Zu Einzelheiten dieser sehr spezifischen Register siehe die Dokumentationen von Intel.

Beschreibung: Seite 150

TEST *8086*

Funktion Logischer Vergleich zweier Operanden.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				0				*	*		?		*		0

Die Flags werden in Abhängigkeit vom Ergebnis des Tests gesetzt. Das Overflow- und das Carry-Flag werden explizit gelöscht.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, i8	TEST AH, 012h	
2	m8, i8	TEST BVar, 034h	
3	r8, r8	TEST BL, BH	
4	m8, r8	TEST BVar, CL	
5	r16, i16	TEST AX, 01234h	
6	m16, i16	TEST WVar, 04321h	
7	r16, r16	TEST BX, CX	
8	m16, r16	TEST WVar, DX	
9	r32, i32	TEST EAX, 012345678h	ab 80386
10	m32, i32	TEST DVar, 087654321h	ab 80386
11	r32, r32	TEST EBX, EAX	ab 80386
12	m32, r32	TEST DVar, EDX	ab 80386
13	AL, i8	TEST AL, 098h	nur AL!
14	AX, i16	TEST AX, 09876h	nur AX!
15	EAX, i32	TEST EAX, 012345678h	nur EAX!

Arbeitsweise TEST vergleicht zwei Operanden über eine logische UND-Verknüpfung miteinander. Hierbei werden die in den beiden Operanden übergebenen Werte wie bei AND bitweise verknüpft und die Flags anhand des Ergebnisses gesetzt. Das Ergebnis selbst wird dann verworfen, der Inhalt der Operanden ändert sich also nicht.

Takte

#	8086	80286	80386	80486	Pentium
1	5	3	2	1	1
2	11+EA	6	5	2	2
3	3	2	2	1	1
4	9+EA	6	5	2	2
5	5	3	2	1	1
6	11+EA	6	5	2	2
7	3	2	2	1	1
8	9+EA	6	5	2	2
9	-	-	2	1	1

10	-	-	5	2	2
11	-	-	2	1	1
12	-	-	5	2	2
13	4	3	2	1	1
14	4	3	2	1	1
15	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	F6	/0	i8		
2	F6	/0	a16	i8	
3	84	/r			
4	84	/m	a16		
5	F7	/0	i16		
6	F7	/0	a16	i16	
7	85	/r			
8	85	/m	a16		
9	F7	/0	i32		mit Präfix OPSIZE
10	F7	/0	a16	i32	mit Präfix OPSIZE
11	85	/r			mit Präfix OPSIZE
12	85	/m	a16		mit Präfix OPSIZE
13	A8	i8			
14	A9	i16			
15	A9	i32			mit Präfix OPSIZE

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Beschreibung Seite 29

UD2 *Pentium Pro*

Funktion Dieser Befehl generiert einen Invalid Opcode. Er dient zum Austesten von Software während der Entwicklungsphase.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keiner	UD2

Arbeitsweise UD2 erzeugt eine Invalid-Instruction-Exception. Darüber hinaus verhält es sich wie NOP.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2
	0F	0B

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#UD	-	11	-	11	11

Beschreibung Seite 149

VERR
VERW *80286*

Funktion Feststellung der Lese/Schreib-Erlaubnis für Segmente.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									?						

Das Zero-Flag ist gesetzt, falls das Segment lesbar (VERR) oder beschreibbar (VERW) ist. Andernfalls ist das Zero-Flag gelöscht.

Verwendung

#	Parameter	Beispiel
1	r16	VERR BX
2	m16	VERR WVar
3	r16	VERW CX
4	m16	VERW WVar

Arbeitsweise

VERR/VERW erhalten im Operanden einen Selektor, der auf einen Deskriptor in der globalenen Deskriptortabelle oder der lokalenl Deskriptortabelle zeigt. VERR prüft, ob das durch den Deskriptor beschriebene Segment lesbar ist, während VERW feststellt, ob es beschreibbar ist. Ist dies der Fall, wird das Zero-Flag gesetzt.

Takte

#	8086	80286	80386	80486	Pentium
1	-	14	10	11	7
2	-	16	11	11	7
3	-	14	15	11	7
4	-	16	16	11	7

Opcodes

#	B1	B2	B3	B4	B5
1	0F	00	/4		
2	0F	00	/4	a16	
3	0F	00	/5		
4	0F	00	/5	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	./.	./.
#GP	0	3, 8	0	./.	./.
#PF	?	1	?	./.	./.
#SS	0	1	0	./.	./.
#UD	-	-	-	1	1

Bemerkungen

VERR/VERW führen eine Reihe von Prüfungen durch. So wird geprüft, ob der Selektor 0 ist, auf gültige Einträge in der GDT oder LDT zeigt und ein Daten- oder Codesegment beschreibt. Da Codesegmente grundsätzlich nicht beschreibbar sind, prüft VERW noch zusätzlich, ob ein Codesegment überprüft werden soll. Anschließend wird geprüft, ob das Segment unter der augenblicklichen Privilegstufe (CPL = *Current Privileg Level*) angesprochen werden darf.

VERR/VERW arbeiten mit den gleichen Befehlsfolgen, die eingesetzt werden, wenn Inhalte der Segmentregister DS, ES, FS oder GS verändert werden und wenn auf die durch sie beschriebenen Segmente zugegriffen wird. Da jedoch die Registerbelegungen und Zugriffe nicht tatsächlich erfolgen, erhält ein Programm die Möglichkeit, *vor* einem Umschalten auf ein anderes Segment zu prüfen, ob Zugriffe darauf zu Problemen führen werden oder nicht.

Beschreibung

Seite 235.

WAIT *8086*

Funktion Warten auf den Coprozessor.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	WAIT

Arbeitsweise WAIT veranlaßt den Prozessor, die Programmausführung zu unterbrechen und die *Busy*-Leitung zu überwachen, bis über diese Leitung ein Signal übermittelt wird. Dann nimmt der Prozessor die Programmausführung mit dem nächsten folgenden Befehl wieder auf.

Takte

#	8086	80286	80386	80486	Pentium
	4+5·n *	3	6	1-3	1

* n gibt die Rate an, mit der der 8086 die *Busy*-Leitung überwacht.

Opcodes

#	B1
	9B

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	3	-	3	3

Bemerkungen WAIT dient zur Synchronisierung der Aktivitäten von Prozessor und Coprozessor.

Beschreibung Seite 79

WBINVD *80486*

Funktion Cache leeren und dann für ungültig erklären.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	WBINVD

Arbeitsweise WBINVD löscht den Inhalt des internen Caches, nachdem dessen Inhalt in den Speicher zurückgeschrieben wurde und signalisiert externen Caches, den Inhalt ebenfalls zu sichern und dann zu löschen.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	5	2000+

Opcodes

#	B1	B2
	0F	09

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	32	0	2	./.

Bemerkungen

WBINVD ist eine privilegierte Funktion. Das heißt, sie kann nur innerhalb der höchsten Privilegstufe (CPL = 0) ausgeführt werden. Dies ist ausschließlich in Betriebssystemmodulen der Fall.

WBINVD wartet ebenso wie INVD nicht darauf, daß externe Caches ihren Inhalt tatsächlich gesichert und gelöscht haben. Nach dem Sichern und Löschen des internen Caches wird die Programmausführung sofort fortgesetzt.

Eine Modifikation des Befehls, der die Daten vor dem Löschen *nicht* sichert, ist INVD.

WRMSR *Pentium*

Funktion

Schreibender Zugriff auf die modellspezifischen Register des Pentium.

Flags

X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	keine	WRMSR

Arbeitsweise

WRMSR beschreibt das *modellspezifische Register* (MSR) des Prozessors, dessen Nummer in ECX übergeben wird, mit der Information, die in der Registerkombination EDX:EAX verzeichnet ist, wobei die oberen 32 Bits aus dem EDX-, die unteren aus dem EAX-Register kopiert werden.

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	30-45

Opcodes

#	B1	B2
	0F	30

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#GP	0	32, 59	0	2	59

Bemerkungen

ACHTUNG: Reservierte oder nicht benutzte Bits des MSR sollten grundsätzlich mit vor Ausführung von WRMSR auszulesenden Werten besetzt werden!

Vor der Verwendung von WRMSR sollte mit Hilfe von CPUID festgestellt werden, ob der Prozessor über modellspezifische Register und damit über den Befehl WRMSR verfügt. Ist das der Fall, so ist Bit 5 in EDX nach dem Aufruf von CPUID mit EAX = 1 gesetzt.

Falls das spezifizierte Register weniger als 64 Bits an Informationen enthält, so sind die nicht implementierten Bits in EDX:EAX undefiniert.

WRMSR muß mit höchster Privilegstufe (CPL = 0) ausgeführt werden, andernfalls wird eine General-Protection-Exception ausgeführt. Dies erfolgt auch, wenn eine reservierte oder undefinierte Registeradresse in ECX übergeben wird.

Beschreibung

Seite 148.

XADD *80486*

Funktion

Austausch zweier Operanden und Addition.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				*				*	*		*		*		*

Die Flags werden aufgrund des Resultats der Addition verändert.

Verwendung

#	Parameter	Beispiel
1	r8, r8	XADD AL, AH
2	m8, r8	XADD BVar, DL
3	r16, r16	XADD AX, BX
4	m16, r16	XADD WVar, CX
5	r32, r32	XADD ESI, EDI
6	m32, r32	XADD DVar, EBX

Arbeitsweise

XADD vertauscht die in den Operanden übergebenen Werte und legt dann im ersten Operanden die Summe aus beiden Werten ab. Die Wirkung von *XADD dest, source* läßt sich wie folgt simulieren:

```
temp := dest;
dest := source;
source := temp;
dest := dest + source
```

Takte

#	8086	80286	80386	80486	Pentium
1	-	-	-	3	3
2	-	-	-	4	4
3	-	-	-	3	3
4	-	-	-	4	4
5	-	-	-	3	3
6	-	-	-	4	4

Opcodes

#	B1	B2	B3	B4	B5	Bemerkungen
1	0F	C0	/r			
2	0F	C0	/m	a16		
3	0F	C1	/r			
4	0F	C1	/m	a16		
5	0F	C1	/r			mit Präfix OPSIZE
6	0F	C1	/m	a16		mit Präfix OPSIZE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 147

XCHG *8086*

Funktion Austausch zweier Operanden.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	XCHG AH, BL	
2	r8, m8	XCHG CL, BVar	
3	m8, r8	XCHG BVar, CH	
4	16, r16	XCHG AX, BX	
5	r16, m16	XCHG CX, WVar	
6	m16, r16	XCHG WVar, DX	
7	r32, r32	XCHG EAX, EBX	ab 80386

8	r32, m32	XCHG ECX, DVar	ab 80386
9	m32, r32	XCHG DVar, EDX	ab 80386
10	AX,r16	XCHG AX, CX	nur ALX!
11	r16, AX	XCHG BX, AX	nur AX!
12	EAX, r32	XCHG EAX, ESI	nur EAX!
13	r32, EAX	XCHG EDX, EAX	nur EAX!

Arbeitsweise

XCHG (e*XCH*anGe) vertauscht den Inhalt des zweiten Operanden (»Quelle«) mit dem des ersten Operanden (»Ziel«).

Takte

#	8086	80286	80386	80486	Pentium
1	4	3	3	3	3
2	17+EA	5	5	5	3
3	17+EA	5	5	5	3
4	4	3	3	3	3
5	17+EA	5	5	5	3
6	17+EA	5	5	5	3
7	-	-	3	3	3
8	-	-	5	5	3
9	-	-	5	5	3
10	3	3	3	3	2
11	3	3	3	3	2
12	-	3	3	3	2
13	-	3	3	3	2

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	86	/r			
2	86	/m	a16		
3	86	/m	a16		
4	87	/r			
5	87	/m	a16		
6	87	/m	a16		
7	87	/r			mit Präfix OPSIZE
8	87	/m	a16		mit Präfix OPSIZE
9	87	/m	a16		mit Präfix OPSIZE
10	90+i				
11	90+i				
12	90+i				mit Präfix OPSIZE
13	90+i				mit Präfix OPSIZE

* i kann Werte zwischen 0 und 7 annehmen und definiert das zu verwendende Register. Die Register werden hierbei wie folgt codiert:

	0	1	2	3	4	5	6	7
r8	AL	CL	DL	BL	AH	CH	DH	BH
r16	AX	CX	DX	BX	SP	BP	SI	DI
r32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 21

XLAT *8086*
XLATB *8086*

Funktion Tabelleneintrag feststellen.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
1	keine	XLATB
2	m8	XLAT BVar

Arbeitsweise XLATB kopiert aus einer Tabelle ein Byte und legt es in AL ab. Hierzu werden ein Index, der auf den gewünschten Tabelleneintrag zeigt, sowie die Adresse einer Tabelle verwendet.

Beide Parameter werden nicht explizit vorgegeben, sondern müssen vor dem Aufruf von XLATB schon bekannt sein. So erwartet XLATB die Adresse der Tabelle in DS:BX, wobei der Segmentanteil der Tabellenadresse in DS, ihr Offsetanteil in BX verzeichnet sein muß. Der Index auf den Tabelleneintrag muß in AL stehen.

XLAT ist der allgemeine Befehl, dem ein Pseudooperand übergeben wird. Auch bei diesem Befehl müssen Tabellenoffset und Index schon in BX bzw. AL stehen. Über den Pseudooperanden wird lediglich ermöglicht, die implizite Segmentvorgabe des XLATB-Befehls (DS) zu übergehen. So ermittelt der Assembler aus dem Befehl *XLAT BVar* lediglich das Segment, das zu *BVar* gehört und stellt ggf. das entsprechende Segmentpräfix dem Opcode für XLAT/XLATB als *Segment-Override-Präfix* voran.

Takte

#	8086	80286	80386	80486	Pentium
1	11	5	5	4	4
2	11	5	5	4	4

Opcodes

#	B1	Bemerkungen
1	D7	
2	D7	mit Segmentpräfix

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Beschreibung Seite 22

XOR *8086*

Funktion Logische exklusive ODER-Verknüpfung.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
				0				*	*		?		*		0

Die Flags werden in Abhängigkeit vom Ergebnis der Verknüpfung gesetzt. Carry-Flag und Overflow-Flag werden explizit gelöscht.

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	r8, r8	XOR AH, BL	
2	r8, m8	XOR CL, BVar	
3	m8, r8	XOR BVar, CH	
4	r8, i8	XOR AL, 012h	
5	m8, i8	XOR BVar, 034h	
6	r16, r16	XOR AX, BX	
7	r16, m16	XOR CX, WVar	
8	m16, r16	XOR WVar, DX	
9	r16, i16	XOR AX, 04711h	
10	m16, i16	XOR WVar, 00815h	
11	r16, i8	XOR DX, 012h	ab 80386
12	m16, i8	XOR WVar, 012h	ab 80386
13	r32, r32	XOR EAX, EBX	ab 80386
14	r32, m32	XOR ECX, DVar	ab 80386
15	m32, r32	XOR DVar, EDX	ab 80386

16	r32, i32	XOR EAX, 012345678h	ab 80386
17	m32, i32	XOR DVar, 098765432h	ab 80386
18	r32, i8	XOR ESI, 012h	ab 80386
19	m32, i8	XOR DVar, 034h	ab 80386
20	AL, i8	XOR AL, 012h	nur AL!
21	AX, i16	XOR AX, 01234h	nur AX!
22	EAX, i32	XOR EAX, 012345678h	nur EAX!

Arbeitsweise

XOR führt eine logische exklusive ODER-Verknüpfung durch. Hierbei wird bitweise der erste Operand mit dem zweiten Operanden exklusiv ODER-verknüpft, das Resultat wird dann in den ersten Operanden eingetragen. Der zweite Operand bleibt unverändert!

Takte

#	8086	80286	80386	80486	Pentium
1	3	2	2	1	1
2	9+EA	7	6	2	2
3	16+EA	7	7	3	3
4	4	3	2	1	1
5	17+EA	7	7	3	3
6	3	2	2	1	1
7	9+EA	7	6	2	2
8	16+EA	7	7	3	3
9	4	3	2	1	1
10	17+EA	7	7	3	3
11	-	-	2	1	1
12	-	-	7	3	3
13	-	-	2	1	1
14	-	-	6	2	2
15	-	-	7	3	3
16	-	-	2	1	1
17	-	-	7	3	3
18	-	-	2	1	1
19	-	-	7	3	3
20	4	3	2	1	1
21	4	3	2	1	1
22	-	-	2	1	1

Opcodes

#	B1	B2	B3	B4	B5	B6	B7	B8	Bemerkungen
1	32	/r							
2	32	/m	a16						
3	30	/m	a16						
4	80	/6	i8						
5	80	/6	a16		i8				
6	33	/r							
7	33	/m	a16						
8	31	/m	a16						
9	81	/6	i16						
10	81	/6	a16		i16				
11	83	/6	i8						
12	83	/6	a16		i8				
13	33	/r							mit Präfix OPSIZE
14	33	/m	a16						mit Präfix OPSIZE
15	31	/m	a16						mit Präfix OPSIZE
16	81	/6	i32						mit Präfix OPSIZE
17	81	/6	a16		i32				mit Präfix OPSIZE
18	83	/6	i8						mit Präfix OPSIZE
19	83	/6	a16		i8				mit Präfix OPSIZE
20	34								
21	35								
22	35								

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#PF	?	1	?	1	./.
#SS	0	1	1	1	1

Beschreibung Seite 46

36 Coprozessorbefehle

In der Titelzeile ist für jeden Befehl angegeben, ab welchem Coprozessor er verfügbar ist. Bei der Besprechung der Coprozessorbefehle kommen folgende Abkürzungen zum Einsatz:

B *Busy-Flag*

C3 – C0 *Condition-Code* Bits 3 – 0

S *Stack-Fault* ab dem 80387, davor nicht definiert, *Sign-Flag* beim 80x86
P *Precision-Exception-Flag* beim 80x87, *Parity-Flag* beim 80x86
U *Underflow-Exception-Flag*
O *Overflow-Exception-Flag*
Z *Zero-Exception-Flag* beim 80x87, *Zero-Flag* beim 80x86
D *Denormalized-Operand-Flag*
I *Invalid-Operation-Flag*

Die Flags des Coprozessors werden bitweise gemäß ihrer Position im *Statuswort* des Coprozessors dargestellt. Zur Vereinfachung werden auch die betroffenen Flags des Prozessors angegeben, die nach der Sequenz *FSTSW WordVar; MOV AX,WordVar; SAHF* durch die korrespondierenden Bits des Condition Codes verändert werden können. Bei den Flagangaben bezeichnet ein * ein durch den Befehl manipuliertes Flag, bei dem die Stellung vom Funktionsergebnis abhängig ist. Ein Leer-Eintrag steht für ein nicht verändertes Flag, ein undefinierter Zustand des Flags wird durch ein *?* und explizit gesetzte oder gelöschte Flags werden mit *0* und *1* symbolisiert.

Falls eine Operation das Invalid-Operation-Flag setzt, um anzuzeigen, daß die Operation mit den aktuellen Operanden nicht möglich ist, wird eine spezielle NaN erzeugt, die *negative* qNaN indefinite (undefiniert) mit der Bitfolge 1.10…0 in der Mantisse. Dies erfolgt in folgenden Situationen:

- jegliche arithmetischen Operationen mit einem nicht unterstützten Format (beim 80387, 80486/80487 und Pentium also pNaNs und unnormale Zahlen).
- Addition von Unendlichkeiten mit entgegengesetztem Vorzeichen oder Subtraktion von Unendlichkeiten mit gleichem Vorzeichen.
- Multiplikation von 0 mit ∞.
- Division von ∞ durch ∞ oder 0 durch 0.
- FPREM/FPREM1 mit 0 in ST(1) oder ∞ im TOS.
- Trigonometrische Operationen mit ∞ als Argument.
- FSQRT und FYL2X mit negativen Operanden oder FYL2XP1 mit Operanden < –1.0.
- FXCH, wenn eines oder beide Register als *empty* markiert sind.
- FIST(P) und FBSTP bei leerem Register, mit NaN oder ∞ im TOS oder beim Überschreiten des Darstellungsbereichs für Integer bzw. BCDs.

Das Zero-Exception-Flag wird gesetzt, wenn bei F(I)DIV(R)(P), FYL2X und FXTRACT versucht wird, durch 0 zu dividieren. In diesen Fällen wird eine Unendlichkeit erzeugt: bei den Divisionsbefehlen mit einem Vorzeichen, das durch XOR der Vorzeichen der beiden Operanden erzeugt wird, bei FYL2X mit entgegengesetztem Vorzeichen des Operanden, der nicht 0 ist und bei FXTRACT mit negativem Vorzeichen in ST(1).

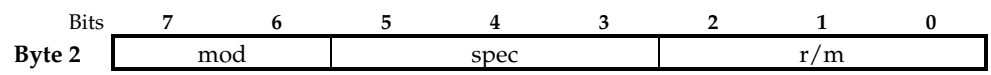
Weitere Symbole:

shortreal 32-Bit-Realzahl im Speicher
longreal 64-Bit-Realzahl im Speicher
tempreal 80-Bit-Realzahl im Speicher oder Stackregister
word 16-Bit-Integer im Speicher
shortint 32-Bit-Integer im Speicher
longint 64-Bit-Integer im Speicher
BCD gepackte BCD mit 18 Ziffern; Umfang: 10 Byte
TOS Top-of-SStack; identisch mit ST(0) und ST
ST(i) i-tes Stackregister. Werte zwischen 0 und 7

Die Einflüsse des Befehls auf den Stack-Pointer des Coprozessors werden wie folgt angegeben:

+ 1 Inkrement um 1, gleichbedeutend mit einem Poppen des Stacks.
± 0 neutral; der Stack-Pointer wird nicht verändert.
- 1 Dekrement um 1, gleichbedeutend mit einem Pushen des Stacks.
EA *effektive Adresse*; zur Besprechung siehe Einleitung zu den Prozessorreferenzen.

Es werden auch die Opcodes angegeben, die zu jedem Befehl gehören. Soweit einem Befehl auch Speicheradressen als Parameter übergeben werden können, werden diese in zwei Bytes nach den ersten beiden Bytes des Opcodes als *B3* und *B4* angegeben. In diesem Fall besteht das zweite Byte des Opcodes aus einer Codezahl, die wie folgt zusammengesetzt wird:



mod *Modus*; zwei Bits, die die Art der folgenden Adresse angeben:

00 Es folgt *keine* Adressenangabe.
01 Es folgt eine 8-Bit-Adresse, die vorzeichenbehaftet auf 16 Bit ausgedehnt wird. Es werden *zwei* Bytes Adresse angegeben!
10 Es folgt eine vollständige 16-Bit-Adresse mit der üblichen Intel-Konvention: zunächst das *lo-byte*, dann das *hi-byte*.
11 r/m wird als ST(i)-Feld behandelt, d.h. der Code gibt direkt die Registernummer des Stacks an.

spec *Specifier*; diese Bits werden zusätzlich zur Codierung der Coprozessorbefehle verwendet und haben mit Speicherzugriffen nichts zu tun (s. u.).

r/m *Register/Memory*; drei Bits (Bit 2, 1 und 0 des Bytes), die Aufschluß über die Art der Adressierung ermöglichen:

000 EA = DS:[BX + SI + 16-Bit-*disp*]
001 EA = DS:[BX + DI + 16-Bit-*disp*]
010 EA = SS:[BP + SI + 16-Bit-*disp*]
011 EA = SS:[BP + DI + 16-Bit-*disp*]
100 EA = DS:[SI + 16-Bit-*disp*]
101 EA = DS:[DI + 16-Bit-*disp*]
110 EA = SS:[BP + 16-Bit-*disp*]. *Ausnahme*: *mod* = 00
→ EA = 16-Bit-Adresse
111 EA = DS:[BX + 16-Bit-*disp*]

falls *mod* = 11 bezeichnet *r/m* die Nummer des zu verwendenden Stackregisters.

/0 .. /7 Angabe über die Bit-Stellung im *spec*-Feld des *mod-spec-r/m*-Byte eines Coprozessorbefehls. Diese Bits codieren zusätzlich zu dem eigentlichen, vorangehenden Code-Byte den Coprozessorbefehl. So ist z.B. das erste Byte im Opcode für die Befehle FIADD, FICOM und FICOMP gleich: $DE. Durch unterschiedlich gesetzte Bits im *spec*-Feld des *mod-spec-r/m*-Bytes, was ja zusätzlich noch für die Adressierung zuständig ist, können die Befehle unterschieden werden. Bei FIADD ist *spec* = 0, bei FICOM 2 und bei FICOMP 3. Dies wird im Opcode durch die Symbole /0, /2 und /3 repräsentiert. Achtung: /0 heißt *nicht*, daß das *mod-spec-r/m*-Byte = 0 ist, sondern lediglich, daß in diesem Byte das *spec*-Feld = 0 ist!

Im Rahmen der 32-Bit-Adressierung kann auch bei den Coprozessorbefehlen ein *sib*-Byte notwendig werden. Dies ist dann der Fall, wenn *r/m* = 100b ist.

Bei einigen Befehlen werden bei den Taktangaben zur Ausführungsgeschwindigkeit des Pentium zwei Werte angegeben, die durch einen Schrägstrich »/« getrennt werden. Die Unterschiede beruhen auf der (verglichen mit den Coprozessorvorläufern) geänderten Arbeitsweise der Floating-Point-Unit des Pentium. So gibt der erste Wert die sogenannte Latenz (*latency*) an, der zweite den Durchsatz (*througput*). Letzterer kann aufgrund des *Pipelinings* geringer sein.

Bei einigen FPU-Befehlen können durch falsche Operanden oder andere Bedingungen Ausnahmesituationen auftreten, sogenannte Exception*s*. Bei der Besprechung der einzelnen Befehle sind die bei diesem Befehl möglichen Exception*s* aufgelistet, sie sind vom Betriebsmodus unabhängig. Es werden neben der Art der Exception auch deren mögliche Ursachen in Form einer Codezahl angegeben, die Sie anhand folgender Auflistung decodieren können (für die Decodierung der eventuell beteiligten CPU-Exceptions siehe die Vorbemerkungen bei der Besprechung der CPU-Befehle):

#D

1 Der Quelloperand ist denormalisiert.
2 Der Zieloperand ist denormalisiert.
3 Ein oder beide Operanden sind denormalisiert.

#IS

1 Es erfolgte ein Stack-Unterlauf.
2 Es erfolgte ein Stack-Überlauf.

#IA

1 Der Quelloperand ist eine sNaN oder ein nicht unterstütztes Format.
2 Der Quelloperand ist leer, enthält eine NaN, ±∞, ein nicht unterstütztes Format oder einen Wert, der nicht mit 18 BCD-Ziffern dargestellt werden kann.
3 Der Quelloperand ist negativ (Ausnahme: -0).
4 Der Quelloperand ist zu groß für das Zielformat.
5 Der Operand ist ∞.
6 Die Operanden sind Unendlichkeiten mit gleichem Vorzeichen.
7 Die Operanden sind Unendlichkeiten mit ungleichem Vorzeichen.
8 Der eine Operand ist ±0, der andere ±∞.
9 Ein oder beide Operanden sind NaNs oder weisen ein nicht unterstütztes Format auf.
10 Ein oder beide Operanden sind sNaNs (keine qNaNs!) oder weisen ein nicht unterstütztes Format auf.
11 Der Quelloperand ist eine sNaN, der Divisor ist 0, der Dividend ∞ oder ein nicht unterstütztes Format.
12 Das Register ist als empty markiert.
13 Division von ±∞ durch ±∞ oder ±0 durch ±0.

#O

1 Das Ergebnis ist zu groß für das Zielformat.

#P

1 Das Ergebnis kann im Zielformat nicht korrekt dargestellt werden

#U

1 Das Ergebnis ist zu klein für das Zielformat

#Z

1 Division des Zieloperanden durch ±0, wenn der Zieloperand nicht den Wert ±0 hat.
2 Division des Quelloperanden durch ±0, wenn der Quelloperand nicht den Wert ±0 hat.
3 Der TOS (=ST(0)) enthält den Wert ±0.

FABS 8087

Funktion FABS bestimmt den Absolutwert einer Zahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	0	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Stack-Fault und das Invalid-Operation-Flag, falls ein Stack-Underflow stattfindet. Andere Flags werden *nicht* gesetzt, selbst wenn im TOS eine NaN vorgefunden wird!

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FABS

Takte

#	8087	80287	80387	80487	Pentium
	10-17	10-17	22	3	1

Opcodes

#	B1	B2
	D9	E1

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	./.	#IA	./.	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FABS wirkt nur auf den TOS.

Beschreibung Seite 108

FADD *8087*

Funktion Addition zweier Realzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*
(80x86-Flags)	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	ST, ST(i)	FADD ST, ST(4)
2	ST(i), ST	FADD ST(2), ST
3	shortreal	FADD ShortRealVar
4	longreal	FADD LongRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	70-100	70-100	23-34	8-20	3/1
2	70-100	70-100	23-24	8-20	3/1
3	90-120+EA	90-120	24-32	8-20	3/1
4	95-125+EA	95-125	29-37	8-20	3/1

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	D8	C0+i			i = Registernummer
2	DC	C0+i			i = Registernummer
3	D8	/0	a16		
4	DC	/0	a16		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 7	#IS 1	#O ./.	#P 1	#U 1	#Z ./.

Bemerkungen FADD ist der allgemeine Befehl, um zwei Zahlen zu addieren. FADD addiert Integer, BCDs und Reals, solange nur Stackregister als Operanden angegeben werden (also Fall 1 und 2 in obiger Liste). Der Hintergrund ist, daß alle diese Zahlen intern als 80-Bit-TEMPREAL dargestellt werden. Darüber hinaus ist mit FADD auch das Addieren einer Zahl aus dem Speicher zum TOS möglich. *Dies ist aber nur mit Realzahlen möglich*, wobei FADD automatisch die Konvertierung des *Realformat*s in die TEMPREAL-Darstellung vornimmt. *FADD ist daher nicht auf Integer anwendbar* – hierfür gibt es den Befehl FIADD, der seinerseits für die korrekte Konvertierung zuständig ist.

Beschreibung Seite 95

FADDP *8087*

Funktion Addition zweier Realzahlen und Poppen des Stacks

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST(i), ST	FADDP ST(2), ST

Takte

#	8087	80287	80387	80487	Pentium
	75-105	75-105	23-34	8-20	3/1

Opcodes

#	B1	B2	Bemerkungen
	DE	C0+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 7	#IS 1	#O ./.	#P 1	#U 1	#Z ./.

Bemerkungen FADDP arbeitet wie FADD. Lediglich nach der Addition erfolgt ein Inkrementieren des Stack-Pointers sowie ein Markieren des *Bottom-of-Stack* als *empty*.

Beschreibung Seite 95

FBLD *8087*

Funktion FBLD lädt eine (gepackte, 18-stellige) BCD in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

Z			P		C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	BCD	FBLD BcdVar

Takte

#	8087	80287	80387	80487	Pentium
	290-310	290-310	45-90	70-103	48-58

Opcodes

#	B1	B2	B3	B4
	DE	/4	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception											
	#D	./.	#IA	./.	#IS	2	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen

FBLD arbeitet nur mit BCDs korrekt. Der Befehl interpretiert die 10 Bytes an der als Parameter übergebenen Speicherstelle als gepackte BCD mit 18 Stellen und transformiert sie in das interne TEMPREAL-Format! Aus diesem Grunde kann man mit FBLD keine Realzahlen oder Integer laden. Hierfür dienen die Befehle FLD und FILD.

FBLD verringert den Stack-Pointer um 1 und führt somit ein Pushen des Stacks durch.

Beschreibung

Seite 109

FBSTP *8087*

Funktion

FBSTP speichert eine TEMPREAL im TOS als (gepackte, 18-stellige) BCD.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*					*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer

+ 1

Verwendung

#	Parameter	Beispiel
	BCD	FBSTP BcdVar

Takte

#	8087	80287	80387	80487	Pentium
	520-540+EA	520-540+EA	512-534	172-176	148-154

Opcodes

#	B1	B2	B3	B4
	DF	/6	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception											
	#D	./.	#IA	2	#IS	1	#O	./.	#P	1	#U	./.	#Z	./.

Bemerkungen FBSTP arbeitet nur mit BCDs korrekt. Der Befehl interpretiert die TEMPREAL im TOS als gepackte BCD mit 18 Stellen und transformiert sie daher in das BCD-Format. Anschließend wird sie in die 10-Byte-Variable gespeichert, deren Adresse als Operand übergeben wird. Zusätzlich wird in jedem Fall der Stack gepoppt.

Beschreibung Seite 109

FCHS *8087*

Funktion FCHS kehrt das Vorzeichen eines Werts im TOS um.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	0	?		*						*

(80x86-Flags)

Z		P		C

Der Coprozessor setzt das Stack-Fault- und das Invalid-Operation-Flag, falls ein Stack-Underflow stattfindet. Andere Flags werden *nicht* gesetzt, selbst wenn im TOS eine NaN vorgefunden wird!

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FCHS

Takte

#	8087	80287	80387	80487	Pentium
	10-17	10-17	24-25	6	1

Opcodes

#	B1	B2
	D9	E0

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception											
	#D	./.	#IA	./.	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FCHS wirkt nur auf den TOS.

Beschreibung Seite 108

FCLEX *8087*
FNCLEX

Funktion FCLEX löscht alle Exception-Flags, falls sie nach einer Operation gesetzt sein sollten.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	?	?								

(80x86-Flags)

	Z				P		C								

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FCLEX
	keine	FNCLEX

Takte

#	8087	80287	80387	80487	Pentium
	2-8	2-8	11	7	9*

* zuzügl. mindestens 1 Takt für FWAIT im Falle von FCLEX.

Opcodes

#	B1	B2
	DB	E2

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

Bemerkungen FNCLEX ist der gleiche Befehl wie FCLEX. Es wird lediglich anstelle des dem Befehl FCLEX vorangehenden WAIT-Befehls ein NOP eingestreut, so daß der Prozessor nicht auf den Coprozessor warten muß.

Beschreibung Seite 112

FCMOVcc *Pentium Pro*

Funktion Bedingtes Kopieren einer Gleitkommazahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?								

(80x86-Flags)

Z			P		C

Der Coprozessor setzt C1 auf 0, falls ein Stack-Underflow stattgefunden hat. C0, C2 und C3 sind undefiniert.

Stack-Pointer -1

Verwendung

#	Parameter	Beispiel	Bemerkungen
	ST(0), ST(i)	FCMOVU ST(0), ST(5)	

Arbeitsweise FCMOVcc arbeitet wie CMOVcc, mit dem Unterschied daß als Operanden Gleitkommazahlen zur Verwendung kommen. Ziel ist immer der TOS, Quelle eines der restlichen Register des Coprozessors. In Abhängigkeit der Flags des EFlagregisters wird der Kopiervorgang durchgeführt oder nicht. Zur Verwendung kommen natürlich nur Bedingungen, die auf vorzeichenbehaftete Zahlen anwendbar sind:

- FCMOVB; kopieren, wenn kleiner (CF = 0), wenn nicht größer oder gleich.
- FCMOVE; kopieren, wenn gleich (ZF = 1).
- FCMOVBE; kopieren, wenn kleiner oder gleich (CF = 1 oder ZF = 1), wenn nicht größer.

- FCMOVU; kopieren, wenn ungeordnet (PF = 1).
- FCMOVNB; kopieren, wenn nicht kleiner (CF = 0), wenn größer oder gleich.
- FCMOVNE; kopieren, wenn nicht gleich (ZF = 0).
- FCMOVNBE; kopieren, wenn nicht kleiner oder gleich (CF = 0 und ZF = 0), wenn größer.
- FCMOVNU; kopieren, wenn nicht ungeordnet (PF = 0).

Takte

#	8086	80286	80386	80486	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2	B3	B4	Befehle
	DA	C0+i			FCMOVB
	DA	C8+i			FCMOVE
	DA	D0+i			FCMOVBE
	DA	D8+i			FCMOVU
	DB	C0+i			FCMOVNB
	DB	C8+i			FCMOVNE
	DB	D0+i			FCMOVNBE
	DB	D8+i			FCMOVNU

i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	./.	#IA	./.	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen Nicht alle Prozessoren der Pentium-Pro-Familie unterstützen diesen Befehl. Ob er bei einem bestimmten Prozessor unterstützt wird oder nicht, kann mit der CPUID-Instruktion festgestellt werden. Falls sowohl das CMOV- als auch das FPU-Flag gesetzt sind, wird FCMOVcc unterstützt.

Beschreibung Seite 149

FCOM 8087

Funktion Vergleich zweier Realzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				(*)	(*)	*		*					*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Falls das Invalid-Operation-Flag *nicht* gesetzt ist, werden lediglich die Flags C3 und C0 verändert. Mögliche Zustände sind:

C3	C0	Bedeutung
0	0	Operand 1 > Operand 2
0	1	Operand 1 < Operand 2
1	0	Operand 1 = Operand 2
1	1	Operanden nicht vergleichbar

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	ST(i)	FCOM ST(4)
2	shortreal	FCOM ShortRealVar
3	longreal	FCOM LongRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	40-50	40-50	24	4	4/1
2	60-70+EA	60-70	26	4	4/1
3	65-75+EA	65-75	31	4	4/1

Opcodes

#	B1	B2	B3	B4
1	D8	D0+i		
3	D8	/2	a16	
4	DC	/2	a16	

Bemerkungen
i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 3	#IA 9, 12	#IS 1	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FCOM ist der allgemeine Befehl, um zwei Zahlen zu vergleichen. FCOM vergleicht den TOS mit Integers, BCDs und Reals, solange nur ein Stackregister als Operand angegeben wird (also Fall 1 in obiger Liste). Der Hintergrund dafür ist, daß alle diese Zahlen intern als 80-Bit-TEMPREAL dargestellt werden.

Darüber hinaus ist mit FCOM auch der Vergleich einer Zahl aus dem Speicher mit dem TOS möglich. Dies ist aber nur mit Realzahlen möglich, wobei FCOM automatisch die Konvertierung des Realformats in die TEMPREAL-Darstellung vornimmt. FCOM ist daher nicht auf Integer anwendbar – hierfür gibt es den Befehl FICOM, der seinerseits für die korrekte Konvertierung zuständig ist.

TEMPREAL-Variablen können ebenfalls nicht direkt verglichen werden. Sie müssen zunächst über FLD in ein Register geladen und dann mit FCOM mit einem anderen Register verglichen werden.

Beschreibung Seite 97

FCOMI *Pentium Pro*

Funktion Vergleich zweier Realzahlen und Setzen des Ergebnisses in EFlags.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									*				*		*

ACHTUNG, dieser Befehl verändert die Flags im EFlagregister, nicht etwa die Coprozessorflags, obwohl es sich um einen Coprozessorbefehl handelt!

Das Zero-Flag, Parity-Flag und Carry-Flag werden wie folgt gesetzt:

ZF	PF	CF	Bedeutung
0	0	0	ST(0) > ST(i)
0	0	1	ST(0) < ST(i)
1	0	0	ST(0) = ST(i)
1	1	1	ungeordnet.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	ST, ST(i)	FCOMI ST, ST(3)

Takte

#	8087	80287	80387	80487	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2	B3	B4	Bemerkungen
	DB	F0+i			i=Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception											
	#D	./.	#IA	9	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FCOMI arbeitet wie FCOM, setzt aber anstelle der Coprozessorflags die Prozessorflags entsprechend dem Ergebnis des Vergleichs.

Beschreibung Seite 149

FCOMIP *Pentium Pro*

Funktion Vergleich zweier Realzahlen und Setzen des Ergebnisses in EFlags mit anschließendem Bereinigen des Stacks.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									*				*		*

ACHTUNG, dieser Befehl verändert die Flags im EFlagregister, nicht etwa die Coprozessorflags, obwohl es sich um einen Coprozessorbefehl handelt.

Das Zero-Flag, Parity-Flag und Carry-Flag werden wie folgt gesetzt:

ZF	PF	CF	Bedeutung
0	0	0	ST(0) > ST(i)
0	0	1	ST(0) < ST(i)
1	0	0	ST(0) = ST(i)
1	1	1	ungeordnet.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST, ST(i)	FCOMIP ST, ST(2)

Takte

#	8087	80287	80387	80487	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2	B3	B4	Bemerkungen
	DF	F0+i			i=Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception											
	#D	./.	#IA	9	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FCOMIP arbeitet wie FCOMI, bereinigt jedoch nach dem Vergleich den Stack.

Beschreibung Seite 149

FCOMP *8087*

Funktion Vergleich des TOS-Inhalts mit einer Realzahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				(*)	(*)	*		*					*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Falls das Invalid-Operation-Flag *nicht* gesetzt ist, werden lediglich die Flags C3 und C0 verändert. Mögliche Zustände sind:

C3	C0	Bedeutung
0	0	Operand 1 > Operand 2
0	1	Operand 1 < Operand 2
1	0	Operand 1 = Operand 2
1	1	Operanden nicht vergleichbar

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
1	ST(i)	FCOMP ST(4)
2	shortreal	FCOMP ShortRealVar
3	longreal	FCOMP LongRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	42-52	45-52	26	4	4/1
2	63-73+EA	63-73	26	4	4/1
3	67-77+EA	67-77	31	4	4/1

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	D8	D8+i			i = Registernummer
3	D8	/3	a16		
4	DC	/3	a16		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 3	#IA 9, 12	#IS 1	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FCOMP arbeitet wie FCOM, nur wird nach dem Vergleich der Stack-Pointer um 1 inkrementiert, so daß ein Poppen des Stacks durchgeführt wird.

Beschreibung Seite 97

FCOMPP 8087

Funktion Vergleich des TOS-Inhalts mit dem Inhalt von ST(1).

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				(*)	(*)	*		*					*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Falls das Invalid-Operation-Flag *nicht* gesetzt ist, werden lediglich die Flags C3 und C0 verändert. Mögliche Zustände sind:

C3	C0	Bedeutung
0	0	Operand 1 > Operand 2
0	1	Operand 1 < Operand 2
1	0	Operand 1 = Operand 2
1	1	Operanden nicht vergleichbar

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	keine	FCOMPP

Takte

#	8087	80287	80387	80487	Pentium
	45-55	45-55	26	5	4/1

Opcodes

#	B1	B2
	DE	D9

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 3	#IA 9, 12	#IS 1	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FCOMPP arbeitet wie FCOM, nur wird nach dem Vergleich der Stack-Pointer um 2 inkrementiert, so daß ein doppeltes Poppen des Stacks durchgeführt wird.

Beschreibung Seite 97

FCOS *80387*

Funktion Ermittlung des Cosinus des TOS-Inhalts.

Flags	B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
		?				*	*	?		*	*				*	*
(80x86-Flags)		Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. C2 signalisiert in diesem Fall, daß eine Reduktion auf den gültigen Wertebereich nicht durchgeführt werden konnte und der Befehl nicht ausgeführt wurde. Ist neben dem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt, so zeigt C1 an, ob ein Stack-Underflow (C1 = 1) oder ein Stack-Overflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung	#	Parameter	Beispiel
		keine	FCOS

Takte	#	8087	80287	80387	80487	Pentium
		-	-	123-772 *	193-279	18-124

* wenn das Argument im Bereich $-\pi/4 < x < \pi/4$ ist. Andernfalls müssen 76 Takte für die Standardisierung hinzugerechnet werden.

Opcodes

#	B1	B2
	D9	FF

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	2	#IA	1, 5	#IS	1	#O	./.	#P	1	#U	1	#Z	./.

Beschreibung Seite 138

FDECSTP *8087*

Funktion Dekrementieren des Stack-Pointers.

Flags B C3 X X X C2 C1 C0 X S P U O Z D I

(80x86-Flags) Z P C

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FDECSTP

Takte

#	8087	80287	80387	80487	Pentium
	6-12	6-12	22	3	1

Opcodes

#	B1	B2
	D9	F6

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

Bemerkungen FDECSTP löscht keine Stackinhalte und markiert auch nicht den *Bottom-of-Stack* als *empty*! Es wird lediglich ein zyklisches Vertauschen der Registerinhalte »nach oben« durchgeführt, so daß z.B. der TOS zum *Bottom-of-Stack* wird, dieser zu ST(6) und ST(1) zum TOS!

Der umgekehrte Vorgang wird durch FINCSTP erreicht. Das Markieren eines Registers als *empty* wird durch FFREE bewerkstelligt.

Beschreibung Seite 110

FDISI *8087*
FNDISI

Funktion: Abschalten der Interrupts.

Flags:

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags): Z P C

Stack-Pointer: ± 0

Verwendung:

#	Parameter	Beispiel	Bemerkungen
	keine	FDISI	nur bis 80286!
	keine	FNDISI	nur bis 80286!

Takte:

#	8087	80287	80387	80487	Pentium
	2-8	-	-	-	-

Opcodes:

#	B1	B2
	DB	E1

Exceptions: Keine

Bemerkungen: FDISI und FNDISI haben nur beim 8087 eine Bedeutung, da nur bei diesem Coprozessor bei verschiedenen Ursachen ein Interrupt ausgelöst wird. Ab dem 80287 erfolgt dies durch Exceptions, so daß auf die Fähigkeit zur Interrupt-Auslösung verzichtet werden konnte. FDISI und FNDISI sind dennoch programmierbar, haben ab dem 80287 jedoch keine Auswirkung mehr.

FNDISI unterscheidet sich von FDISI nur dadurch, daß der Assembler bei FNDISI anstelle des vorgeschalteten WAIT-Befehls ein NOP voranstellt.

Das Einschalten der Interrupts erfolgt mit FENI.

Beschreibung: Seite 112

FDIV 8087

Funktion Division zweier Realzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*	*	*	*

(80x86-Flags) Z P C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	ST,ST(i)	FDIV ST, ST(1)	
2	ST(i), ST	FDIV ST(3), ST	
3	shortreal	FDIV ShortRealVar	Ziel: ST(0)
4	longreal	FDIV LongRealVar	Ziel: ST(0)

Takte

#	8087	80287	80387	80487	Pentium
1	193-203	193-203	88-91	73	39
2	193-203	193-203	88-91	73	39
3	215-225	215-225	89	73	39
4	220-230	220-230	94	73	39

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	D8	F0+i			i = Registernummer
2	DC	F0+i			i = Registernummer
3	D8	/6	a16		
4	DC	/6	a16		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 13	#IS 1	#O 1	#P 1	#U 1	#Z 1

Bemerkungen FDIV ist der allgemeine Befehl, um zwei Zahlen zu dividieren. FDIV dividiert zwei Registerinhalte (wovon ein Register der TOS sein muß) unabhängig davon voneinander, ob sie Realzahlen, Integerzahlen oder BCDs sind (also Fall 1 und 2 in obiger Liste). Der Hintergrund ist, daß alle diese Zahlen intern als 80-Bit-TEMPREAL dargestellt werden.

Darüber hinaus ist mit FDIV auch die Division des TOS durch eine Zahl aus dem Speicher möglich. *Dies ist aber nur mit Realzahlen möglich*, wobei FDIV automatisch die Konvertierung des Realformats in die TEMPREAL-Darstellung vornimmt. *FDIV ist daher nicht auf Integer anwendbar* – hierfür gibt es den Befehl FIDIV, der seinerseits für die korrekte Konvertierung zuständig ist!

TEMPREAL-Variablen können ebenfalls nicht direkt zur Division verwendet werden. Sie müssen zunächst über FLD in ein Register geladen und dann mit FDIV über ein Register dividiert werden.

Beschreibung Seite 95

FDIVP *8087*

Funktion Division zweier Realzahlen im Stack mit anschließendem Poppen.

Flags	B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
		?				?	*	?		*	*	*	*	*	*	*
(80x86-Flags)		Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST(i),ST	FDIVP ST(1), ST

Takte

#	8087	80287	80387	80487	Pentium
	197-207	198-209	88-91	73	39

Opcodes

#	B1	B2	Bemerkungen
	DE	F8+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 13	#IS 1	#O 1	#P 1	#U 1	#Z 1

Bemerkungen

FDIVP arbeitet wie FDIV, nur daß als Parameter nur Stack-Register zugelassen sind. Im Unterschied zu FDIV wird anschließend der Wert im TOS durch ein Poppen des Stacks gelöscht.

Da FDIVP nach der Division den Wert im TOS entfernt und ein Poppen des Stacks durchführt, macht die Konstruktion *FDIV ST,ST(i)* keinen Sinn und wird daher vom Assembler nicht unterstützt!

Beschreibung

Seite 95

FDIVR *8087*

Funktion

Reziproke Division zweier Realzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*	*	*	*

(80x86-Flags)

Z	P	C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	ST,ST(i)	FDIVR ST, ST(7)	
2	ST(i), ST	FDIVR ST(5), ST	
3	shortreal	FDIVR ShortRealVar	Ziel: ST(0)
4	longreal	FDIVR LongRealVar	Ziel: ST(0)

Takte

#	8087	80287	80387	80487	Pentium
1	194-204	194-204	88-91	73	39
2	194-204	194-204	88-91	73	39
3	216-226+EA	215-225	89	73	39
4	221-231+EA	220-230	94	73	39

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	D8	F8+i			i = Registernummer
2	DC	F8+i			i = Registernummer
3	D8	/7	a16		
4	DC	/7	a16		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 13	#IS 1	#O 1	#P 1	#U 1	#Z 2

Bemerkungen FDIVR arbeitet wie FDIV. Allerdings wird die Division reziprok durchgeführt, das heißt, daß in das Ziel (= 1. Operand) nicht wie bei FDIV das Ergebnis der Division »Ziel durch Quelle«, sondern das der Division »Quelle durch Ziel« eingetragen wird.

Beschreibung Seite 96

FDIVRP 8087

Funktion Reziproke Division zweier Realzahlen im Stack mit anschließendem Poppen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*	*	*	*

(80x86-Flags) Z P C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST(i),ST	FDIVRP ST(1), ST

Takte

#	8087	80287	80387	80487	Pentium
	198-208	198-208	88-91	73	39

Opcodes

#	B1	B2	Bemerkungen
	DE	F0+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 13	#IS 1	#O 1	#P 1	#U 1	#Z 2

Bemerkungen Auch für FDIVR gibt es wie bei dem Paar FDIV/FDIVP eine Version, die nach der Division ein Poppen des Stacks durchführt. Da FDIVRP nach der Division den Wert im TOS entfernt und ein Poppen des Stacks durchführt, macht die Konstruktion *FDIVRP ST,ST(i)* keinen Sinn und wird daher vom Assembler nicht unterstützt.

Beschreibung Seite 96

FENI *8087*
FNENI

Funktion Einschalten der Interrupts.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags)

Z		P	C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
	keine	FENI	nur bis 80286!
	keine	FNENI	nur bis 80286!

Takte

#	8087	80287	80387	80487	Pentium
	2-8	-	-	-	-

Opcodes

#	B1	B2
	DB	E0

Exception: Keine

Bemerkungen FENI und FNENI haben nur beim 8087 eine Bedeutung, da nur bei diesem Coprozessor bei verschiedenen Ursachen ein Interrupt ausgelöst wird. Ab dem 80287 erfolgt dies durch Exceptions, so daß auf die Fähigkeit zur Interrupt-Auslösung verzichtet werden konnte. FENI und FNENI sind dennoch programmierbar, haben ab dem 80287 jedoch keine Auswirkung mehr. FNENI unterscheidet sich von FENI nur dadurch, daß der Assembler bei FNENI anstelle des vorgeschalteten WAIT-Befehls ein NOP voranstellt.

Das Ausschalten der Interrupts erfolgt mit FDISI.

Beschreibung Seite 112

FFREE 8087

Funktion Register als *empty* markieren.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags)

Z			P		C

Stack pointer ± 0

Verwendung

#	Parameter	Beispiel
	ST(i)	FFREE ST(5)

Takte

#	8087	80287	80387	80487	Pentium
	9-16	9-16	18	3	1

Opcodes

#	B1	B2	Bemerkungen
	DD	C0+i	i = Registernummer

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	2	-	2	2

Bemerkungen FFREE trägt in das *Tag-Feld* des betreffenden Registers einen Code ein, der das Register als leer kennzeichnet. Ein solchermaßen gekennzeichnetes Register kann als Ziel für die Ladebefehle FLD, FILD und FBLD dienen.

Beschreibung Seite 110

FIADD 8087

Funktion Addition zweier Integerzahlen

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*		*		*	*

(80x86-Flags)

Z			P		C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FIADD WordVar
2	shortint	FIADD ShortIntlVar

Takte

#	8087	80287	80387	80487	Pentium
1	102-137+EA	102-137	71-85	19-32	7/4
2	108-143+EA	108-143	57-72	20-35	7/4

Opcodes

#	B1	B2	B3	B4
1	DE	/0	a16	
2	DA	/0	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 7	#IS 1	#O ./.	#P 1	#U 1	#Z ./.

Bemerkungen FIADD ist der fehlende Teil von FADD, der Integer direkt zu Stack-Inhalten addieren kann. FIADD ermöglicht dies, indem es automatisch eine Konvertierung des Integerformats in das interne 80-Bit-TEMPREAL-Format des Coprozessors vornimmt. Aus diesem Grunde kann auch das Ziel der Operation immer nur der TOS sein!

Für alle anderen Additionen, also die von Realzahlen oder den Inhalten von Stackregistern untereinander, ist FADD zuständig!

FIADD kann nur Integer mit 2 oder 4 Bytes direkt addieren! Falls LongInts verwendet werden sollen, so muß dieser Datentyp zunächst mit dem Ladebefehl FILD in den TOS geladen werden, wobei die Konvertierung in den 80-Bit-TEMPREAL-Modus erfolgt. Danach kann mit FADD die Addition durchgeführt werden.

Beschreibung Seite 107

FICOM 8087

Funktion Vergleich zweier Integerzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				(*)	(*)	*		*					*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Falls das Invalid-Operation-Flag *nicht* gesetzt ist, werden lediglich die Flags C3 und C0 verändert. Mögliche Zustände sind:

C3	C0	Bedeutung
0	0	Operand 1 > Operand 2
0	1	Operand 1 < Operand 2
1	0	Operand 1 = Operand 2
1	1	Operanden nicht vergleichbar

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FICOM WordVar
2	shortint	FICOM ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	72-86+EA	72-86	71-75	16-20	8/4
2	78-91+EA	78-91	56-63	15-17	8/4

Opcodes

#	B1	B2	B3	B4
1	DE	/2	a16	
2	DA	/2	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 3	#IA 9, 12	#IS 1	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FICOM ist der fehlende Teil von FCOM, der Integer direkt mit Stack-Inhalten vergleichen kann. FICOM ermöglicht dies, indem es automatisch eine Konvertierung des Integerformats in das interne 80-Bit-TEMPREAL-Format des Coprozessors vornimmt.

Für alle anderen Vergleiche, also die von Realzahlen oder den Inhalten von Stackregistern untereinander ist FCOM zuständig!

FICOM kann nur Integer mit 2 oder 4 Bytes direkt vergleichen! Falls Long-Ints verwendet werden sollen, so muß dieser Datentyp zunächst mit dem Ladebefehl FILD in den TOS geladen werden, wobei die Konvertierung in den 80-Bit-TEMPREAL-Modus erfolgt. Danach kann mit FCOM der Vergleich durchgeführt werden.

Beschreibung Seite 107

FICOMP *8087*

Funktion Vergleich des TOS-Inhalts mit einer Integerzahl und anschließendes Poppen des Stacks.

Flags	B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
		*				(*)	(*)	*		*					*	*
(80x86-Flags)		Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Falls das Invalid-Operation-Flag *nicht* gesetzt ist, werden lediglich die Flags C3 und C0 verändert. Mögliche Zustände sind:

C3	C0	Bedeutung
0	0	Operand 1 > Operand 2
0	1	Operand 1 < Operand 2
1	0	Operand 1 = Operand 2
1	1	Operanden nicht vergleichbar

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
1	word	FICOMP WordVar
2	shortint	FICOMP ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	74-88+EA	74-88	71-75	16-20	8/4
2	80-93+EA	80-93	56-63	15-17	8/4

Opcodes

#	B1	B2	B3	B4
1	DE	/3	a16	
2	DA	/3	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 3	#IA 9, 12	#IS 1	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FICOMP arbeitet wie FICOM, nur wird nach dem Vergleich der Stack-Pointer um 1 inkrementiert, so daß ein Poppen des Stacks durchgeführt wird.

Beschreibung Seite 107

FIDIV *8087*

Funktion Division zweier Integerzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*	*	*	*

(80x86-Flags)

Z			P		C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FIDIV WordVar
2	shortint	FIDIV ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	224-238+EA	224-238	136-140	73	42
2	230-243+EA	230-243	120-127	73	42

Opcodes

#	B1	B2	B3	B4
1	DE	/6	a16	
2	DA	/6	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 13	#IS 1	#O 1	#P 1	#U 1	#Z 1

Bemerkungen FIDIV ist der fehlende Teil von FDIV, der Stackregister direkt durch Integer dividieren kann. FIDIV ermöglicht dies, indem es automatisch eine Konvertierung des Integerformats in das interne 80-Bit-TEMPREAL-Format des Coprozessors vornimmt.

Für alle anderen Divisionen, also die von Realzahlen oder den Inhalten von Stackregistern untereinander ist FDIV zuständig!

FIDIV kann nur Integer mit 2 oder 4 Bytes direkt verwenden! Falls LongInts verwendet werden sollen, so muß dieser Datentyp zunächst mit dem Ladebefehl FILD in den TOS geladen werden, wobei die Konvertierung in den 80-Bit-TEMPREAL-Modus erfolgt. Danach kann mit FDIV dividiert werden.

Beschreibung Seite 107

FIDIVR 8087

Funktion Reziproke Division zweier Integer.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*	*	*	*

(80x86-Flags) Z P C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FIDIVR WordVar
2	shortint	FIDIVR ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	225-239+EA	224-238	135-141	73	42
2	231-245+EA	230-243	121-128	73	42

Opcodes

#	B1	B2	B3	B4
1	DE	/7	a16	
2	DA	/7	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 13	#IS 1	#O 1	#P 1	#U 1	#Z 2

Bemerkungen FIDIVR arbeitet wie FIDIV. Allerdings wird die Division reziprok durchgeführt, das heißt, daß in das Ziel (= 1. Operand) nicht wie bei FIDIV das Ergebnis der Division »Ziel durch Quelle«, sondern das der Division »Quelle durch Ziel« eingetragen wird. Mit anderen Worten: Bei FIDIV wird der TOS durch die Integer dividiert, bei FIDIVR die Integer durch den TOS. Beide Ergebnisse werden jedoch im TOS abgelegt!

Beschreibung Seite 107

FILD *8087*

Funktion FILD lädt eine Integer in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
1	word	FILD WordVar
2	shortint	FILD ShortIntVar
3	longint	FILD LongIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	46-54+EA	46-54	61-65	9-12	3/1
2	52-60+EA	52-60	45-52	13-16	3/1
3	60-68+EA	60-68	56-67	10-18	3/1

Opcodes

#	B1	B2	B3	B4
1	DF	/0	a16	
2	DB	/0	a16	
3	DF	/5	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception											
	#D	./.	#IA	./.	#IS	2	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FILD kann nur dann seine Funktion korrekt ausführen, wenn das letzte Register des Stacks, also ST(7) oder auch *Bottom-of-Stack*, als *empty* markiert ist. Andernfalls wird das Invalid-Exception-Flag gesetzt.

Beschreibung Seite 107

FIMUL *8087*

Funktion Multiplikation zweier Integerzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?			*	*	*		*	*

(80x86-Flags)

Z			P		C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FIMUL WordVar
2	shortint	FIMUL ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	124-138+EA	124-238	76-87	8	7/4
2	130-144+EA	130-244	61-82	8	7/4

Opcodes

#	B1	B2	B3	B4
1	DE	/1	a16	
2	DA	/1	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 8	#IS 1	#O 1	#P 1	#U 1	#Z ./.

Bemerkungen

FIMUL ist der fehlende Teil von FMUL, der Stackregister direkt mit Integer multiplizieren kann. FIMUL ermöglicht dies, indem es automatisch eine Konvertierung des Integerformats in das interne 80-Bit-TEMPREAL-Format des Coprozessors vornimmt.

Für alle anderen Multiplikationen, also die von Realzahlen oder den Inhalten von Stackregistern untereinander ist FMUL zuständig!

FIMUL kann nur Integer mit 2 oder 4 Bytes direkt verwenden! Falls LongInts verwendet werden sollen, so muß dieser Datentyp zunächst mit dem Ladebefehl FILD in den TOS geladen werden, wobei die Konvertierung in den 80-Bit-TEMPREAL-Modus erfolgt. Danach kann mit FMUL multipliziert werden.

Beschreibung Seite 107

FINCSTP *8087*

Funktion Inkrementieren des Stack-Pointers.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags)

Z		P	C

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	keine	FINCSTP

Takte

#	8087	80287	80387	80487	Pentium
	6-12	6-12	21	3	1

Opcodes

#	B1	B2
	D9	F7

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	2	-	2	2

Bemerkungen FINCSTP löscht keine Stack-Inhalte! Es wird lediglich ein zyklisches Vertauschen der Registerinhalte »nach unten« durchgeführt, so daß z.B. der *Bottom-of-Stack* zum TOS wird, dieser zu ST(1) und ST(1) zu ST(2)!

Der umgekehrte Vorgang wird durch FDECSTP erreicht.

Beschreibung Seite 110

FINIT *8087*
FNINIT

Funktion Initialisierung des Coprozessors.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	0				0	0	0								

(80x86-Flags)

Z	P	C

Der Condition Code wird explizit gelöscht.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FINIT

Takte

#	8087	80287	80387	80487	Pentium
	6-12	6-12	22	17	16/12*

* Der zweite Wert gilt für FNINIT.

Opcodes

#	B1	B2
	DB	E3

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

Bemerkungen FINIT schreibt Defaultwerte in das *Kontrollwort* (vgl. FLDCW). Die Rechenregister werden als *empty* markiert, und das *Statuswort* wird gelöscht.

FNINIT ist eine Variante von FINIT, bei der der Assembler anstelle des sonst üblichen WAIT-Befehls einen NOP-Befehl vor den eigentlichen Befehl stellt.

Beschreibung Seite 112

FIST *8087*

Funktion FIST speichert eine Integer aus dem TOS in den Speicher.

Flags	B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
		?				?	*	?		*	*					*
(80x86-Flags)		Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FIST WordVar
2	shortint	FIST ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	80-90+EA	80-90	82-95	29-34	6
2	82-92+EA	82-92	79-93	28-34	6

Opcodes

#	B1	B2	B3	B4
1	DF	/2	a16	
2	DB	/2	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception									
	#D	./.	#IA	1, 4	#IS	1	#O	./.	#P	1	#U	./.	#Z	./.

Bemerkungen

FIST speichert nur die Zahl im TOS und konvertiert sie dabei in das WORDINT- oder SHORTINT-Format. LONGINT-Daten können mit FIST nicht geschrieben werden. Hierzu muß der Befehl FISTP verwendet werden.

FIST konvertiert nur in Integer! Falls die Zahl im TOS in anderen Formaten gespeichert werden soll, so ist FST für Reals und FBSTP für BCDs zu verwenden.

Beschreibung

Seite 107

FISTP *8087*

Funktion

FISTP speichert eine Integer aus dem TOS in den Speicher und führt anschließend ein Poppen des Stacks aus.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*					*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer

+ 1

Verwendung

#	Parameter	Beispiel
1	word	FISTP WordVar
2	shortint	FISTP ShortIntVar
3	longint	FISTP LongIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	82-92+EA	82-92	82-95	29-34	6
2	84-94+EA	84-94	79-93	29-34	6
3	94-105+EA	94-105	80-97	29-34	6

Opcodes

#	B1	B2	B3 B4
1	DF	/3	a16
2	DB	/3	a16
3	DF	/7	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception							
	#D ./.	#IA 1, 4	#IS 1	#O ./.	#P 1	#U ./.	#Z ./.	

Bemerkungen FISTP arbeitet wie FIST, nimmt aber den gespeicherten Wert anschließend vom Stack. FISTP ist auch die einzige Möglichkeit, LONGINT-Formate zu schreiben.

Beschreibung Seite 107

FISUB *8087*

Funktion Subtraktion zweier Integerzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags) Z P C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FISUB WordVar
2	shortint	FISUB ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	102-137+EA	102-127	71-83	19-32	7/4
2	108-143+EA	108-143	57-82	20-35	7/4

Opcodes

#	B1	B2	B3	B4
1	DE	/4	a16	
2	DA	/4	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 6	#IS 1	#O 1	#P 1	#U 1	#Z ./.

Bemerkungen FISUB ist der fehlende Teil von FSUB, der direkt Integer vom Stackregister subtrahieren kann. FISUB ermöglicht dies, indem es automatisch eine Konvertierung des Integerformats in das interne 80-Bit-TEMPREAL-Format des Coprozessors vornimmt.

Für alle anderen Subtraktionen, also die von Realzahlen oder den Inhalten von Stackregistern untereinander, ist FSUB zuständig!

FISUB kann nur Integer mit 2 oder 4 Bytes direkt verwenden! Falls LongInts verwendet werden sollen, so muß dieser Datentyp zunächst mit dem Ladebefehl FILD in den TOS geladen werden, wobei die Konvertierung in den 80-Bit-TEMPREAL-Modus erfolgt. Danach kann mit FSUB dividiert werden.

Beschreibung Seite 107

FISUBR *8087*

Funktion Reziproke Subtraktion zweier Integerzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	word	FISUBR WordVar
2	shortint	FISUBR ShortIntVar

Takte

#	8087	80287	80387	80487	Pentium
1	102-137+EA	102-127	71-83	19-32	7/4
2	108-143+EA	108-143	57-82	20-35	7/4

Opcodes

#	B1	B2	B3	B4
1	DE	/5	a16	
2	DA	/5	a16	

Bemerkungen FISUBR arbeitet wie FISUB. Allerdings wird die Subtraktion reziprok durchgeführt, das heißt, daß in das Ziel (= 1. Operand) nicht wie bei FISUB das Ergebnis der Subtraktion »Ziel von Quelle«, sondern das der Subtraktion »Quelle von Ziel« eingetragen wird. Mit anderen Worten: Bei FISUB wird die Integer vom TOS abgezogen, bei FISUBR der TOS von der Integer. Beide Ergebnisse werden jedoch im TOS abgelegt!

Beschreibung Seite 107

FLD 8087

Funktion FLD lädt eine Realzahl in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*					*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
1	ST(i)	FLD ST(4)
2	shortreal	FLD ShortRealVar
3	longreal	FLD LongIRealVar
4	tempreal	FLD TempRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	17-22	17-22	14	4	1
2	38-56+EA	38-56	20	3	1
3	40-60+EA	40-60	25	3	1
4	53-65+EA	53-65	44	6	3

Opcodes

#	B1	B2	B3	B4
1	D9	C0+i		
2	D9	/0	a16	
3	DD	/0	a16	
4	DB	/5	a16	

Bemerkungen
i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception															
	#D	1	#IA	1	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.		

Bemerkungen FLD kann nur dann seine Funktion korrekt ausführen, wenn das letzte Register des Stacks, also ST(7) oder auch *Bottom-of-Stack*, als *empty* markiert ist. Andernfalls wird das Invalid-Exception-Flag gesetzt.

FLD ist der allgemeine Ladebefehl. Mit ihm können sowohl Registerinhalte in den TOS geladen werden als auch Realzahlen aus dem Speicher. Für BCDs ist der Befehl FBLD zuständig, für Integer FILD.

Beschreibung Seite 91

FLDCW *8087*

Funktion FLDCW lädt einen Wert für das *Kontrollwort* aus dem Speicher in das *Kontrollwortregister*.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags)

Z	P	C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	word	FLDCW WordVar

Takte

#	8087	80287	80387	80487	Pentium
	7-14+EA	7-14	19	4	7

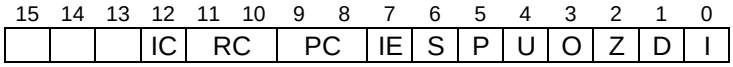
Opcodes

#	B1	B2	B3	B4
	D9	/5	a16	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen Das Wort, das dem Befehl FLDCW übergeben wird, muß folgenden Aufbau besitzen:



Die Abkürzungen haben hierbei folgende Bedeutungen:

IC *Infinity Contro*l. Ist dieses Bit gesetzt, so wird das affine Modell gewählt, in dem ein Unterschied zwischen -∞ und +∞ gemacht wird. Das bedeutet, daß bei positiven und negativen Unendlichkeiten unterschiedliche NaNs erzeugt werden.

Bei gelöschtem Bit wird das projektive Modell eingestellt, in dem diese Unterscheidung nicht stattfindet, also ggf. die gleiche NaN erzeugt wird.

RC *Round Control*. Hiermit kann eingestellt werden, wie der Prozessor Rundungen durchführen soll:

00 Rundung zur nächsten oder geraden Zahl
01 Rundung in Richtung -∞
10 Rundung in Richtung +∞
11 Abschneiden des Restes (»*Truncation*«)

PC *Precision Control*. Mit diesen beiden Bits kann die Genaukeit eingestellt werden, mit der intern die Berechnungen erfolgen. Dies betrifft nur die interne Darstellung der Mantisse. Mögliche Bitkombinationen sind:

00 24 Bits Genauigkeit (ShortReal)
01 reserviert
10 53 Bits Genauigkeit (LongReal)
11 64 Bits Genauigkeit (TempReal)

IE *Interrupt Enable*. Hiermit wird die Interrupt-Fähigkeit des Coprozessors ein- oder ausgeschaltet. Ist dieses Bit gelöscht, so erzeugt der Coprozessor einen Interrupt $00, wenn bestimmte Ausnahmezustände vorliegen. Welche Zustände einen Interrupt auslösen sollen, kann mit den folgenden Bits eingestellt werden. Beachten Sie bitte, daß ab dem 80386/80387 IE keinerlei Bedeutung und Verwendung mehr hat!

S Stack-Fault. Ist dieses Bit gelöscht, so wird Bit 6 im *Statuswort* gesetzt, falls ein Über- oder Unterlauf des Stacks stattfindet. Beachten Sie bitte, daß dieses Bit erst ab dem 80387 definiert ist!

P *Precision exception*. Ist dieses Bit gelöscht, so wird beim 8087/80287 ein INT $00 ausgelöst, falls IE ebenfalls gelöscht ist und Rundungsfehler auftreten. Ab dem 80387 wird lediglich das korrespondierende Statusbit im *Statuswort* gesetzt.

U *Underflow exception*. Bei gelöschtem U- und IE-Bit wird ein INT $00 ausgelöst (8087/80287) oder das Statusbit gesetzt (ab 80387), falls der mögliche Darstellungsbereich einer Zahl nach einer Operation unterschritten wird.

O *Overflow exception*. Wie *Underflow*, nur muß der darstellbare Bereich überschritten werden.

Z *Zero division*. Löst im gelöschten Zustand in Verbindung mit dem gelöschtem IE einen INT $00 oder das Setzen des dazugehörigen Statusbits aus, falls versucht wird, durch 0 zu dividieren.

D *Denormalized operand exception*. Wird dazu benutzt, zusammen mit IE einen Hinweis auf die Verwendung nicht normalisierter Zahlen (NaNs) zu erzeugen.

I *Invalid operation exception*. Bei gelöschtem I-Bit wird ein INT $00 ausgelöst oder das Statusbit gesetzt, falls die Operation nicht durchgeführt werden konnte, z.B. weil ein leeres Register einbezogen werden sollte. Auch hier entscheidet IE grundsätzlich, ob der Interrupt dann auch tatsächlich ausgelöst wird!

Beachten Sie bitte, daß IE, S, P, U, O, Z, D und I sogenannte »Maskenbits« sind. Dies heißt, daß die entsprechende Funktion »maskiert« wird, falls die Bits gesetzt sind. Maskiert bedeutet, daß die Aktion *nicht* ausgelöst werden kann, da der Coprozessor das Bit nicht »erkennt«. Das gesetze Bit bedeutet also das Unterbleiben der Aktion, ein gelöschtes Bit bewirkt die Ausführung!

Das *Kontrollwort* wird auch durch FINIT verändert. FINIT trägt als *Kontrollwort* den Wert $037F in das *Kontrollwortregister* ein. Nach dem obigen Schema heißt das, daß

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1
			IC	RC		PC		IE	S	P	U	O	Z	D	I

das projektive Modell mit der Gleichbehandlung der Unendlichkeiten gewählt wird, zur nächsten oder geraden Zahl gerundet wird und die Genauigkeit der Mantissendarstellung 64 Bits beträgt. Die Interruptfähigkeit ist eingeschaltet (Maskenbit = 0!), jedoch löst keine der Exceptions tatsächlich einen Interrupt aus (Maskenbits = 1!).

Beschreibung Seite 113

FLDENV 8087

Funktion FLDENV lädt eine Coprozessorumgebung. Dies sind alle coprozessorspezifischen Register außer den Rechenregistern.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				*	*	*								

(80x86-Flags)

	Z				P		C								

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	14-Byte-Var	FLDENV Array14	16-Bit-Umgebungen
1b	28-Byte-Var	FLDENV Array28	32-Bit-Umgebungen

Takte

#	8087	80287	80387	80487	Pentium
	35-45+EA	35-45	71	44	37

Opcodes

#	B1	B2	B3	B4
	D9	/4	a16	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen FLDENV entnimmt der übergebenen Struktur 14 bzw. 28 Bytes an Informationen und trägt diese in die entsprechenden Register ein. Zu Einzelheiten über die Operandengröße und -struktur siehe FSTENV.

Bei der Benutzung von FLDENV muß darauf geachtet werden, daß dieser Befehl in der gleichen Umgebung ausgeführt wird wie ein FSTENV/FNSTENV-Befehl, mit dem die Coprozessorumgebung gesichert wurde. Da sich in 16- bzw. 32-Bit-Umgebungen sowie zwischen Protected-Mode und Real-Mode Unterschiede in der Struktur und Größe der Operanden ergeben, ist ansonsten mit Problemen zu rechnen, da verschiedene Register mit nicht korrekten Daten geladen werden könnten – siehe FSTENV.

Wenn im Datenbereich des Operanden, an dem das Statuswort abgelegt worden war, Bits gesetzt sind, die eine anhängige Exception codieren, so führt das Laden der Coprozessorumgebung mittels FLDENV beim Ausfüh-

ren des nächsten FPU-Befehls zu der/den anhängigen Exceptions. Um dies zu vermeiden, sollten vor dem Laden der Umgebung mittels FLDENV die entsprechenden Bits im zuständigen Feld des Operanden für das Statuswort gelöscht werden.

Beschreibung Seite 113

FLDLG2 *8087*

Funktion FLDLG2 lädt den dekadischen Logarithmus von 2, also den Logarithmus von 2 zur Basis 10, in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FLDLG2

Takte

#	8087	80287	80387	80487	Pentium
	18-24	18-24	41	8	5/3

Opcodes

#	B1	B2
	D9	EC

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-		-		

#MF	Grund: FPU-Exception						
	#D ./.	#IA ./.	#IS 2	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FLDLG2 arbeitet wie jeder Ladebefehl, nur wird eine Konstante als Quelle verwendet.

Beschreibung Seite 100

FLDLN2 8087

Funktion FLDLN2 lädt den natürlichen Logarithmus von 2, also den Logarithmus von 2 zur Basis e, in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel	Bemerkungen
	keine	FLDLN2	

Takte

#	8087	80287	80387	80487	Pentium
	17-23	17-23	41	8	5/3

Opcodes

#	B1	B2
	D9	ED

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-		-		

#MF	Grund: FPU-Exception						
	#D ./.	#IA ./.	#IS 2	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FLDLN2 arbeitet wie jeder Ladebefehl, nur wird eine Konstante als Quelle verwendet.

Beschreibung Seite 100

FLDL2E *8087*

Funktion FLDL2E lädt den Logarithmus Dualis von e, also den Logarithmus von e zur Basis 2, in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FLDL2E

Takte

#	8087	80287	80387	80487	Pentium
	15-21	15-21	40	8	5/3

Opcodes

#	B1	B2
	D9	EA

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-		-		

#MF	Grund: FPU-Exception						
	#D ./.	#IA ./.	#IS 2	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FLDL2E arbeitet wie jeder Ladebefehl, nur wird eine Konstante als Quelle verwendet.

Beschreibung Seite 100

FLDL2T 8087

Funktion FLDL2T lädt den Logarithmus Dualis von 10, also den Logarithmus von 10 zur Basis 2, in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FLDL2T

Takte

#	8087	80287	80387	80487	Pentium
	16-22	16-22	40	8	5/3

Opcodes

#	B1	B2
	D9	E9

Bemerkungen FLDL2T arbeitet wie jeder Ladebefehl, nur wird eine Konstante als Quelle verwendet.

Beschreibung Seite 100

FLDPI 8087

Funktion FLDPI lädt die Konstante π in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FLDPI

Takte

#	8087	80287	80387	80487	Pentium
	16-22	16-22	40	8	5/3

Opcodes

#	B1	B2
	D9	EB

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-		-		

#MF	Grund: FPU-Exception						
	#D ./.	#IA ./.	#IS 2	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FLDPI arbeitet wie jeder Ladebefehl, nur wird eine Konstante als Quelle verwendet.

Beschreibung Seite 100

FLDZ *8087*

Funktion FLDZ lädt die Konstante 0.0 in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

Z			P		C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FLDZ

Takte

#	8087	80287	80387	80487	Pentium
	11-17	11-17	20	4	2/2

Opcodes

#	B1	B2
	D9	EE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-		-		

#MF	Grund: FPU-Exception						
	#D ./.	#IA ./.	#IS 2	#O ./.	#P ./.	#U ./.	#Z ./.

Bemerkungen FLDZ arbeitet wie jeder Ladebefehl, nur wird eine Konstante als Quelle verwendet.

Beschreibung Seite 100

FLD1 *8087*

Funktion FLD1 lädt die Konstante 1.0 in den TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

Z			P		C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FLD1

Takte

#	8087	80287	80387	80487	Pentium
	15-21	15-21	24	4	2/2

Opcodes

#	B1	B2
	D9	E8

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-		-		

#MF	Grund: FPU-Exception											
	#D	./.	#IA	./.	#IS	2	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FLD1 arbeitet wie jeder Ladebefehl, nur wird eine Konstante als Quelle verwendet.

Beschreibung Seite 100

FMUL *8087*

Funktion Multiplikation zweier Realzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?			*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	ST,ST(i)	FMUL ST, ST(3)	s.u.
2	ST(i), ST	FMUL ST(7), ST	s.u.
3	shortreal	FMUL ShortRealVar	Ziel: ST(0)
4	longreal	FMUL LongRealVar	Ziel: ST(0), s.u.

Takte

#	8087	80287	80387	80487	Pentium
1	130-145	90-145	29-57	16	3/1
2	130-145	90-145	29-57	16	3/1
3	110-125+EA	110-125	32-57	11	3/1
4	154-168+EA	112-168	32-57	14	3/1

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	D8	C8+i			i = Registernummer
2	DC	C8+i			i = Registernummer
3	D8	/1	a16		
4	DC	/1	a16		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception											
	#D	2	#IA	1, 8	#IS	1	#O	1	#P	1	#U	1	#Z	./.

Bemerkungen

FMUL ist der allgemeine Befehl, um zwei Zahlen zu multiplizieren. FMUL multipliziert zwei Registerinhalte (wovon ein Register der TOS sein muß) unabhängig davon miteinander, ob sie Realzahlen, Integerzahlen oder BCDs sind (also Fall 1 und 2 in obiger Liste!). Der Hintergrund dafür ist, daß alle diese Zahlen intern als 80-Bit-TEMPREAL dargestellt werden!

Darüber hinaus ist mit FMUL auch die Multiplikation des TOS mit einer Zahl aus dem Speicher möglich. Dies ist aber nur mit Realzahlen möglich, wobei FMUL automatisch die Konvertierung des Realformats in die TEMPREAL-Darstellung vornimmt. FMUL ist daher nicht auf Integerzahlen anwendbar – hierfür gibt es den Befehl FIMUL, der seinerseits für die korrekte Konvertierung zuständig ist!

TEMPREAL-Variablen können ebenfalls nicht direkt multipliziert werden. Sie müssen zunächst über FLD in ein Register geladen und dann mit FMUL über ein Register multipliziert werden.

ACHTUNG: Beim 8087 wird die Bearbeitungsgeschwindigkeit höher, wenn einer oder beide Operanden lediglich 24 signifikante Stellen der Mantisse mit 40 nachfolgenden 0-Bits hat, z.B. wenn über FLD eine *shortint* geladen wurde. In diesem Fall liegt eine SHORTINT-Genauigkeit vor, die unabhängig von der Genauigkeitsvorgabe im *Kontrollwort* mit 24-Bit-Genauigkeit bearbeitet wird. Die Ausführungsgeschwindigkeit der Register-mit-Register-Multiplikation reduziert sich in diesem Fall auf 90-105 Takte. Register-mit-LONGINT-Multiplikationen werden dann in 112-126+EA Takten ausgeführt.

Beschreibung

Seite 95

FMULP *8087*

Funktion Multiplikation zweier Realzahlen im Stack mit anschließendem Poppen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?			*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel	Bemerkungen
	ST(i), ST	FMULP ST(4), ST	s.u

Takte

#	8087	80287	80387	80487	Pentium
	134-148	198-208	29-57	16	3/1

Opcodes

#	B1	B2	Bemerkungen
	DE	C8+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 8	#IS 1	#O 1	#P 1	#U 1	#Z ./.

Bemerkungen FMULP arbeitet wie FMUL, nur daß als Parameter nur Stackregister zugelassen sind. Im Unterschied zu FMUL wird anschließend der Wert im TOS durch ein Poppen des Stacks gelöscht.

ACHTUNG: Beim 8087 wird die Bearbeitungsgeschwindigkeit höher, wenn einer oder beide Operanden lediglich 24 signifikante Stellen der Mantisse mit 40 nachfolgenden 0-Bits haben z.B. wenn über FLD eine *shortint* geladen wurde. In diesem Fall liegt eine SHORTINT-Genauigkeit vor, die unabhängig von der Genauigkeitsvorgabe im Kontrollwort mit 24-Bit-Genauigkeit bearbeitet wird. Die Multiplikationen werden dann in 94-108 Takten ausgeführt.

Beschreibung Seite 95

FNOP 8087

Funktion Nulloperation.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags) Z P C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FNOP

Takte

#	8087	80287	80387	80487	Pentium
	10-16	10-16	12	3	1

Opcodes

#	B1	B2
	D9	D0

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	2	-	2	2

Bemerkungen FNOP ist das Coprozessor-Pendant zum Prozessorbefehl NOP.

Beschreibung Seite 114

FPATAN 8087

Funktion Bildung des Arcus *Tangens* einer Zahl, die als Quotient vorliegt.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
		?			?	*	?		*	*	*			*	(*)

(80x86-Flags) Z P C

Der Coprozessor setzt ab dem 80387 das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat. Bis zum 80387 werden die Operanden nicht überprüft, weshalb bei diesen Coprozessoren auch *nicht* das Invalid-Operation-Flag verändert wird!

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	keine	FPATAN

Takte

#	8087	80287	80387	80487	Pentium
	250-800	250-800	314-487	2-17	17-173

Opcodes

#	B1	B2
	D9	F3

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception						
	#D 2	#IA 1	#IS 1	#O ./.	#P 1	#U 1	#Z ./.

Bemerkungen FPATAN bildet den Arcus Tangens einer Zahl. Diese Zahl muß jedoch als Quotient vorliegen, wobei der Divisor im TOS und der Dividend in ST(1) stehen müssen (diese Konstruktion wird z.B. von FPTAN geliefert!). Der berechnete Wert wird in ST(1) geschrieben, und ein Poppen des Stacks wird durchgeführt, so daß der Arcus Tangens in den TOS kommt.

Das Ergebnis wird im Bogenmaß dargestellt. Der gültige Wertebereich für den Übergabeparameter ist $0 \le ST(0) < St(1) < + \infty$

Beschreibung Seite 100

FPREM *8087*

Funktion Bildung des Restes einer Ganzzahldivision.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				*	*	*		*		*			*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat. Andernfalls signalisiert C2, daß die Reduktion vollständig (C2 = 0) bzw. nicht vollständig (C2 = 1) verlaufen ist.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FPREM

Takte

#	8087	80287	80387	80487	Pentium
	15-190	15-190	74-155	2-8	16-64

Opcodes

#	B1	B2
	D9	F8

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	2	#IA	11	#IS	1	#O	./.	#P	./.	#U	1	#Z	./.

Bemerkungen FPREM bildet den Rest einer Modulodivision, also einer Division mit Restbildung. Diese Restbildung erfolgt durch eine wiederholte Subtraktion des Divisors vom Dividenden, bis ein Rest übrig bleibt, der kleiner als der Divisor ist. FPREM führt also keine Division im eigentlichen Sinne durch. Das Resultat ist aber sehr viel genauer als die sonst üblichen Verfahren zur Bestimmung des Restes einer Integerdivision: *Es ist exakt*! Das Vorzeichen des Restes entspricht dem des verwendeten Dividenden.

Zur Benutzung von FPREM muß der Dividend im TOS stehen, der Divisor in ST(1). FPREM subtrahiert nun analog FSUB ST, ST(1) so lange den Divisor, bis eine der beiden folgenden Bedingungen erfüllt ist. Entweder konnte die Restbildung korrekt abgeschlossen werden, dann findet sich im TOS der Divisionsrest, und C2 des Condition Codes wird gelöscht.

Oder die Anzahl der Subtraktionen überschreitet eine bestimmte Anzahl (2^{64}). Dann wird die Restbildung abgebrochen und C2 im Condition Code gesetzt. Der Wert, der nun im TOS steht, ist der Rest, der bis zu diesem Zeitpunkt durch die Subtraktionen gebildet wurde. Er ist immer größer als der Divisor, so daß durch einen erneuten Aufruf von FPREM korrekt weitergearbeitet werden kann.

In den Flags C0, C3 und C1 sind (in dieser Reihenfolge!) die drei niederwertigen Bits des Quotienten codiert. Eine häufige Anwendung des FPREM-Befehls ist das Überführen von Argumenten in den für die Winkelfunktionen gültigen Bereich. Da dieser bei FPTAN z.B. als $0 \le X < \pi/4$ definiert ist, läßt sich mittels FPREM und dem Wert $\pi/4$ in ST(1) (fast) jedes Argument auf diesen Bereich abbilden.

Beschreibung Seite 102

FPREM1 *80387*

Funktion Bildung des Restes einer Ganzzahldivision nach IEEE.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				*	*	*		*		*			*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat. Andernfalls signalisiert C2, daß die Reduktion vollständig (C2 = 0) bzw. nicht vollständig (C2 = 1) verlaufen ist.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FPREM1

Takte

#	8087	80287	80387	80487	Pentium
	-	-	95-185	72-167	20-70

Opcodes

#	B1	B2
	D9	F5

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception							
	#D 2	#IA 11	#IS 1	#O ./.	#P ./.	#U 1	#Z ./.	

Bemerkungen Der FPREM1-Befehl ersetzt ab dem 80387 den FPREM-Befehl. Dieser ist dann zwar auch noch verfügbar, allerdings lediglich aus Kompatibilitätsgründen. Er sollte immer dann, wenn das Programm *nicht* auf Rechnern bis zum 80287 laufen *muß*, nicht mehr verwendet werden.

Der Unterschied zwischen beiden Befehlen liegt lediglich in der Art, wie der Quotient gerundet wird. FPREM hält sich hierbei an keinen Standard, FPREM1 folgt dem Standard nach IEEE-754.

Beschreibung Seite 139

FPTAN 8087

Funktion Bildung des Tangens einer Zahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
					*	*			(*)	*	*				(*)

(80x86-Flags)

Z			P		C

Der Coprozessor setzt ab dem 80386 das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. C2 signalisiert in diesem Fall, daß eine Reduktion auf den gültigen Wertebereich nicht durchgeführt werden konnte und der Befehl nicht ausgeführt wurde! Ist neben dem Invalid-Operation-Flag auch das Stack-Overflow-Flag gesetzt, so zeigt C1 an, ob ein Stack-Underflow (C1 = 1) oder ein Stack-Overflow (C1 = 0) stattgefunden hat. Vor dem 80386 wird das Invalid-Operation- sowie das Stack-Overflow-Flag *nicht* verändert, da hier eine Überprüfung der Operanden unterbleibt.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FPTAN

Takte

#	8087	80287	80387	80487	Pentium
	30-540	30-504	191-573	200-273	17-173

Opcodes

#	B1	B2
	D9	F2

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	2	#IA	1, 5	#IS	1	#O	./.	#P	1	#U	1	#Z	./.

Bemerkungen FPTAN bildet den Tangens einer Zahl im TOS. Diese Zahl muß im Bogenmaß im Bereich $0 \le X < \pi/4$ vorliegen.

FPTAN bildet den sogenannten *partiellen* Tangens. Dazu erfolgt ein Pushen des Stacks. Anschließend wird in ST(1) der partielle Tangens eingetragen und im TOS ein Divisor, durch den der Wert in ST(1) dividiert werden muß, um den eigentlichen Tangens zu erhalten!

Gültige Werte für den Übergabeparameter liegen im Bereich $0 \le ST(0) < \pi/4$.

Beschreibung Seite 99

FRNDINT *8087*

Funktion Runden auf die nächste ganze Zahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*				*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FRNDINT

Takte

#	8087	80287	80387	80487	Pentium
	16-50	16-50	66-80	21-30	9-20

Opcodes

#	B1	B2
	D9	FC

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception													
	#D	2	#IA	1	#IS	1	#O	./.	#P	1	#U	./.	#Z	./.

Bemerkungen FRNDINT rundet gemäß der in den *Round-Control-Bits* im Kontrollwort gesetzten Rundungsart den Wert in TOS auf die nächste Integer.

Beschreibung Seite 102

FRSTOR 8087

Funktion Wiederherstellen der Coprozessorregister.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				*	*	*								

(80x86-Flags) Z P C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	94-Byte-Var	FRSTOR Array94	16-Bit-Umgebungen
1b	108-Byte-Var	FRSTOR Array108	32-Bit-Umgebungen

Takte

#	8087	80287	80387	80487	Pentium
	197-207+EA	205-215	308	131	75/95

Opcodes

#	B1	B2	B3	B4
	DD	/4	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen FRSTOR liest aus der als Parameter übergebenen 94/108-Byte-Struktur die Inhalte der Coprozessorregister und die Coprozessorumgebung, die durch FSAVE dorthin gesichert wurden.

ACHTUNG: Beim 80287 kann dieser Befehl nicht dazu benutzt werden, die Ausführungszeit zu messen. Der Grund dafür ist, daß die Ausführung des Befehls in etwa genauso lange dauert wie die Übertragung der Werte in den Speicher, weshalb dieser Vorgang die Ausführungszeit insgesamt bestimmt, nicht aber die Ausführung des Coprozessorbefehls.

FRSTOR arbeitet analog zu FLDENV, restauriert jedoch auch die Inhalte der Rechenregister, weshalb zu den 14/28 Bytes, die FLDENV benutzt, noch einmal 80 (= 8 Register · 10 Bytes/Register) Bytes für diese hinzukommen.

Die Größe und Struktur des Operanden ist von der Umgebung abhängig, in der der Befehl verwendet wird. In 16-Bit-Umgebungen beträgt die Größe 94 Bytes, in 32-Bit-Umgebungen 108 Bytes. Zu Einzelheiten über den Aufbau des Operanden siehe FSTENV.

Bei der Benutzung von FRSTOR muß darauf geachtet werden, daß dieser Befehl in der gleichen Umgebung ausgeführt wird wie ein FSAVE/FNSAVE-Befehl, mit dem die Coprozessorumgebung gesichert wurde. Da sich in 16- bzw. 32-Bit-Umgebungen sowie zwischen Protected-Mode und Real-Mode Unterschiede in der Struktur und Größe der Operanden ergeben, ist ansonsten mit Problemen zu rechnen, da verschiedene Register mit nicht korrekten Daten geladen werden könnten – siehe FSTENV.

Wenn im Datenbereich des Operanden, an dem das Statuswort abgelegt worden war, Bits gesetzt sind, die eine anhängige Exception codieren, so führt das Laden der Coprozessorumgebung mittels FRSTOR beim Ausführen des nächsten FPU-Befehls zu der/den anhängigen Exceptions. Um dies zu vermeiden, sollten vor dem Laden der Umgebung mittels FRSTOR die entsprechenden Bits im für das Statuswort zuständigen Feld des Operanden gelöscht werden.

Beschreibung Seite 114

FSAVE / *FNSAVE* — *8087*

Funktion Sichern der Coprozessorregister und anschließende Initialisierung des Coprozessors.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	0				0	0	0								
(80x86-Flags)		Z				P			C						

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	94-Byte-Var	FSAVE Array94	16-Bit-Umgebungen
1b	108-Byte-Var	FNSAVE Array108	32-Bit-Umgebungen

Takte

#	8087	80287	80387	80487	Pentium
	197-207+EA	205-215	375-376	154	172/151*

* zuzüglich mindestens 3 Takte für FWAIT bei FSAVE!

Opcodes

#	B1	B2	B3	B4
	DD	/6	a16	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen

FSAVE sichert neben der Coprozessorumgebung (vgl. FSTENV) auch die Inhalte der Coprozessorregister, z.B. wenn innerhalb einer Interrupt-Routine mit dem Coprozessor gearbeitet werden soll. Dann kann bei Eintritt in diese Routine der gesamte Coprozessorinhalt mit FSAVE gesichert, mit dem Coprozessor gearbeitet und anschließend vor dem Rücksprung in die unterbrochene Routine durch FRSTOR der ursprüngliche Inhalt wiederhergestellt werden. Auf diese Weise verhindert man Konflikte, wenn sowohl die unterbrochene als auch die unterbrechende Routine auf den Coprozessor zurückgreifen (sollen).

ACHTUNG: Beim 80287 kann dieser Befehl nicht dazu benutzt werden, die Ausführungszeit zu messen. Der Grund dafür ist, daß die Ausführung des Befehls in etwa genauso lange dauert wie die Übertragung der Werte in den Speicher, weshalb dieser Vorgang die Ausführungszeit insgesamt bestimmt, nicht aber die Ausführung des Coprozessorbefehls.

FNSAVE ist eine Variante von FSAVE, bei der der Assembler anstelle des sonst üblichen WAIT-Befehls einen NOP-Befehl vor den eigentlichen Befehl stellt.

Die Größe und Struktur des Operanden ist von der Umgebung abhängig, in der der Befehl verwendet wird. In 16-Bit-Umgebungen beträgt die Größe 94 Bytes, in 32-Bit-Umgebungen 108 Bytes. Zu Einzelheiten über den Aufbau des Operanden siehe FSTENV. Zu der dort angegebenen Operandengröße kommen bei FSAVE/FNSAVE noch 80 Bytes (= 8 Register · 10 Bytes/Register) für die Inhalte der FPU-Register hinzu.

Die Inhalte dieser acht Rechenregister werden ab dem Offset 14 bzw. 28 der Struktur pro Register wie folgt abgelegt: Zunächst das Wort mit den Bits 15 bis 0 der Mantisse, danach die Bits 31 bis 16, dann die Bits 47 bis 32 und schließlich die Bits 63 bis 48 der Mantisse. Das fünfte Wort enthält das Vorzeichen an Bitposition 15 sowie den Exponenten.

FSAVE prüft auf anhängige Exceptions und behandelt diese, bevor die Sicherung der Umgebung und der Register erfolgt. FNSAVE tut dies nicht.

ACHTUNG: FSAVE/FNSAVE initialisiert den Coprozessor nach der Sicherung der Inhalte der Register und der Umgebung, so wie es FINIT/FNINIT tun.

Beschreibung

Seite 114

FSCALE 8087

Funktion Erzeugen einer Zahl aus Mantisse und Exponent.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FSCALE

Takte

#	8087	80287	80387	80487	Pentium
	32-38	32-38	67-86	30-32	20-31

Opcodes

#	B1	B2
	D9	FD

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception													
	#D	2	#IA	1	#IS	1	#O	1	#P	1	#U	1	#Z	./.

Bemerkungen FSCALE ist die Umkehrfunktion zu FXTRACT. FSACLE erwartet zwei Werte: die Mantisse der zu erzeugenden Zahl im TOS und deren Exponent in ST(1).

FSCALE führt dann mit diesen Werten eine Operation der Art TOS = TOS· $2^{ST(1)}$ durch. Das Ergebnis der Operation überschreibt also den TOS. Ein Poppen des Stacks findet *nicht* statt!

ACHTUNG: FSCALE erwartet die Mantisse und den Exponenten in binärer Zahlendarstellung! So muß, falls die Zahl 32 über FSCALE aus Mantisse und Exponenten erzeugt werden soll, im TOS 1.000 stehen und in ST(1) 5.0. Die (für uns realistischere) Darstellung von 3.2 als Mantisse und 1 als Exponenten des Dezimalsystems unterstützt FSACLE *nicht*!

ACHTUNG: als Exponent kann FSACLE nur ganze Zahlen verarbeiten. Steht in ST(1) also eine Realzahl, so wird nur ihr Vorkommateil verwendet!

Gültiger Wertebereich für FSCALE ist $-2^{15} \leq ST(1) < 2^{15}$.

Beschreibung Seite 101

FSETPM *80287*

Funktion Umschalten in den Protected-Mode des Coprozessors.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags)

Z	P	C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
	keine	FSETPM	nur 80287

Takte

#	8087	80287	80387	80487	Pentium
	-	2-8	0	0	0

Opcodes

#	B1	B2
	DB	E4

Exceptions Keine

Bemerkungen FSETPM dient dazu, den Coprozessor ebenfalls in den Protected-Mode zu schalten, nachdem der Prozessor selbst in diesem Modus betrieben wird.

Erst der 80287 kennt den Protected-Mode, so daß auch erst ab diesem Prozessor der Befehl FSETPM verfügbar ist – und auch nur bei diesem, da ab dem 80387 kein Unterschied zwischen der Adressierung in den verschiedenen Betriebsmodi des Prozessors gemacht wird. FSETPM hat daher bei diesen Coprozessoren keine Wirkung.

Beschreibung Seite 126

FSIN 80387

Funktion Bildung des Sinus einer Zahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				*	*	?		*	*	*			*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. C2 signalisiert in diesem Fall, daß eine Reduktion auf den gültigen Wertebereich nicht durchgeführt werden konnte und der Befehl nicht ausgeführt wurde! Ist neben dem Invalid-Operation-Flag auch das Stack-Overflow-Flag gesetzt, so zeigt C1 an, ob ein Stack-Underflow (C1 = 1) oder ein Stack-Overflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FSIN

Takte

#	8087	80287	80387	80487	Pentium
	-	-	122-771 *	193-279	16-126

* wenn das Argument im Bereich $-\pi/4 < x < \pi/4$ liegt. Andernfalls müssen 76 Takte für die Standardisierung hinzugerechnet werden.

Opcodes

#	B1	B2
	D9	FE

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception													
	#D	2	#IA	1, 5	#IS	1	#O	./.	#P	1	#U	1	#Z	./.

Bemerkungen FSIN überschreibt den im TOS stehenden Wert durch seinen Sinus. Im Gegensatz zu FPTAN braucht hierbei der Wert nicht auf den Bereich $0 \le |x| < \pi/4$ reduziert werden, es sind Argumente bis 2^{63} zugelassen. Die angegebene Taktzahl des Befehls gilt jedoch nur, falls das Argument im TOS in dem Bereich $0 \le |x| < \pi/4$ liegt. Für alle Werte außerhalb dieses Bereichs müssen 76 Takte für die Standardisierung hinzugezählt werden.

Beschreibung Seite 138

FSINCOS 80387

Funktion Bildung des Sinus und Cosinus einer Zahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				*	*	?			*	*			*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. C2 signalisiert in diesem Fall, daß eine Reduktion auf den gültigen Wertebereich nicht durchgeführt werden konnte und der Befehl nicht ausgeführt wurde! Ist neben dem Invalid-Operation-Flag auch das Stack-Overflow-Flag gesetzt, so zeigt C1 an, ob ein Stack-Underflow (C1 = 1) oder ein Stack-Overflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FSINCOS

Takte

#	8087	80287	80387	80487	Pentium
	-	-	194-809 *	243-329	17-137

* wenn das Argument im Bereich $-\pi/4 < x < \pi/4$ liegt. Andernfalls müssen 76 Takte für die Standardisierung hinzugerechnet werden.

Opcodes

#	B1	B2
	D9	FB

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception															
	#D	2	#IA	1, 5	#IS	1	#O	./.	#P	1	#U	1	#Z	./.		

Bemerkungen FSINCOS führt zunächst ein Pushen des Stacks durch. Anschließend wird der TOS mit dem Sinus des Wertes in ST(1), also des Arguments, das vor dem Pushen im TOS stand, geladen. ST(1) wird mit dem Cosinus überschrieben. Im Gegensatz zu FPTAN braucht hierbei der Wert nicht auf den Bereich $0 \le |x| < \pi/4$ reduziert werden, es sind Argumente bis 2^{63} zugelassen. Die angegebene Taktzahl des Befehls gilt jedoch nur, falls das Argument im TOS in dem Bereich $0 \le |x| < \pi/4$ liegt. Für alle Werte außerhalb dieses Bereichs müssen 76 Takte für die Standardisierung hinzugezählt werden.

Beschreibung Seite 138

FSQRT *8087*

Funktion Bildung der Quadratwurzel einer Zahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*				*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat. Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FSQRT

Takte

#	8087	80287	80387	80487	Pentium
	180-186	180-186	122-129	83-87	70

Opcodes

#	B1	B2
	D9	FA

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 3	#IS 1	#O ./.	#P 1	#U ./.	#Z ./.

Bemerkungen FSQRT überschreibt den Wert im TOS mit seiner Quadratwurzel. Erlaubt sind lediglich positive Zahlen.

Beschreibung Seite 100

FST 8087

Funktion Abspeichern der Zahl im TOS.

Flags	B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
		?				?	*	?		*	*	*	*			*
(80x86-Flags)		Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat. Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	ST(i)	FST ST(4)
2	shortreal	FST ShortRealVar
3	longreal	FST LongIRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	15-22	15-22	11	3	1
2	84-90+EA	84-90	44	7	2
3	96-104+EA	96-104	45	8	2

Opcodes

#	B1	B2	B3	B4
1	DD	D0+i		
2	D9	/2	a16	
3	DD	/2	a16	

Bemerkungen
i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception													
	#D	./.	#IA	1	#IS	1	#O	1	#P	1	#U	1	#Z	./.

Bemerkungen FST ist der allgemeine Speicherbefehl. Mit ihm kann sowohl der Inhalt vom TOS in ein Register als auch in den Speicher kopiert werden.

FST speichert nur Realzahlen. Hierzu konvertiert der Befehl die TEMPREAL im TOS in das Realzahlformat, bevor die Speicherung erfolgt. Für Integer und BCDs ist dieser Befehl daher nicht geeignet! Für BCDs ist der Befehl FBST zuständig, für Integer FIST. FST speichert keine TEMPREAL-Zahlen! Hierzu muß der Befehl FSTP verwendet werden.

Beschreibung Seite 93

FSTCW *8087*
FNSTCW

Funktion FSTCW speichert das Kontrollwort aus dem Kontrollwortregister in eine Wortvariable.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags)

Z		P		C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	word	FSTCW WordVar

Takte

#	8087	80287	80387	80487	Pentium
	12-18+EA	12-18	15	4-6	2*

* zuzüglich mindestens 1 Takt für FWAIT bei FSTCW

Opcodes

#	B1	B2	B3	B4
	D9	/7	a16	

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen FSTCW ist das Pendant zu FLDCW, mit dem ein Kontrollwort aus dem Speicher geladen werden kann. Einzelheiten siehe dort!

FNSTCW ist eine Variante von FSTCW, bei der der Assembler anstelle des sonst üblichen WAIT-Befehls einen NOP-Befehl vor den eigentlichen Befehl stellt.

Beschreibung Seite 113

FSTENV *8087*
FNSTENV

Funktion FSTENV speichert die aktuelle Coprozessorumgebung. Dies sind alle coprozessorspezifischen Register außer den Rechenregistern.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I

(80x86-Flags)

Z		P		C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1a	14-Byte-Var	FSTENV Array14	16-Bit-Umgebungen
1b	28-Byte-Var	FSTENV Array28	32-Bit-Umgebungen

Takte

#	8087	80287	80387	80487	Pentium
	40-50+EA	40-50	103-104	67	50/48*

* zuzüglich mindestens 13Takte für FWAIT bei FSTENV

Opcodes

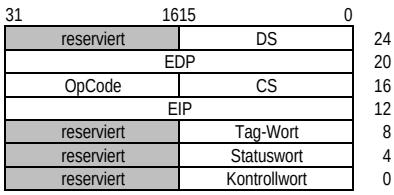
#	B1	B2	B3	B4
	D9	/6	a16	

Exceptions

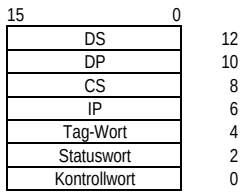
	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen FSTENV ist das Pendant zu FLDENV, mit dem eine Coprozessorumgebung geladen werden kann. FNSTENV ist eine Variante von FSTENV, bei der der Assembler anstelle des sonst üblichen WAIT-Befehls einen NOP-Befehl vor den eigentlichen Befehl stellt.

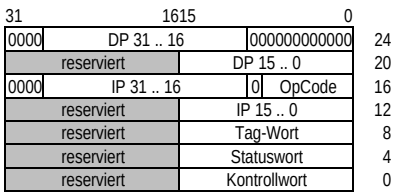
Die Größe und Struktur des Operanden ist von der Umgebung, in der der Befehl verwendet wird, abhängig. So ist, wie die folgenden Schaubilder zeigen, in 16-Bit-Umgebungen der Operand 14 Bytes groß, in 32-Bit-Umgebungen 28 Bytes.



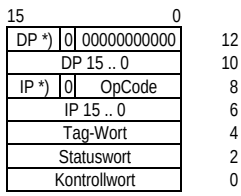
32 Bit Protected-Mode



16 Bit Protected-Mode



32 Bit Real-Mode



16 Bit Protected-Mode

CS:IP bzw. CS:EIP enthält die Adresse (Offset und Selektor) des letzten Nicht-Kontroll-FPU-Befehls, Opcode dessen Opcode. ACHTUNG: CS:IP (CS:EIP) zeigt auf das erste der zum FPU-Befehl gehörenden Präfixe. (Ausnahme: beim 8087 zeigt CS:IP auf den nächsten auszuführenden Befehl!) Opcode enthält die 11 signifikanten Bits des Zwei-Byte-OpCodes des FPU-Befehls. (Da alle FPU-Befehle mit »ESC« und der Bitkombination 11011xxx beginnen, sind lediglich die niedrigstwertigen drei Bits sowie die folgenden acht Bits des nächsten Bytes wesentliche Information, die in Opcode gehalten wird.) DS:DP bzw. DS:EDP enthält die Adresse (Offset und Selektor) des Operanden für den betreffenden Befehl. Nicht-Kontrollbefehle sind alle Befehle außer FSTENV/FNSTENV/FLDENV, FINIT/FNINIT und FSAVE/FNSAVE/FRSTOR.

FSTENV prüft auf anhängige Exceptions und behandelt diese, bevor die Sicherung der Umgebung und der Register erfolgt. FNSTENV tut dies nicht. Nach der Sicherung werden alle Exceptions maskiert.

Beschreibung Seite 113

FSTP 8087

Funktion Abspeichern und Entfernen der Zahl im TOS vom Stack.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*			*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
1	ST(i)	FST ST(4)
2	shortreal	FST ShortRealVar
3	longreal	FST LongIRealVar
4	tempreal	FST TempRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	17-24	17-24	12	3	1
2	86-92+EA	86-92	44	7	2
3	98-106+EA	98-106	45	8	2
4	52-58+EA	52-58	53	6	3

Opcodes

#	B1	B2	B3	B4
1	DD	D8+i		
2	D9	/3	a16	
3	DD	/3	a16	
4	DB	/7	a16	

Bemerkungen
i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception															
	#D	./.	#IA	1	#IS	1	#O	1	#P	1	#U	1	#Z	./.		

Bemerkungen FSTP arbeitet wie FST, nur daß nach dem Abspeichern der Zahl der Stack-Pointer inkrementiert und ST(7) auf *empty* gesetzt wird. FSTP ist der einzige Befehl, der TEMPREAL-Variablen als Ziel der Speicherung benutzen kann.

Beschreibung Seite 93

FSTSW *8087*
FNSTSW

Funktion FSTSW speichert das Statuswort aus dem Statuswortregister in eine Wortvariable.

Flags B C3 X X X C2 C1 C0 X S P U O Z D I

(80x86-Flags) Z P C

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel	Bemerkungen
1	word	FSTSW WordVar	
2	AX	FSTSW AX	ab 80287

Takte

#	8087	80287	80387	80487	Pentium
1	12-18+EA	12-18	15	3	2*
2	-	10-16	13	3	2*

* zuzüglich mindestens 3 Takte für FWAIT bei FSTSW

Opcodes

#	B1	B2	B3	B4
1	DD	/7	a16	
2	DF	E0		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen FSTSW ist die einzige Möglichkeit, an das Statuswort des Coprozessors heranzukommen, um die Flagstellungen auszuwerten. Beim 8087 ist der Umweg über eine Wortvariable notwendig, ab dem 80287 kann direkt in das AX-Register kopiert werden. Dies wird beim 8087 über die Sequenz *FSTSW WordVar; MOV AX, WordVar* emuliert.

FNSTSW ist eine Variante von FSTSW, bei der der Assembler anstelle des sonst üblichen WAIT-Befehls einen NOP-Befehl vor den eigentlichen Befehl stellt.

Beschreibung Seite 114, 126

FSUB *8087*

Funktion Subtraktion zweier Realzahlen

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	ST, ST(i)	FSUB ST, ST(4)
2	ST(i), ST	FSUB ST(2), ST
3	shortreal	FSUB ShortRealVar
4	longreal	FSUB LongRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	70-100	70-100	26-37	5-17	3/1
2	70-100	70-100	26-37	5-17	3/1
3	90-120+EA	90-120	24-32	5-17	3/1
4	95-125+EA	95-125	28-36	5-17	3/1

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	D8	E0+i			i = Registernummer
2	DC	E8+i			i = Registernummer
3	D8	/4	a16		
4	DC	/4	a16		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception						
	#D 2	#IA 1, 6	#IS 1	#O 1	#P 1	#U 1	#Z ./.

Bemerkungen

FSUB ist der allgemeine Befehl, um zwei Zahlen zu subtrahieren. FSUB subtrahiert Integers, BCDs und Reals, solange nur Stackregister als Operanden angegeben werden (also Fall 1 und 2 in obiger Liste). Der Grund dafür ist, daß all diese Zahlen intern als 80-Bit-TEMPREAL dargestellt werden!

Darüber hinaus ist mit FSUB auch das Subtrahieren einer Zahl aus dem Speicher vom TOS möglich. Dies ist aber nur mit Realzahlen möglich, wobei FSUB automatisch die Konvertierung des Realformats in die TEMPREAL-Darstellung vornimmt. FSUB ist daher nicht auf Integerzahlen anwendbar – hierfür gibt es den Befehl FISUB, der seinerseits für die korrekte Konvertierung zuständig ist!

Beschreibung Seite 95

FSUBP 8087

Funktion Subtraktion zweier Realzahlen und Poppen des Stacks.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST(i), ST	FSUBP ST(4), ST

Takte

#	8087	80287	80387	80487	Pentium
	75-105	75-105	26-37	5-17	3/1

Opcodes

#	B1	B2	Bemerkungen
	DE	E8+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception											
	#D	2	#IA	1, 6	#IS	1	#O	1	#P	1	#U	1	#Z	./.

Bemerkungen FSUBP arbeitet wie FSUB. Allerdings wird im Anschluß an die Subtraktion ein Poppen des Stacks durchgeführt und ST(7) als *empty* markiert.

Da FSUBP nach der Subtraktion den Wert im TOS entfernt und ein Poppen des Stacks durchführt, macht die Konstruktion *FSUBP ST,ST(i)* keinen Sinn und wird daher vom Assembler nicht unterstützt!

Beschreibung Seite 95

FSUBR 8087

Funktion Reziproke Subtraktion zweier Realzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
1	ST, ST(i)	FSUBR ST, ST(4)
2	ST(i), ST	FSUBR ST(2), ST
3	shortreal	FSUBR ShortRealVar
4	longreal	FSUBR LongRealVar

Takte

#	8087	80287	80387	80487	Pentium
1	70-100	70-100	26-37	5-17	3/1
2	70-100	70-100	26-37	5-17	3/1
3	90-120+EA	90-120	24-32	5-17	3/1
4	95-125+EA	95-125	28-36	5-17	3/1

Opcodes

#	B1	B2	B3	B4	Bemerkungen
1	D8	E8+i			i = Registernummer
2	DC	E0+i			i = Registernummer
3	D8	/5	a16		
4	DC	/5	a16		

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception							
	#D 2	#IA 1, 6	#IS 1	#O 1	#P 1	#U 1	#Z ./.	

Bemerkungen FSUBR arbeitet wie FSUB. Allerdings wird die Reihenfolge der Operanden umgekehrt. Während bei FSUB der zweite Operand (die Quelle) vom ersten Operanden (dem Ziel) abgezogen wird, erfolgt bei FSUBR eine Subtraktion des Ziels von der Quelle. Das Ergebnis jedoch wird in beiden Fällen in den ersten Operanden (das Ziel) eingetragen.

Beschreibung Seite 96

FSUBRP *8087*

Funktion Reziproke Subtraktion zweier Realzahlen und Poppen des Stacks.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*	*		*	*

(80x86-Flags) Z P C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST(i), ST	FSUBRP ST(4), ST

Takte

#	8087	80287	80387	80487	Pentium
	75-105	75-105	26-37	5-17	3/1

Opcodes

#	B1	B2	Bemerkungen
	DE	E0+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

#MF	Grund: FPU-Exception													
	#D	2	#IA	1, 6	#IS	1	#O	1	#P	1	#U	1	#Z	./.

Bemerkungen FSUBRP arbeitet wie FSUBP. Allerdings wird die Reihenfolge der Operanden umgekehrt. Während bei FSUBP der zweite Operand (die Quelle) vom ersten Operanden (dem Ziel) abgezogen wird, erfolgt bei FSUBRP eine Subtraktion des Ziels von der Quelle. Das Ergebnis jedoch wird in beiden Fällen in den ersten Operanden (das Ziel) eingetragen.

Da FSUBRP nach der Subtraktion den Wert im TOS entfernt und ein Poppen des Stacks durchführt, macht die Konstruktion *FSUBRP ST,ST(i)* keinen Sinn und wird daher vom Assembler nicht unterstützt!

Beschreibung Seite 96

FTST *8087*

Funktion Vergleichen des TOS mit der Konstanten 0.0

Flags	B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
		*				(*)	(*)	*		*					*	*
(80x86-Flags)		Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Falls das Invalid-Operation-Flag *nicht* gesetzt ist, werden lediglich die Flags C3 und C0 verändert. Mögliche Zustände sind:

C3	C0	Bedeutung
0	0	TOS > 0
0	1	TOS < 0
1	0	TOS = 0
1	1	TOS nicht mit 0 vergleichbar

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FTST

Takte

#	8087	80287	80387	80487	Pentium
	38-48	38-48	28	4	4/1

Opcodes

#	B1	B2
	D9	E4

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception													
	#D	2	#IA	1	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FTST arbeitet wie FCOM, wobei jedoch kein Parameter übergeben werden kann. FTST vergleicht den Inhalt vom TOS mit der Konstanten 0.0 und setzt anhand des Ergebnisses den Condition Code im Kontrollwortregister des Coprozessors.

Beschreibung Seite 111

FUCOM *80387*

Funktion »Ungeordneter« Vergleich zweier Realzahlen.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				(*)	(*)	*							*	

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Dies ist dann der Fall, wenn mindestens ein Operand eine *sNaN* (*signaling NaN*) ist oder ein undefiniertes Format besitzt. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Der Coprozessor setzt das Invalid-Operation-Flag *nicht*, falls mindestens einer der Operanden eine *qNaN* (*quiet NaN*) ist oder beide Operanden gültig sind. Statt dessen wird der Condition Code verändert. Mögliche Zustände der Bits C3, *C2* (!) und C0 sind:

C3	C2	C0	Bedeutung
0	0	0	Operand 1 > Operand 2
0	0	1	Operand 1 < Operand 2
1	0	0	Operand 1 = Operand 2
1	1	1	Operanden ungeordnet

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	ST(i)	FUCOM ST(4)

Takte

#	8087	80287	80387	80487	Pentium
	-	-	24	4	4/1

Opcodes

#	B1	B2	Bemerkungen
	DD	E0+i	i = Registernummer

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception											
	#D	2	#IA	10	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FUCOM ist ein Spezialfall von FCOM. Während nämlich bei letzterem die Verwendung einer NaN zu einer Exception, also einem Ausnahmezustand, führt, die durch Setzen des entsprechenden Flags angezeigt wird, behandelt FUCOM NaNs als gültige Zahlen, die lediglich nicht verglichen werden können! Es wird somit nur durch Setzen der Bits C3, C2 und C0 des Condition Codes angezeigt, daß ein Vergleich nicht möglich war. FUCOM kann, im Gegensatz zu FCOM, nur ein Register als Operanden verwenden, keine Speicherstelle!

Beschreibung Seite 140

FUCOMI — Pentium Pro

Funktion: Ungeordneter Vergleich zweier Realzahlen und Setzen des Ergebnisses in EFlags.

Flags:

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									*				*		*

ACHTUNG: Dieser Befehl verändert die Flags im EFlagregister, nicht etwa die Coprozessorflags, obwohl es sich um einen Coprozessorbefehl handelt!

Zero-Flag, Parity-Flag und Carry-Flag werden wie folgt gesetzt:

ZF	PF	CF	Bedeutung
0	0	0	ST(0) > ST(i)
0	0	1	ST(0) < ST(i)
1	0	0	ST(0) = ST(i)
1	1	1	ungeordnet.

Stack-Pointer: ± 0

Verwendung:

#	Parameter	Beispiel
	ST, ST(i)	FUCOMI ST, ST(7)

Takte:

#	8087	80287	80387	80487	Pentium
	-	-	-	-	-

Opcodes:

#	B1	B2	B3	B4	Bemerkungen
	DB	E8+i			i=Registernummer

Exceptions:

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception															
	#D	./.	#IA	10	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.		

Bemerkungen: FUCOMI arbeitet wie FUCOM, setzt aber anstelle der Coprozessorflags die Prozessorflags entsprechend dem Ergebnis des Vergleichs.

Beschreibung: Seite 149

FUCOMIP *Pentium Pro*

Funktion Vergleich zweier Realzahlen und Setzen des Ergebnisses in EFlags. Anschließend wird der Stack gepoppt.

Flags

X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C
									*				*		*

ACHTUNG: Dieser Befehl verändert die Flags im EFlagregister, nicht etwa die Coprozessorflags, obwohl es sich um einen Coprozessorbefehl handelt!

Zero-Flag, Parity-Flag und Carry-Flag werden wie folgt gesetzt:

ZF	PF	CF	Bedeutung
0	0	0	ST(0) > ST(i)
0	0	1	ST(0) < ST(i)
1	0	0	ST(0) = ST(i)
1	1	1	ungeordnet.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST, ST(i)	FUCOMIP ST, ST(1)

Takte

#	8087	80287	80387	80487	Pentium
	-	-	-	-	-

Opcodes

#	B1	B2	B3	B4	Bemerkungen
	DF	E8+i			i=Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception											
	#D	./.	#IA	10	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FUCOMIP arbeitet wie FUCOMI, poppt jedoch nach dem Vergleich den Stack.

Beschreibung Seite 149

FUCOMP 80387

Funktion »Ungeordneter« Vergleich des TOS-Inhalts mit einer Realzahl.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				(*)	(*)	*		*					*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Dies ist dann der Fall, wenn mindestens ein Operand eine *sNaN* (*signaling NaN*) ist oder ein undefiniertes Format besitzt. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Der Coprozessor setzt das Invalid-Operation-Flag *nicht*, falls mindestens einer der Operanden eine *qNaN* (*quiet NaN*) ist oder beide Operanden gültig sind. Statt dessen wird der Condition Code verändert.

Mögliche Zustände der Bits C3, *C2* (!) und C0 sind:

C3	C2	C0	Bedeutung
0	0	0	Operand 1 > Operand 2
0	0	1	Operand 1 < Operand 2
1	0	0	Operand 1 = Operand 2
1	1	1	Operanden ungeordnet

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	ST(i)	FUCOMP ST(4)

Takte

#	8087	80287	80387	80487	Pentium
	-	-	26	4	4/1

Opcodes

#	B1	B2	Bemerkungen
	DD	E8+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception															
	#D	2	#IA	10	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.		

Bemerkungen FUCOMP arbeitet wie FUCOM, nur wird nach dem Vergleich der Stack-Pointer um 1 inkrementiert, so daß ein Poppen des Stacks durchgeführt wird. ST(7) ist danach als *empty* markiert.

FUCOMP kann, im Gegensatz zu FCOMP, nur ein Register als Operanden verwenden, keine Speicherstelle!

Beschreibung Seite 140

FUCOMPP *80387*

Funktion »Ungeordneter« Vergleich des TOS-Inhalts mit dem Inhalt von ST(1).

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				(*)	(*)	*		*					*	*

(80x86-Flags)

Z			P		C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Dies ist dann der Fall, wenn mindestens ein Operand eine *sNaN* (*signaling NaN*) ist oder ein undefiniertes Format besitzt. In diesem Fall sind die Bits C3, C2 und C0 gesetzt und signalisieren unvergleichbare Operanden. Falls bei gesetztem Invalid-Operation-Flag auch das Stack-Fault-Flag gesetzt ist, zeigt C1 einen Stack-Underflow (0) oder Stack-Overflow (1) an. Andernfalls ist C0 = 0.

Der Coprozessor setzt das Invalid-Operation-Flag *nicht*, falls mindestens einer der Operanden eine *qNaN* (*quiet NaN*) ist oder beide Operanden gültig sind. Statt dessen wird der Condition Code verändert.

Mögliche Zustände der Bits C3, *C2* (!) und C0 sind:

C3	C2	C0	Bedeutung
0	0	0	Operand 1 > Operand 2
0	0	1	Operand 1 < Operand 2
1	0	0	Operand 1 = Operand 2
1	1	1	Operanden ungeordnet

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	keine	FUCOMPP

Takte

#	8087	80287	80387	80487	Pentium
	-	-	26	5	4/1

Opcodes

#	B1	B2
	DA	E9

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	2	#IA	10	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FUCOMPP arbeitet wie FUCOM, nur wird nach dem Vergleich der Stack-Pointer um 2 inkrementiert, so daß ein doppeltes Poppen des Stacks durchgeführt wird. ST(6) und ST(7) sind danach als *empty* markiert.

Beschreibung Seite 140

FWAIT *8087*

Funktion Anhalten des Prozessors während der Aktivität des Coprozessors.

Takte

#	8087	80287	80387	80487	Pentium
	-	-	3+5n*	1-4	1-3

* n gibt an, wie schnell die CPU die BUSY-Leitung abfragt, nachdem der Coprozessor den Befehl abgearbeitet hat.

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	3	-	3	3

Bemerkungen FWAIT ist kein echter Coprozessorbefehl. Es ist vielmehr der Prozessorbefehl WAIT. Der Assembler übersetzt FWAIT in den gleichen Opcode wie WAIT. Da ab dem 80387 die Synchronisation mit dem Coprozessor auf anderem Wege erfolgt, kann hier ein Wert für die Ausführungsdauer angegeben werden.

Beschreibung Seite 115

FXAM 8087

Funktion Untersuchung der Zahl im TOS.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				*	*	*								

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt lediglich die Bits des Condition Codes (siehe Seite 815).

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	FXAM

Takte

#	8087	80287	80387	80487	Pentium
	12-23	12-23	30-38	8	21

Opcodes

#	B1	B2
	D9	E5

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	2	-	2	2

Bemerkungen FXAM untersucht die Zahl im TOS und stellt das Ergebnis der Untersuchung als Condition Code im Kontrollwort dar.

Beschreibung Seite 111

FXCH 8087

Funktion Vertauschen der Inhalte zweier Rechenregister.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*						*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Stack-Fault- und das Invalid-Operation-Flag, falls ein Stack-Underflow stattfindet. Andere Flags werden *nicht* gesetzt, selbst wenn im TOS eine NaN vorgefunden wird.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	ST(i)	FXCH St(4)

Takte

#	8087	80287	80387	80487	Pentium
	10-15	10-15	18	4	1

Opcodes

#	B1	B2	Bemerkungen
	D9	C8+i	i = Registernummer

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	./.	#IA	./.	#IS	1	#O	./.	#P	./.	#U	./.	#Z	./.

Bemerkungen FXCH dient zum Vertauschen des Inhalts des TOS mit einem anderen Register.

Beschreibung Seite 111

FXRSTOR *MMX*

Funktion Wiederherstellen der FPU- bzw. MMX-Umgebung

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	512-Byte-Var	FXRSTOR Array512

Arbeitsweise FXRSTOR ist eine MMX-Anpassung des FRSTOR-Befehls des Coprozessorbefehlssatzes. Der Befehl wurde auf die veränderten Bedingungen beim Umgang der Register mit MMX-Befehlen angepaßt. Er lädt die verschiedenen Status- und Kontrollregister sowie die FPU- bzw. MMX-Register mit den Werten, die FXSAVE in eine Speichervariable eingetragen hat.

Opcodes

#	B1	B2	B3	B4	B5
	0F	AE	/1	a16	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1

Bemerkungen

FXRSTOR erwartet, daß das obere Byte des FPU-Tag-Wortes (Bits 15-8), das im Operanden gespeichert wurde, auf »0« gesetzt ist (FXSAVE bewerkstelligt das, FSAVE/FNSAVE *nicht*!). Erfolgt dies nicht, führt die Ausführung von FXRSTOR zu einer falschen Umgebung für den Prozessor mit unvorhersehbaren Auswirkungen bis hin zu Systemabstürzen.

Zur Struktur und Ausrichtung des Operanden und weiteren Einzelheiten siehe FXSAVE.

Anders als FRSTOR behandelt FXRSTOR es nicht als Fehler, wenn Informationen über anhängige Exception*s* aus dem Operanden geladen werden (die von FXSAVE dort gespeichert wurden!). Vielmehr könnte lediglich das nächste Auftreten einer als anhängig markierten Exception zu Problemen führen. FXRSTOR behandelt selbst auch nicht anhängige Exception*s* oder löscht deren Bits. Falls dies gewünscht wird, muß dem FXRSTOR-Befehl ein FWAIT folgen.

ACHTUNG: Vor FXRSTOR sollten keine Präfixe wie REP, REPNE oder OPSIZE ($66) stehen! Diese Präfix-Befehlskombinationen gelten als reserviert und können, wenn sie dennoch benutzt werden, zu Inkompatibilitäten bei unterschiedlichen Prozessoren führen. So könnte eine solche Kombination in künftigen Prozessoren andere, nicht vorhergesehene Reaktionen bewirken. Das ADRSIZE-Präfix ($67) hat zwar Einfluß auf die Adressenberechnung, nicht aber auf das Verhalten von FXRSTOR.

Die Befehle der Kombinationen FSAVE/FNSAVE-FRSTOR und FXSAVE-FXRSTOR dürfen nicht gemischt werden. Die Übergabe eines mittels FSAVE erstellten FPU-Abbildes an FXRSTOR hätte das Wiederherstellen der FPU/MMX-Einheit mit nicht korrekten Daten zur Folge und würde zu unvorhersehbaren Problemen bis hin zum Systemabsturz führen. Zu Einzelheiten siehe FXSAVE.

Beschreibung

Seite 169.

FXSAVE MMX

Funktion Sichern einer FPU- bzw. MMX-Umgebung.

Flags X X X X O D I T S Z X A X P X C

Die Flags werden nicht verändert.

Verwendung

#	Parameter	Beispiel
	512-Byte-Var	FXSAVE Array512

Arbeitsweise FXSAVE ist eine MMX-Anpassung des FSAVE-Befehls des Coprozessorbefehlssatzes. Der Befehl wurde auf die leicht veränderten Bedingungen beim Umgang der Register mit MMX-Befehlen angepaßt. Er speichert die verschiedenen Status- und Kontrollregister sowie die FPU- bzw. MMX-Register. Hierbei wartet FXSAVE nicht darauf, daß die Operation ausgeführt wird, und ähnelt insofern dem FNSAVE-Befehl. Soll auf den Abschluß der Operation gewartet werden, so hat dem FXSAVE-Befehl ein FWAIT zu folgen!

Opcodes

#	B1	B2	B3	B4	B5
	0F	AE	/0	a16	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	3, 8, 20	0	8	8
#NM	-	2	-	2	2
#PF	?	1	?	1	./.
#SS	0	1	0	1	1
#UD	-	12			

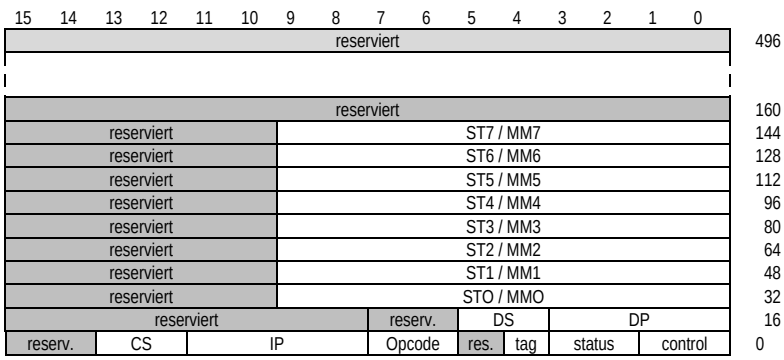
Bemerkungen FXSAVE geht davon aus, daß der Operand an 16-Byte-Grenzen ausgerichtet ist. Andernfalls wird eine #GP-Exception ausgelöst.

ACHTUNG: Vor FXSAVE sollten keine Präfixe wie REP, REPNE oder OPSIZE ($66) stehen! Diese Präfix-Befehlskombinationen gelten als reserviert und können, wenn sie dennoch benutzt werden, zu Inkompatibilitäten bei unterschiedlichen Prozessoren führen. So könnte eine solche Kombination in künftigen Prozessoren andere, nicht vorhergesehene Reaktionen bewirken. Das ADRSIZE-Präfix ($67) hat zwar Einfluß auf die Adressenberechnung, nicht aber auf das Verhalten von FXSAVE.

Vorsicht bei der Verwendung der Befehlssequenz FXSAVE, ..., FWAIT, ..., FXRSTOR. Falls bei der Ausführung von FXSAVE Exceptions anhängig sind, sichert FXSAVE zwar die Umgebung, initialisiert den Coprozessor je-

doch nicht neu. Das folgende FWAIT findet daher die wartenden Exceptions und behandelt sie; die Information darüber wird aber nirgendwo gespeichert. Daher restauriert FXRSTOR eine »falsche« Umgebung, indem noch anhängige Exceptions angezeigt werden, die längst behandelt wurden. FXRSTOR ruft die entsprechenden Handler nochmals auf – was zu Problemen führen kann.

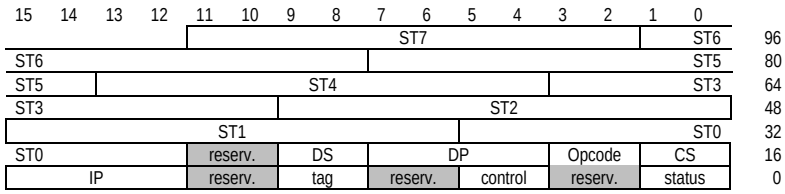
Der Operand muß an 16-Byte-Grenzen ausgerichtet sein und folgenden Aufbau besitzen:



Bit 7 bis 0 des Wortes an Offset 0 nehmen das Kontrollwort der FPU-Umgebung auf, Bit 7 bis 0 des Wortes an Offset 2 das Statuswort. Byte 4 und 5 der Struktur repräsentieren das Tag-Wort, wobei, wie gesagt, FXRSTOR die Bits 15 bis 8 dieses Wortes als reserviert behandelt. In den Bytes 13 und 12 bzw. 21 und 20 stehen die Selektoren für das Code- und Datensegment, die Bytes 11 bis 8 bzw. 19 bis 16 stellen die Pointer in das Code- bzw. Datensegment dar, an denen der vorangegangene Nicht-Kontroll-FPU- bzw. MMX-Befehl steht bzw. mit dem der Datenaustausch mit dem dazugehörigen Operanden erfolgt. (In 32-Bit-Umgebungen enthalten IP und DP 32-Bit-Offsets. In 16-Bit-Umgebungen werden die 16-Bit-Offsets in den niedrigstwertigen 16 Bits kodiert, die höherwertigen sind reserviert!) Der entsprechende FPU-Befehl steht im Opcode, wobei nur die Bits 10 bis 0 verwendet werden. Die Bits 15 bis 11 sind reserviert. (Da alle FPU-Befehle mit »ESC« sowie der Bitkombination 11011xxx beginnen, sind nur die niedrigstwertigen drei Bits sowie die acht Bits des folgenden Bytes für den FPU-Befehl ausschlaggebend. Diese werden im Opcode gesichert.) Falls der Befehl, auf den CS:IP zeigt und der im Opcode eingetragen ist, keinen Operanden benötigt, wird DS:DP auch nicht gesichert! Die Einträge haben dann als reserviert zu gelten!

Die jeweils ersten 10 von 16 Bytes ab dem Offset 32 der Datenstruktur nehmen die Inhalte der FPU- bzw. MMX-Register auf, die restlichen 6 Bytes gelten als reserviert. Ab Offset 160 bis zum Ende der Struktur sind alle Bytes als reserviert zu betrachten.

ACHTUNG: Die Strukturen der Operanden für FSAVE/FNSAVE und FXSAVE unterscheiden sich, wie der Vergleich mit dem nächsten Schaubild zeigt (zum Vergleich ist hier die 108-Byte-Struktur, die FSAVE/FNSAVE im Protected-Mode mit 32-Bit-Operanden verwendet, im analogen Aufbau zu oben gezeigt).



Aus diesem Grund sollten die Befehle der Kombinationen FSAVE/FNSAVE-FRSTOR und FXSAVE-FXRSTOR nicht gemischt werden. Die Übergabe eines mittels FXSAVE erstellten FPU/MMX-Abbildes an FRSTOR hätte das Wiederherstellen der FPU/MMX-Einheit mit nicht korrekten Daten zur Folge und würde zu unvorhersagbaren Problemen bis hin zum Systemabsturz führen.

Anders als FSAVE und analog zu FNSAVE prüft FXSAVE vor der Sicherung der Daten *nicht* auf anhängige Exceptions! Der Coprozessor wird auch im Unterschied zu FSAVE/FNSAVE nach der Sicherung der Daten *nicht* initialisiert, so daß der Zustand auch nach FXSAVE erhalten bleibt! FXSAVE wurde auf maximale Sicherungs-Performance hin optimiert.

FXSAVE codiert die Inhalte des Tag-Worts des Coprozessors anders als FSAVE. Während FSAVE den tatsächlichen Inhalt der Tag-Felder der einzelnen Register sichert, sichert FXSAVE lediglich die Information, ob das Register als leer markiert ist oder nicht. Folgendes Schaubild soll das erläutern:

Zustand des betreffenden Registers	valid	zero	special	empty
Code im Tag-Feld des Registers	00	01	10	11
durch FSAVE gesicherter Code	00	01	10	11
durch FXSAVE gesicherter Code	1	1	1	0

FXSAVE unterscheidet also nicht zusätzlich nach Eigenschaften des Inhalts wie »Nullwert«, NaN oder ähnliches. Es trägt damit den Änderungen Rechnung, die durch das »Aufbohren« der FPU-Befehle bzw. -Daten um die MMX-Befehle bzw. -Daten erforderlich waren. Da also lediglich gesichert wird, ob das betreffende Register leer ist oder nicht, wird auch nur ein Bit pro Register notwendig. Für alle verfügbaren Register reicht also ein Byte aus, so daß das Tag-Wort auch lediglich als Byte verwendet wird: Die höherwertigen acht Bits des Wortes gelten als reserviert. (Trotz des vermeintlichen Informationsverlustes läßt sich die notwendige Information auch bei FXSAVE bzw. FXRSTOR wiedergewinnen: Ist das FXSAVE/FXRSTOR Tag-Bit 1, so ist das Register leer und der entsprechende Eintrag in das Tag-Feld das Tag-Worts im Coprozessor ist 11. Andernfalls kann anhand der Exponent-/Mantisse-/Vorzeichenverhältnisse des korrespondierenden Registerinhalts festgestellt werden, ob der Eintrag gültig ist, Null oder einen Spezialfall darstellt. Siehe hierzu »Darstellung von Zahlen im Coprozessorformat«.)

Beschreibung Seite 169.

FXTRACT *8087*

Funktion Zerlegen einer Realzahl in Mantisse und Exponent.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*				*	*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Stack-Pointer - 1

Verwendung

#	Parameter	Beispiel
	keine	FXTRACT

Takte

#	8087	80287	80387	80487	Pentium
	27-55	27-55	70-76	16-20	13

Opcodes

#	B1	B2
	D9	F4

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception						
	#D 2	#IA 1	#IS 1, 2	#O ./.	#P ./.	#U ./.	#Z 3

Bemerkungen FXTRACT ist die Umkehrfunktion zu FSACLE. Sie zerlegt eine Zahl in Mantisse und Exponent. Dazu wird ein Pushen des Stacks durchgeführt. Danach wird in den TOS die Mantisse der Zahl aus ST(1), dem ursprünglichen TOS, geladen. ST(1) wird mit dem Exponenten dieser Zahl überschrieben.

ACHTUNG: Die Zahlen in TOS und ST(1) sind Manitsse und Exponent einer binären Zahlendarstellung! Das bedeutet, daß die Zahl im TOS immer zwischen 1 und 2 liegt. Der Exponent in ST(1) ist immer eine Integer! So wird beispielsweise 32 als 1.000 für die Mantisse und 5 für den Exponenten dargestellt.

Beschreibung Seite 100

FYL2X 8087

Funktion Berechnung des Ergebnisses der Operation Y·ld(X).

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				?	*	?		*	*	*	*	*	*	*

(80x86-Flags) Z P C

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	keine	FYL2X

Takte

#	8087	80287	80387	80487	Pentium
	900-1100	900-1100	120-538	196-329	22-111

Opcodes

#	B1	B2
	D9	F1

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception						
	#D 2	#IA 3, 10	#IS 1	#O 1	#P 1	#U 1	#Z 3

Bemerkungen Die Operation bildet zunächst den Logarithmus Dualis des Werts im TOS (=X) und multipliziert das Ergebnis mit dem Wert in St(1) (=Y). Das Resultat wird in ST(1) abgelegt, und anschließend wird ein Poppen des Stacks durchgeführt. ST(7) wird dabei als *empty* markiert.

Gültige Werte für FYL2X sind: $0 \le ST(0) < \infty$ und $-\infty \le ST(1) < \infty$.

Beschreibung Seite 101

FYL2XP1 *8087*

Funktion Berechnung des Ergebnisses der Operation Y·ld(X+1).

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	*				?	*	?		*	*	*			*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer + 1

Verwendung

#	Parameter	Beispiel
	keine	FYL2XP1

Takte

#	8087	80287	80387	80487	Pentium
	700-1000	700-1000	257-547	171-326	22-103

Opcodes

#	B1	B2
	D9	F9

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception													
	#D	2	#IA	1	#IS	1	#O	1	#P	1	#U	1	#Z	./.

Bemerkungen Die Operation addiert zunächst zum Wert im TOS (=X) 1.0 und bildet dann aus dieser Zahl den Logarithmus Dualis. Das Ergebnis wird mit dem Wert in ST(1) (=Y) multipliziert. Das Ergebnis dieser Operation wird in ST(1) abgelegt, und anschließend wird ein Poppen des Stacks durchgeführt. ST(7) wird dabei als *empty* markiert.

Gültiger Bereich: $0 \le |ST(0)| < (2-\sqrt{2})/2$.

Beschreibung Seite 101

F2XM1 8087

Funktion Berechnung des Ergebnisses der Operation 2^X-1.

Flags

B	C3	X	X	X	C2	C1	C0	X	S	P	U	O	Z	D	I
	?				?	*	?		*	*	*			*	*

(80x86-Flags)

	Z				P		C								

Der Coprozessor setzt das Invalid-Operation-Flag, falls die Operation nicht korrekt durchgeführt werden konnte. Falls zusätzlich das Stack-Fault-Flag gesetzt ist, zeigt C1 an, ob ein Stack-Overflow (C1 = 1) oder ein Stack-Underflow (C1 = 0) stattgefunden hat.

Bei gesetztem Precision-Exception-Flag zeigt C1 an, ob die Rundung zu größeren (C1 = 1) oder kleineren (C1 = 0) Werten durchgeführt wurde.

Stack-Pointer ± 0

Verwendung

#	Parameter	Beispiel
	keine	F2XM1

Takte

#	8087	80287	80387	80487	Pentium
	900-1100	900-1100	120-538	140-279	13-57

Opcodes

#	B1	B2
	D9	F0

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#NM	-	2	-	2	2

#MF	Grund: FPU-Exception									
	#D	2	#IA	1	#IS	1	#O	./.	#P	1	#U	1	#Z	./.

Bemerkungen Diese Operation ist die Umkehrung zu FYL2XM1. Sie verwendet den Inhalt vom TOS zur Potenzierung der Basis 2 und subtrahiert 1 vom Ergebnis. Das Resultat wird benutzt, um den TOS zu überschreiben, so daß sich am Zustand des Stacks nichts ändert.

Gültige Argumente liegen im Bereich $0 \le ST(0) \le 2$.

Beschreibung Seite 102

37 MMX-Befehle

Bei Prozessoren mit der MMX-Technologie wurde der Befehlssatz um die MMX-Befehle erweitert. Hierbei handelt es sich um Befehle, die die Register der Floating-Point-Unit nutzen, aber nicht zu den FPU-Befehlen gerechnet werden können. Zu Einzelheiten vgl. Teil 1.

Aufbau und Information der Referenzen sind die gleichen wie für die CPU-Befehle im vorletzten Kapitel. Vgl. daher die Vorbemerkungen zu diesem Referenzteil. Es wurden im folgenden lediglich folgende Abkürzungen zusätzlich verwendet:

mm bezeichnet ein MMX-Register. Das betreffende Register wird mit den Mnemonics MM0 bis MM7 angegeben.

m64 64-Bit-Speicherstelle

Beachten Sie bitte, daß MMX-Befehle keine Flags verändern – weder CPU-Flags im EFlags-Register noch FPU-Flags im Statuswort. Daher wurde auf die Ausweisung der Flagzustände verzichtet.

MMX-Befehle erzeugen, obwohl sie in der FPU ausgeführt, werden, keine FPU-Exceptions! Sie können jedoch sehr wohl eine FPU-Exception veranlassen, falls ein vor einem MMX-Befehl ausgeführter FPU-Befehl zu einer Exception geführt hat, die jedoch nicht bearbeitet wurde. In diesem Falle wird sie vor der Bearbeitung des MMX-Befehls ausgelöst.

EMMS *MMX*

Funktion Den MMX-Status als »gelöscht« definieren.

Verwendung

#	Parameter	Beispiel
	keine	EMMS

Arbeitsweise EMMS – *Empty MM State* – setzt die Tag-Bereiche in den FPU-Registern auf »0« und definiert die korrespondierenden Register damit als »leer«. Sie können dadurch wieder mit FPU-Befehlen benutzt werden.

Opcodes

#	B1	B2
	0F	77

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#NM	-	1	-	1	1

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen

Da die MMX-Befehle auf die gleichen Register wirken wie die FPU-Befehle, kann es bei der Verwendung von FPU- und MMX-Befehlen, besonders bei deren Mischung, zu Problemen kommen – solange die betreffenden Routinen nicht sauber »gekapselt« werden und für eine definierte Eintritts- und Austrittssituation gesorgt wird. EMMS kann dies bewerkstelligen.

Da die FPU-Befehle historisch bedingt die älteren Befehle sind, kann nicht davon ausgegangen werden, daß bestehende Anwendungen diese saubere Kapselung bereits vornehmen. Es liegt damit im Verantwortungsbereich des Programmierers der MMX-Befehle, dies zu tun. Er kann das, indem er beim Eintritt in ein MMX-Modul die FPU-Umgebung sichert (z.B. mit FSTENV). Vor dem Verlassen des MMX-Moduls sollte dann mittels FLDENV der ursprüngliche FPU-Status wiederhergestellt werden. Ist eine Sicherung der FPU-Umgebung nicht notwendig, ersetzt EMMS den FLDENV-Befehl.

EMMS wirkt auf alle acht Register gleichzeitig, einzelne Register lassen sich mit diesem Befehl nicht markieren.

Beschreibung

Seite 169.

MOVD
MOVQ *MMX*

Funktion

Kopieren eines Doppelwortes (MOVD) oder Quadworts (MOVQ) in oder von einem MMX-Register.

Verwendung

#	Parameter	Beispiel
1	mm, r32	MOVD MM4, EDX
2	mm, m32	MOVD MM2, DVar
3	r32, mm	MOVD EAX, MM7
4	m32, mm	MOVD DVar, MM1
5	mm, mm	MOVQ MM4, MM3

6	mm, m64	MOVQ MM2, QVar
7	mm, mm	MOVQ MM0, MM7
8	m64, mm	MOVQ QVar, MM1

Arbeitsweise

MOVD verwendet als Operanden 32-Bit-Werte (DWords), MOVQ 64-Bit-Werte (QWords). Wenn im Falle von MOVD der Quellenoperand (zweiter Operand) ein MMX-Register ist, so kopiert MOVD die niederwertigen 32 Bit (Bit 31 – 0) des Registers entweder in das Allzweckregister oder in den Speicheroperanden, der im Zieloperanden (erster Operand) angegeben wird. Ist der Quelloperand ein Allzweckregister oder ein Speicheroperand, so werden die niederwertigen 32 Bits des MMX-Registers (Bits 31 – 0), das im ersten Operanden übergeben wird, mit dem Inhalt des Allzweckregisters oder des Speicheroperanden belegt. Die höherwertigen Bits (Bit 63 – 32) werden auf »0« gesetzt.

Opcodes

#	B1	B2	B3	
1	0F	6E	/r	
2	0F	6E	/m	a16
3	0F	7E	/r	
4	0F	7E	/m	a16
5	0F	6F	/r	
6	0F	6F	/m	a16
7	0F	7F	/r	
8	0F	7F	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8, 20	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen

Mit MOVD können keine Werte von einem MMX-Register in ein anderes kopiert werden. Hierzu dient MOVQ, das den Datenaustausch zwischen MMX-Registern bewerkstelligen kann. MOVQ dagegen kann nicht mit Allzweckregistern bzw. Registerkombinationen verwendet werden.

Beschreibung

Seite 168.

PACKSSWB
PACKSSDW *MMX*

Funktion

Pack with Signed Saturation Words to Bytes bzw. *Doublewords to Words;* Packen mit Saturation von vorzeichenbehafteten Worten in vorzeichenbehaftete Bytes (PACKSSWB) oder von vorzeichenbehafteten Doppelworten in vorzeichenbehaftete Worte (PACKSSDW).

Verwendung

#	Parameter	Beispiel
1	mm, mm	PACKSSWB MM3, MM2
2	mm, m64	PACKSSWB MM0, QVar
3	mm, mm	PACKSSDW MM1, MM0
4	mm, m64	PACKSSDW MM7, QVar

Arbeitsweise

PACKSSWB »packt« im Saturation-Mode jeweils vier im Quell- und Zieloperanden stehende, vorzeichenbehaftete Worte. Für jedes Wort wird dabei zunächst geprüft, ob der Wert größer als +127 (=$7F) oder kleiner als –128 (= $80) ist. Ist dies der Fall, so wird die jeweilige Grenze ($7F bzw. $80) verwendet (= Saturation). Das Vorzeichen bleibt dabei erhalten. Andernfalls wird der tatsächliche Wert verwendet. Das »Packen« besteht darin, daß der Wertebereich eines Wortes auf diese Weise auf ein Byte abgebildet wird. Anschließend kombiniert PACKSSWB die acht Bytes und legt sie im Zieloperanden (zweiter Operand) ab. Dabei liegen die vier (gepackten und saturierten) Bytes aus dem Zieloperanden im »unteren« DWord des Zieloperanden, die vier Bytes aus dem Quelloperanden im »oberen«.

PACKSSDW arbeitet analog, indem es zwei Doppelworte aus dem Zieloperanden an den Grenzen +32.767 (= $7FFF) und –32.768 ($8000) saturiert und auf die niederwertigen zwei Worte im Zieloperanden ablegt. Die höherwertigen zwei Worte werden mit den gepackten und saturierten Doppelworten aus dem Quelloperanden belegt.

Opcodes

#	B1	B2	B3	
1	0F	63	/r	
2	0F	63	/m	a16
3	0F	68	/r	
4	0F	68	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen Zur Abbildung von vorzeichenbehafteten Worten auf vorzeichenlose Bytes dient der Befehl PACKUSWB.

Beschreibung Seite 162.

PACKUSWB *MMX*

Funktion *Pack with Unsigned Saturation Signed Words to Bytes*; Packen mit Saturation von vorzeichenbehafteten Worten in vorzeichenlose Bytes.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PACKUSWB MM3, MM2
2	mm, m64	PACKUSWB MM0, QVar

Arbeitsweise PACKUSWB »packt« im Saturation-Mode jeweils vier im Quell- und Zieloperanden stehende, vorzeichenbehaftete Worte. Für jedes Wort wird dabei zunächst geprüft, ob der Wert größer als +128 (=$FF) oder kleiner als 0 (= $00) ist. Ist dies der Fall, so wird die jeweilige Grenze ($FF bzw. $00) verwendet (= Saturation). Das Vorzeichen geht dabei verloren, da nur positive Werte Verwendung finden. Andernfalls wird der tatsächliche Wert verwendet. Das »Packen« besteht darin, daß der positive Wertebereich eines Wortes auf diese Weise auf ein vorzeichenloses Byte abgebildet wird. Anschließend kombiniert PACKUSWB die acht Bytes und legt sie im Zieloperanden (zweiter Operand) ab. Dabei liegen die vier (gepackten und saturierten) vorzeichenlosen Bytes aus den Zieloperanden im »unteren« DWord des Zieloperanden, die vier Bytes aus dem Quelloperanden im »oberen«.

Opcodes

#	B1	B2	B3	
1	0F	67	/r	
2	0F	67	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen Zur Abbildung von vorzeichenbehafteten Worten bzw. Doppelworten auf vorzeichenbehaftete Byte bzw. Worte dient der Befehl PACKSSWB bzw. PACKSSDW.

Beschreibung Seite 162.

PADDB
PADDW
PADDD *MMX*

Funktion *Packed Addition of Bytes / Words / Doublewords*; Addieren gepackter Bytes, Worte oder Doppelworte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PADDB MM0, MM1
2	mm, m64	PADDB MM7, QVar
3	mm, mm	PADDW MM5, MM6
4	mm, m64	PADDW MM3, QVar
5	mm, mm	PADDD MM4, MM1
6	mm, m64	PADDD MM2, QVar

Arbeitsweise Die Befehle addieren die acht Bytes / vier Worte / zwei Doppelworte, aus denen der Quelloperand (zweiter Operand) besteht, zu den entsprechenden Werten des Zieloperanden. Falls bei der byte-, wort- oder doppelwort-weisen Addition ein Überlauf erfolgt (in den Allzweckregistern würde das Carry-Bit gesetzt!), so wird dieser Überlauf sowohl ignoriert als auch nicht signalisiert. Insbesondere findet kein »Überlauf« in benachbarte Bytes, Worte oder Doppelworte statt.

Opcodes

#	B1	B2	B3	
1	0F	FC	/r	
2	0F	FC	/m	a16
3	0F	FD	/r	
4	0F	FD	/m	a16
5	0F	FE	/r	
6	0F	FE	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen

Die Additionen können wie bei den Additionen in den Allzweckregistern (»Integer-Addition«) mit vorzeichenlosen oder vorzeichenbehafteten Werten durchgeführt werden. Anders als bei der Integer-Addition werden dabei aber keine Flags gesetzt, die eine Interpretation des Ergebnisses erlaubten. Somit liegt es in der Verantwortung des Programmierers, die Wertebereiche zu prüfen und zu korrigieren – oder im Saturation-Mode zu arbeiten: PADDSB / PADDSW bzw. PADDUSB / PADDUSW.

Beschreibung

Seite 158.

PADDSB
PADDSW MMX

Funktion *Packed Addition with Signed Saturation of Bytes / Words*; Addieren und Saturieren gepackter, vorzeichenbehafteter Bytes oder Worte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PADDSB MM0, MM1
2	mm, m64	PADDSB MM3, QVar
3	mm, mm	PADDSW MM4, MM1
4	mm, m64	PADDSW MM2, QVar

Arbeitsweise Die Befehle addieren unter Berücksichtigung eines Vorzeichens die acht Bytes bzw. vier Worte, aus denen der Quelloperand (zweiter Operand) besteht, zu den entsprechenden Werten des Zieloperanden. Falls bei der byte- oder wort-weisen Addition ein Überlauf erfolgt, so wird das Ergebnis saturiert: Überschreitet es den Wert $7F bzw. $7FFF oder unterschreitet es den Wert von $80 (= -128) bzw. $8000 (= -32.768) für Bytes bzw. Worte, so wird es durch diese Grenzwerte ersetzt.

Opcodes

#	B1	B2	B3	
1	0F	EC	/r	
2	0F	EC	/m	a16
3	0F	ED	/r	
4	0F	ED	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen Additionen mit gepackten Bytes oder Worten können auch im Wrap-around-Mode durchgeführt werden. Hierzu dienen die Befehle PADDB bzw. PADDW.

Beschreibung Seite 158.

PADDUSB
PADDUSW *MMX*

Funktion *Packed Addition with Unsigned Saturation of Bytes / Words*; Addieren und Saturieren gepackter, vorzeichenloser Bytes oder Worte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PADDUSB MM0, MM1
2	mm, m64	PADDUSB MM3, QVar
3	mm, mm	PADDUSW MM4, MM1
4	mm, m64	PADDUSW MM2, QVar

Arbeitsweise Die Befehle addieren unter Vernachlässigung eines Vorzeichens die acht Bytes bzw. vier Worte, aus denen der Quelloperand (zweiter Operand) besteht, zu den entsprechenden Werten des Zieloperanden. Falls bei der byte- oder wort-weisen Addition ein Überlauf erfolgt, so wird das Ergebnis saturiert: Überschreitet es den Wert $FF (= 255) bzw. $FFFF (= 65.535) oder unterschreitet es den Wert von $00 bzw. $0000 für vorzeichenlose Bytes bzw. Worte, so wird es durch diese Grenzwerte ersetzt.

Opcodes

#	B1	B2	B3	
1	0F	DC	/r	
2	0F	DC	/m	a16
3	0F	CD	/r	
4	0F	DD	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen Additionen mit gepackten Bytes oder Worten können auch im Wrap-around-Mode durchgeführt werden. Hierzu dienen die Befehle PADDB bzw. PADDW.

Beschreibung Seite 158.

PAND MMX

Funktion Logische UND-Verknüpfung von Quadworten.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PAND MM0, MM1
2	mm, m64	PAND MM4, QVar

Arbeitsweise Der Befehl verknüpft alle 64 einzelnen Bits des Quadworts im Zieloperanden mit denen des Quelloperanden und legt das Ergebnis im Zieloperanden ab. Jedes Bit im Zieloperanden ist genau dann gesetzt, wenn die korrespondierenden Bits im Quell- und Zieloperanden beide gesetzt sind:

D = D AND S:
for $0 \le k \le 63$: if ($d_k = 1$ and $s_k = 1$) then $d_k = 1$ else $d_k = 0$;

Opcodes

#	B1	B2	B3	
1	0F	DB	/r	
2	0F	DB	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 167.

PANDN MMX

Funktion Logische UND-NICHT-Verknüpfung von Quadworten.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PANDN MM0, MM1
2	mm, m64	PANDN MM4, QVar

Arbeitsweise Der Befehl invertiert zunächst alle 64 einzelnen Bits des Quadworts im Zieloperanden, verknüpft sie dann durch eine AND-Verknüpfung mit denen des Quelloperanden und legt das Ergebnis im Zieloperanden ab. Jedes Bit im Zieloperanden ist genau dann gesetzt, wenn die korrespondierenden Bits im Quell- und invertierten Zieloperanden beide gesetzt sind:

D = D ANDN S:
for $0 \le k \le 63$: t_k = NOT d_k; if (t_k = 1 and s_k = 1) then d_k = 1 else d_k = 0;

Opcodes

#	B1	B2	B3	
1	0F	DB	/r	
2	0F	DB	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 167.

PCMPEQB
PCMPEQW
PCMPEQD *MMX*

Funktion

Packed Compare of Bytes / Words / Doublewords if Equal; Vergleich gepackter Bytes, Worte oder Doppelworte auf Gleichheit.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PCMPEQB MM0, MM1
2	mm, m64	PCMPEQB MM7, QVar
3	mm, mm	PCMPEQW MM5, MM6
4	mm, m64	PCMPEQW MM3, QVar
5	mm, mm	PCMPEQD MM4, MM1
6	mm, m64	PCMPEQD MM2, QVar

Arbeitsweise

Die Befehle vergleichen die acht Bytes / vier Worte / zwei Doppelworte, aus denen der Quelloperand (zweiter Operand) besteht, mit den entsprechenden Werten des Zieloperanden. Wenn die jeweiligen Wertepaare gleich sind, wird das entsprechende Byte/Wort/Doppelwort im Zieloperanden auf $FF/$FFFF/ $FFFFFFFF gesetzt, andernfalls auf 0. Die so erhaltene Maske kann bei den logischen Operationen mit gepackten Daten verwendet werden.

Opcodes

#	B1	B2	B3	
1	0F	74	/r	
2	0F	74	/m	a16
3	0F	75	/r	
4	0F	75	/m	a16
5	0F	76	/r	
6	0F	76	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung

Seite 161.

PCMPGTPB
PCMPGTPW
PCMPGTPD *MMX*

Funktion *Packed Compare of Bytes / Words / Doublewords if Greater*; Vergleich gepackter Bytes, Worte oder Doppelworte auf »größer als«.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PCMPGTB MM0, MM1
2	mm, m64	PCMPGTB MM7, QVar
3	mm, mm	PCMPGTW MM5, MM6
4	mm, m64	PCMPGTW MM3, QVar
5	mm, mm	PCMPGTD MM4, MM1
6	mm, m64	PCMPGTD MM2, QVar

Arbeitsweise Die Befehle vergleichen die acht Bytes / vier Worte/ zwei Doppelworte, aus denen der Quelloperand (zweiter Operand) besteht, mit den entsprechenden Werten des Zieloperanden. Wenn die jeweiligen Werte im Zieloperanden größer als die korrespondierenden im Quelloperanden sind, wird das entsprechende Byte/Wort/Doppelwort im Zieloperanden auf $FF/$FFFF/ $FFFFFFFF gesetzt, andernfalls auf 0. Die so erhaltene Maske kann bei den logischen Operationen mit gepackten Daten verwendet werden.

Opcodes

#	B1	B2	B3	
1	0F	64	/r	
2	0F	64	/m	a16
3	0F	65	/r	
4	0F	65	/m	a16
5	0F	66	/r	
6	0F	66	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 161.

PMADDWD MMX

Funktion *Packed Multiply and Add*; Multiplikation und Addition gepackter Worte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PMADDWD MM0, MM1
2	mm, m64	PMADDWD MM7, QVar

Arbeitsweise Der Befehl multipliziert die vier vorzeichenbehafteten Worte aus dem Zieloperanden mit den entsprechenden vorzeichenbehafteten Worten aus dem Quelloperanden. Das Ergebnis sind vier vorzeichenbehaftete Doppelworte. Im Anschluß werden die beiden niederwertigen Doppelworte addiert und in den niederwertigen Teil des Zielregisters abgelegt so wie die beiden höherwertigen Doppelworte, die dann im höherwertigen Teil des Zieloperanden abgelegt werden:

$D_{D0} = (D_{W0} * S_{W0}) + (D_{W1} * S_{W1})$; $D_{D1} = (D_{W2} * S_{W2}) + (D_{W3} * S_{W3})$;

Opcodes

#	B1	B2	B3	
1	0F	F5	/r	
2	0F	F5	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen Wenn alle vier Worte des Quell- und Zieloperanden den Wert $8000 haben, ist der Wert der beiden Ergebnis-Doppelworte $80000000.

Beschreibung Seite 160.

PMULHW
PMULLW *MMX*

Funktion *Packed Multiply High Word /Low Word*; Multiplikation gepackter Worte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PMULHW MM0, MM1
2	mm, m64	PMULHW MM7, QVar
3	mm, mm	PMULLW MM0, MM1
4	mm, m64	PMULLW MM7, QVar

Arbeitsweise Beide Befehle multiplizieren die einzelnen gepackten Worte im Zieloperanden mit den korrespondierenden im Quelloperanden. Das Ergebnis sind vier temporäre Doppelworte. PMULHW entnimmt nun den vier Doppelworten jeweils den höherwertigen Wortanteil und trägt ihn in die korrespondierenden Worte des Zieloperanden ein. PMULLW verfährt analog mit den niederwertigen Wortanteilen des Doppelwortes:

Beide Befehle:
$T_{D0} = D_{W0} * S_{W0}$; $T_{D1} = D_{W1} * S_{W1}$; $T_{D2} = D_{W2} * S_{W2}$; $T_{D3} = D_{W3} * S_{W3}$:

PMULLW:
$D_{W0} = LOW(T_{D0})$; $D_{W1} = LOW(T_{D1})$; $D_{W2} = LOW(T_{D2})$; $D_{W3} = LOW(T_{D3})$;

PMULHW:
$D_{W0} = HIGH(T_{D0})$; $D_{W1} = HIGH(T_{D1})$; $D_{W2} = HIGH(T_{D2})$; $D_{W3} = HIGH(T_{D3})$;

Opcodes

#	B1	B2	B3	
1	0F	E5	/r	
2	0F	E5	/m	a16
3	0F	D5	/r	
4	0F	D5	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6
#MF	Grund: wartende FPU-Exception				
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde				

Beschreibung Seite 159.

POR MMX

Funktion Logische ODER-Verknüpfung von Quadworten.

Verwendung

#	Parameter	Beispiel
1	mm, mm	POR MM0, MM1
2	mm, m64	POR MM4, QVar

Arbeitsweise Der Befehl verknüpft alle 64 einzelnen Bits des Quadwortes im Zieloperanden mit denen des Quelloperanden und legt das Ergebnis im Zieloperanden ab. Jedes Bit im Zieloperanden ist genau dann gelöscht, wenn die korrespondierenden Bits im Quell- und Zieloperanden beide gelöscht sind:

D = D OR S:
for $0 \le k \le 63$: if ($d_k = 0$ and $s_k = 0$) then $d_k = 0$ else $d_k = 1$;

Opcodes

#	B1	B2	B3	
1	0F	EB	/r	
2	0F	EB	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 167.

PSLLW
PSLLD
PSLLQ *MMX*

Funktion Logisches Verschieben der Bits eines Wertes nach links.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PSLLW MM0, MM1
2	mm, m64	PSLLW MM4, WVar
3	mm, i8	PSLLW MM7, $05
4	mm, mm	PSLLD MM0, MM1
5	mm, m64	PSLLD MM4, DVar
6	mm, i8	PSLLD MM7, $FF
7	mm, mm	PSLLQ MM0, MM1
8	mm, m64	PSLLQ MM4, QVar
9	mm, i8	PSLLQ MM7, $01

Arbeitsweise Der Befehl verschiebt die 16/32/64 Bits der vier Worte, zwei Doppelworte bzw. des Quadworts im Zieloperanden (ersten Operanden) um die im zweiten Operanden stehende Anzahl von Stellen »logisch« nach links. Logisch bedeutet dabei, daß nicht auf eventuelle Vorzeichen geachtet wird – alle 16/32/64 Bits eines Datums sind gleichwertig. Die frei werdenden Bitpositionen am unteren Ende des Datums werden mit »0« aufgefüllt, die »herausgeschobenen« Bits werden verworfen. Falls die Anzahl zu verschiebender Bitpositionen im zweiten Operanden größer als 15/31/63 ist, werden alle Bits des Datums gelöscht.

Opcodes

#	B1	B2	B3	B4	B5
1	0F	F1	/r		
2	0F	F1	/m	a16	
3	0F	71	/6	i8	
4	0F	F2	/r		
5	0F	F2	/m	a16	
6	0F	72	/6	i8	
7	0F	F3	/r		
8	0F	F3	/m	a16	
9	0F	73	/6	i8	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen Die Befehle betrachten die Bytes, Worte oder Doppelworte als unabhängig voneinander. Bitverschiebungen von einem Byte/Wort/Doppelwort in das andere erfolgen nicht!

Beschreibung Seite 166.

PSRAW
PSRAD *MMX*

Funktion Arithmetisches Verschieben der Bits eines Wertes nach rechts.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PSRAW MM0, MM1
2	mm, m64	PSRAW MM4, WVar
3	mm, i8	PSRAW MM7, $05
4	mm, mm	PSRAD MM0, MM1
5	mm, m64	PSRAD MM4, DVar
6	mm, i8	PSRAD MM7, $FF

Arbeitsweise Der Befehl verschiebt die 16 bzw. 32 Bits der vier Worte bzw. zwei Doppelworte im Zieloperanden (ersten Operanden) um die im zweiten Operanden stehende Anzahl von Stellen »arithmetisch« nach rechts. Arithmetisch bedeutet dabei, daß auf eventuelle Vorzeichen (Bit 15/31) geachtet wird. Die frei werdenden Bitpositionen am oberen Ende des Datums werden mit dem Wert des »Vorzeichens« aufgefüllt, die »herausgeschobenen« Bits werden verworfen. Falls die Anzahl zu verschiebender Bitpositionen im zweiten Operanden größer ist als 15/31, werden alle Bits des Datums gelöscht.

Opcodes

#	B1	B2	B3	B4	B5
1	0F	E1	/r		
2	0F	E1	/m	a16	
3	0F	71	/4	i8	
4	0F	E2	/r		
5	0F	E2	/m	a16	
6	0F	72	/4	i8	
7	0F	E3	/r		
8	0F	E3	/m	a16	
9	0F	73	/4	i8	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen

Die Befehle betrachten die Bytes, Worte oder Doppelworte als unabhängig voneinander. Bitverschiebungen von einem Byte/Wort/Doppelwort in das andere erfolgen nicht!

Die arithmetische Bitverschiebung nach links gibt es nicht explizit. Sie ist mit der logischen Bitverschiebung nach links identisch. Daher können anstelle von PSLAW/PSLAD die Befehle PSLLW/PSLLD verwendet werden.

Beschreibung

Seite 166.

PSRLW
PSRLD
PSRLQ *MMX*

Funktion Logisches Verschieben der Bits eines Wertes nach rechts.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PSRLW MM0, MM1
2	mm, m64	PSRLW MM4, WVar
3	mm, i8	PSRLW MM7, $05
4	mm, mm	PSRLD MM0, MM1
5	mm, m64	PSRLD MM4, DVar
6	mm, i8	PSRLD MM7, $FF
7	mm, mm	PSRLQ MM0, MM1
8	mm, m64	PSRLQ MM4, QVar
9	mm, i8	PSRLQ MM7, $01

Arbeitsweise Der Befehl verschiebt die 16/32/64 Bits der vier Worte, zwei Doppelworte bzw. des Quadwortes im Zieloperanden (ersten Operanden) um die im zweiten Operanden stehende Anzahl von Stellen »logisch« nach rechts. Logisch bedeutet dabei, daß nicht auf eventuelle Vorzeichen geachtet wird – alle 16/32/64 Bits eines Datums sind gleichwertig. Die frei werdenden Bitpositionen am oberen Ende des Datums werden mit »0« aufgefüllt, die »herausgeschobenen« Bits werden verworfen. Falls die Anzahl zu verschiebender Bitpositionen im zweiten Operanden größer als 15/31/63 ist, werden alle Bits des Datums gelöscht.

Opcodes

#	B1	B2	B3	B4	B5
1	0F	D1	/r		
2	0F	D1	/m	a16	
3	0F	71	/2	i8	
4	0F	D2	/r		
5	0F	D2	/m	a16	
6	0F	72	/2	i8	
7	0F	D3	/r		
8	0F	D3	/m	a16	
9	0F	73	/2	i8	

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Bemerkungen Die Befehle betrachten die Bytes, Worte oder Doppelworte als unabhängig voneinander. Bitverschiebungen von einem Byte/Wort/Doppelwort in das andere erfolgen nicht!

Beschreibung Seite 166.

PSUBB
PSUBW
PSUBD *MMX*

Funktion *Packed Subtraction of Bytes / Words / Doublewords*; Subtrahieren gepackter Bytes, Worte oder Doppelworte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PSUBB MM0, MM1
2	mm, m64	PSUBB MM7, QVar
3	mm, mm	PSUBW MM5, MM6
4	mm, m64	PSUBW MM3, QVar
5	mm, mm	PSUBD MM4, MM1
6	mm, m64	PSUBD MM2, QVar

Arbeitsweise Die Befehle addieren die acht Bytes / vier Worte / zwei Doppelworte, aus denen der Quelloperand (zweiter Operand) besteht, von den entsprechenden Werten des Zieloperanden. Falls bei der byte-, wort- oder doppelwortweisen Subtraktion ein Überlauf erfolgt (in den Allzweckregistern würde das Carry-Bit gesetzt!), so wird dieser Überlauf sowohl ignoriert als auch nicht signalisiert. Insbesondere findet kein »Überlauf« in benachbarte Bytes, Worte oder Doppelworte statt.

Opcodes

#	B1	B2	B3	
1	0F	F8	/r	
2	0F	F8	/m	a16
3	0F	F9	/r	
4	0F	F9	/m	a16
5	0F	FA	/r	
6	0F	FA	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 158.

PSUBSB
PSUBSW *MMX*

Funktion *Packed Subtraction with Signed Saturation of Bytes / Words*; Subtrahieren und Saturieren gepackter, vorzeichenbehafteter Bytes oder Worte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PSUBSB MM0, MM1
2	mm, m64	PSUBSB MM3, QVar
3	mm, mm	PSUBSW MM4, MM1
4	mm, m64	PSUBSW MM2, QVar

Arbeitsweise Die Befehle subtrahieren unter Berücksichtigung eines Vorzeichens die acht Bytes bzw. vier Worte, aus denen der Quelloperand (zweiter Operand) besteht, von den entsprechenden Werten des Zieloperanden. Falls bei der byte- oder wortweisen Subtraktion ein Überlauf erfolgt, so wird das Ergebnis saturiert: Überschreitet es den Wert $7F bzw. $7FFF oder unterschreitet es den Wert von $80 (= -128) bzw. $8000 (= -32.768) für Bytes bzw. Worte, so wird es durch diese Grenzwerte ersetzt.

Opcodes

#	B1	B2	B3	
1	0F	EC	/r	
2	0F	EC	/m	a16
3	0F	ED	/r	
4	0F	ED	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 158.

PSUBUSB
PSUBUSW *MMX*

Funktion *Packed Subtraction with Unsigned Saturation of Bytes / Words*; Subtrahieren und Saturieren gepackter, vorzeichenloser Bytes oder Worte.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PSUBUSB MM0, MM1
2	mm, m64	PSUBUSB MM3, QVar
3	mm, mm	PSUBUSW MM4, MM1
4	mm, m64	PSUBUSW MM2, QVar

Arbeitsweise Die Befehle subtrahieren unter Vernachlässigung eines Vorzeichens die acht Bytes bzw. vier Worte, aus denen der Quelloperand (zweiter Operand) besteht, von den entsprechenden Werten des Zieloperanden. Falls bei der byte- oder wort-weisen Subtraktion ein Überlauf erfolgt, so wird das Ergebnis saturiert: Überschreitet es den Wert $FF (= 255) bzw. $FFFF (= 65.535) oder unterschreitet es den Wert von $00 bzw. $0000 für vorzeichenlose Bytes bzw. Worte, so wird es durch diese Grenzwerte ersetzt.

Opcodes

#	B1	B2	B3	
1	0F	D8	/r	
2	0F	D8	/m	a16
3	0F	C9	/r	
4	0F	D9	/m	a16

Exceptions

	Protected Mode code	Protected Mode Grund	Virtual 8086 Mode code	Virtual 8086 Mode Grund	Real Mode Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 158.

PUNPCKHBW
PUNPCKHWD
PUNPCKHDQ *MMX*

Funktion *Unpack Packed High Bytes to Words / Words to Doublewords / Doublewords to Quadwords*; »Entpacken« von gepackten Bytes in Worte / von Worten in Doppelworte / von Doppelworten in Quadworte aus den höherwertigen Teilen der Operanden.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PUNPCKHBW MM0, MM1
2	mm, m64	PUNPCKHBW MM7, QVar
3	mm, mm	PUNPCKHWD MM5, MM6
4	mm, m64	PUNPCKHWD MM3, QVar
5	mm, mm	PUNPCKHDQ MM4, MM1
6	mm, m64	PUNPCKHDQ MM2, QVar

Arbeitsweise

Die Befehle »entpacken« Bytes in Worte, Worte in Doppelworte oder Doppelworte in Quadworte. Hierbei werden jeweils vier/zwei/ein Bytes/Worte/Doppelwort vom Zieloperanden verwendet und vier/zwei/ein Bytes/Worte/Doppelworte vom Quelloperanden, um im Zieloperanden zu acht/vier/zwei Bytes/Worte/Doppelworte zusammengesetzt zu werden.

Die Routinen für die höherwertigen Anteile an den Ausgangsoperanden (PUNPCKHBW, PUNPCKHWD und PUNPCKHDQ) verwenden dazu jeweils die vier/zwei/ein höherwertigen Bytes/Worte/Doppelworte von Quell- und Zieloperanden. Die Bytes/Worte/Doppelwort des Zieloperanden besetzen im Resultat die jeweils niederwertigen Anteile, die des Quelloperanden die höherwertigen Anteile der Worte/Doppelworte/ Quadworts (=»Interleaving«):

PUNPCKHBW
D_{W0}=shl(S_{B4},8)+D_{B4}; D_{W1}=shl(S_{B5},8)+D_{B5}; D_{W2}=shl(S_{B6},8)+D_{B6}; D_{W3}=shl(S_{B7},8)+D_{B7};

PUNPCKHWD
D_{D0}=shl(S_{W2},16)+D_{W2}, D_{D1}=shl(S_{W3},16)+D_{W3};

PUNPCKHDQ
D_{Q}=shl(S_{D1},32)+D_{D1};

Opcodes

#	B1	B2	B3	
1	0F	68	/r	
2	0F	68	/m	a16
3	0F	69	/r	
4	0F	69	/m	a16
5	0F	6A	/r	
6	0F	6A	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung

Seite 164.

PUNPCKLBW
PUNPCKLWD
PUNPCKLDQ *MMX*

Funktion

Unpack Packed Low Bytes to Words / Words to Doublewords / Doublewords to Quadwords; »Entpacken« von gepackten Bytes in Worte / von Worten in Doppelworte / von Doppelworten in Quadworte aus den niederwertigen Teilen der Operanden.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PUNPCKLBW MM0, MM1
2	mm, m64	PUNPCKLBW MM7, QVar
3	mm, mm	PUNPCKLWD MM5, MM6
4	mm, m64	PUNPCKLWD MM3, QVar
5	mm, mm	PUNPCKLDQ MM4, MM1
6	mm, m64	PUNPCKLDQ MM2, QVar

Arbeitsweise

Die Befehle »entpacken« Bytes in Worte, Worte in Doppelworte oder Doppelworte in Quadworte. Hierbei werden jeweils vier/zwei/ein Bytes/Worte/Doppelwort vom Zieloperanden verwendet und vier/zwei/ein Bytes/Worte/Doppelworte vom Quelloperanden, um im Zieloperanden zu acht/vier/zwei Bytes/Worte/Doppelworte zusammengesetzt zu werden.

Die Routinen für die niederwertigen Anteile an den Ausgangsoperanden (PUNPCKLBW, PUNPCKLWD und PUNPCKLDQ) verwenden dazu jeweils die vier/zwei/ein niederwertigen Bytes/Worte/Doppelworte vom Quell- und Zieloperanden. Die Bytes/Worte/Doppelwort des Zieloperanden besetzen im Resultat die jeweils niederwertigen Anteile, die des Quelloperanden die höherwertigen Anteile der Worte/Doppelworte/Quadworte (=»Interleaving«):

PUNPCKLBW
$D_{W0}=shl(S_{B0},8)+D_{B0}$; $D_{W1}=shl(S_{B1},8)+D_{B1}$; $D_{W2}=shl(S_{B2},8)+D_{B2}$; $D_{W3}=shl(S_{B3},8)+D_{B3}$;

PUNPCKLWD
$D_{D0}=shl(S_{W0},16)+D_{W0}$, $D_{D1}=shl(S_{W1},16)+D_{W1}$;

PUNPCKLDQ
$D_{Q}=shl(S_{D0},32)+D_{Do}$;

Opcodes

#	B1	B2	B3	
1	0F	60	/r	
2	0F	60	/m	a16
3	0F	61	/r	
4	0F	61	/m	a16
5	0F	62	/r	
6	0F	62	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 164

PXOR *MMX*

Funktion Logische Exclusiv-ODER-Verknüpfung von Quadworten.

Verwendung

#	Parameter	Beispiel
1	mm, mm	PAND MM0, MM1
2	mm, m64	PAND MM4, QVar

Arbeitsweise Der Befehl verknüpft alle 64 einzelnen Bits des Quadworts im Zieloperanden mit denen des Quelloperanden und legt das Ergebnis im Zieloperanden ab. Jedes Bit im Zieloperanden ist genau dann gesetzt, wenn die korrespondierenden Bits im Quell- und Zieloperanden sich voneinander unterscheiden:

D = D XOR S:
for $0 \le k \le 63$: if ($d_k = 1$ and $s_k = 1$) or ($d_k = 0$ and $s_k = 0$) then $d_k = 0$ else $d_k = 1$;

Opcodes

#	B1	B2	B3	
1	0F	DB	/r	
2	0F	DB	/m	a16

Exceptions

	Protected Mode		Virtual 8086 Mode		Real Mode
	code	Grund	code	Grund	Grund
#AC	0	1	0	1	./.
#GP	0	8	0	9	9
#NM	-	1	-	1	1
#PF	?	1	?	1	./.
#SS	0	1	0	-	-
#UD	-	6	-	6	6

#MF	Grund: wartende FPU-Exception
	Ein bereits bearbeiteter FPU-Befehl hat eine Exception ausgelöst, die noch nicht behandelt wurde

Beschreibung Seite 167.

38 Condition Codes

38.1 Condition Codes bei Vergleichsbefehlen

Bei den Vergleichsbefehlen des Coprozessors haben die Flags C3, C2 und C0 des Statusworts eine Bedeutung. C1 ist entweder 0 oder zeigt, wenn das Stack-Fault-Flag in Verbindung mit dem Invalid-Instruction-Flag gesetzt ist, einen *Stack-Overflow* oder *-Underflow* an.

C3	C2	C1	C0	Bedeutung
0	0	0	0	Operand 1 ist größer als Operand 2
0	0	0	1	Operand 1 ist kleiner als Operand 2
1	0	0	0	Operand 1 ist gleich groß wie Operand 2
1	1	*	1	Die Werte sind nicht vergleichbar

Diese Flags lassen sich so auswerten, daß die *Bedingten Sprungbefehle* des Prozessors verwendet werden können, um nach Vorzeichen zu unterscheiden. ACHTUNG! Obwohl Realzahlen vorzeichenbehaftet sind, müssen hier die korrespondierenden Befehle für den Vergleich vorzeichenloser Integer verwendet werden, da die Flagzuweisung durch die Bits des Condition Codes dies nicht anders ermöglicht!

```
          fstsw ax                ; < 80386: fstsw    [Temp]; mov   ax,[Temp]
          sahf                    ; condition code in Flags
          jp   NotComparable      ; C2 = parity flag
          je   Equal              ; C0 = carry flag, C3 = zero flag
          jb   Less               ; less = below in diesem Fall!
          jbe  LessOrEqual
          ja   Greater            ; greater = above in diesem Fall!
          jae  GreaterOrEqual
          :
Equal:    :
Less:     :
Greater: :
          :
```

38.2 Condition Codes bei Restbildung

Bei der Restbildung mit FPREM und FPREM1 hat nur das Flag C2 eine Bedeutung zur Feststellung der korrekten Arbeitsweise.

C3	C2	C1	C0	Bedeutung
	0			Die Restbildung konnte abgeschlossen werden
	1			Der Rest konnte nicht vollständig reduziert werden

Falls C2 gesetzt ist, so war die Restbildung nicht vollständig möglich, da die Anzahl der Subtraktionen, mit denen FPREM/FPREM1 den Divisionsrest bildet, die vorgegebene maximale Anzahl überschritten haben. Dennoch ist der gebildete Rest gültig, er kann aber weiter reduziert werden!

Die Flags C3, C1 und C0 enthalten die niedrigstwertigen drei Bits des Divisionsergebnisses, und zwar C0 Bit 2, C3 Bit 1 und C1 Bit 0. Der so zusammengesetzte Wert kann wiederum als Rest aufgefaßt werden, und zwar als Rest der Division des Division*sergebnisses* durch 8.

38.3 Condition Codes nach FXAM

Die Flags C3 bis C0 codieren Eigenschaften des TOS-Inhaltes. Es treten folgende Kombinationen auf:

C3	C2	C1	C0	Bedeutung
0	0	0	0	Positive, unnormalisierte Zahl*
0	0	0	1	+NAN
0	0	1	0	Negative, unnormalisierte Zahl*
0	0	1	1	-NAN
0	1	0	0	Positive, normalisierte Zahl
0	1	0	1	Positive Unendlichkeit
0	1	1	0	Negative, normalisierte Zahl
0	1	1	1	Negative Unendlichkeit
1	0	0	0	+ 0
1	0	0	1	TOS als *empty* markiert
1	0	1	0	- 0
1	0	1	1	TOS als *empty* markiert
1	1	0	0	Positive, denormalisierte Zahl
1	1	0	1	TOS als *empty* markiert*
1	1	1	0	Negative, denormalisierte Zahl
1	1	1	1	TOS als *empty* markiert*

* 80386/80486/Pentium benutzen diese Codes nicht, da von diesen Coprozessoren bzw. floating point units keine unnormalen Zahlen und nur zwei »Leer-Markierungen« verwendet werden.

39 Exceptions

39.1 Exception-Klassen

Exceptions werden in verschiedene Klassen eingeteilt, je nachdem, ob der Befehl, der zu der Exception geführt hat, nach Aufruf des Exception-Handlers wiederholt werden kann (und muß) oder nicht:

- Faults; Faults sind »Stolpersteine«, also Exceptions, die durch einen Exception-Handler korrigiert werden können. Faults können wie die Steine, über die man gestolpert ist, »beiseite geräumt« werden. Nach der Korrektur kann das Programm ohne Probleme und/oder Datenverlust fortgesetzt werden. Daher stellt der Prozessor bei solchen Exceptions den Zustand wieder her, der vor der Ausführung des Befehls herrschte, der zur Exception führte. Dem Handler wird als Rücksprungadresse die Adresse des Befehls übergeben, der zur Ausnahmesituation führte; dadurch kann nach Korrektur des Fehlers durch einen einfachen Rücksprung die Programmausführung an der Stelle fortgesetzt werden, die zur Exception führte – hoffentlich diesmal mit Erfolg.

  Beispiel:
  Division durch »0«. Dadurch, daß der Handler die Bedingung, die zur Division durch »0« führte, ändert, kann die fehlerhafte Division mit korrekten Zahlen wiederholt werden, ohne daß ein Schaden entstanden wäre.

- Traps; das sind »Fallen«, in die man getreten ist. Das bedeutet, daß sich etwas ereignet hat, das man nicht mehr korrigieren kann: Man ist bereits in die Falle getreten, wenn man sie bemerkt. Also wurde der Befehl bereits ausgeführt – und kann nicht wiederholt werden. Die Rücksprungadresse für den Exception-Handler zeigt somit auf die Adresse des Befehls, der dem Befehl unmittelbar folgt, der die Exception ausgelöst hat. Daß der »Schaden« bereits eingetreten ist, heißt nicht, daß das Programm nicht eventuell dennoch ohne Probleme weiterlaufen könnte. Es kommt auf den Fehler an, der dazu führte.

  Beispiel:
  Die Einzelschrittausführung von Programmen in einem Debugger. Dadurch, daß nach der Ausführung des Programms der Handler aufgerufen wird, kann dieser das Programm anhalten, bis der Benutzer es fortsetzt, und er ist in der Lage, die Informationen, die der Benutzer sehen will, darzustellen (Prozessorregisterinhalte, Speicheradresseninhalte etc.)

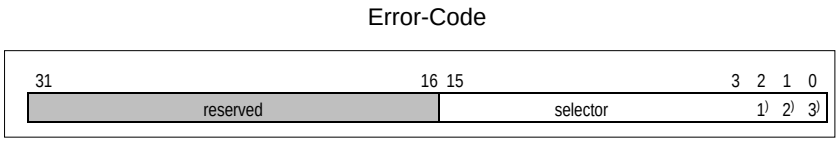
- Aborts; Programmabbrüche sind der schlimmste Fall von Exceptions. Je nach Grund für einen Abort kann nicht immer exakt die Quelle ermittelt und damit eine Adresse angegeben werden, an der eine Programmausführung wieder aufgenommen werden könnte. Aborts legen daher keine Rücksprungadresse auf den Stack.

  Beispiel:
  Fehler bei einem Hardwarebaustein.

Eine weitere Einteilung in Klassen ist möglich und im Hinblick auf ihr Verhalten erforderlich. So gibt es

- benigne (»gutartige«) Exceptions und Interrupts
- Contributory-Exceptions und
- Page-Faults.

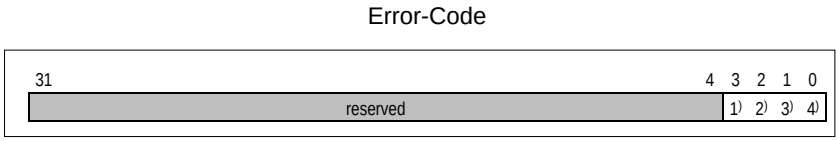
Manche Exceptions übergeben dem Handler einen Fehlercode. Dieser ist wie folgt aufgebaut:



1) =TI; 2) = IDT; 3) = EXT

Das External-Event-Bit EXT zeigt im gesetzten Zustand an, daß die Ursache für die Exception außerhalb des derzeit ablaufenden Programms ist. Das Descriptor-Location-Flag (IDT) signalisiert, daß der Selektor auf die IDT zeigt. Andernfalls zeigt er entweder auf die GDT (TI = 0) oder auf die aktuelle LDT (TI = 0). Der Table-Index TI hat keine Bedeutung, wenn IDT = 1 ist. Der Selektor wiederum zeigt auf den Descriptor in einer der Tabellen (IDT, GDT oder LDT), der mit dem Fehler verknüpft ist.

Bei der Page-Fault-Exception #PF hat der Error-Code einen etwas anderen Aufbau:

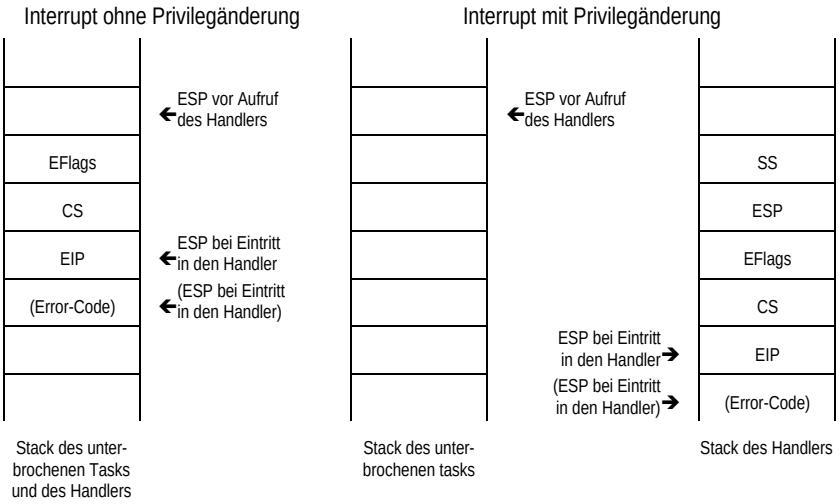


1) =RSVD; 2) = R/W; 3) = U/S; 4) = P

Der Error-Code übermittelt dem Handler Informationen über die Ursache der Exception. So gibt das Page-not-present-Flag P, Bit 0, an, ob der Page-Fault aufgrund einer nicht vorhandenen Page erfolgte (P = 0) oder aber aufgrund einer Zugriffsverletzung (auf Page-Ebene) oder der Nutzung eines reservierten Bits (ein einem Page-Directory-Entry oder Page-Table-Entry) erfolgte (P = 1). Im Falle einer Zugriffsverletzung geben die Bits 1, Read/write-Flag, und 2, User/supervisor-Flag, an, ob die Exception bei einem Lesezugriff (R/W = 0) oder Schreibzugriff (R/W = 1) im User-Mode (U/S = 1) oder im Supervisor-Mode (U/S = 0) erfolgte. (Zur Erinnerung: der Supervisor-Mode ist der Betriebsmodus, bei dem der CPL ≠ 3 ist. Im User-Mode ist CPL = 3!). Sollte dagegen die Nutzung von reservierten Bits der Grund der

Page-Fault sein, ist das Reserved-Flag, Bit 3, gelöscht (!). Ein gesetztes Bit 3 signalisiert, daß keine reservierten Bits benutzt worden sind, die #PF also aus einem der anderen genannten Gründe erfolgte.

Bei der Auslösung von Exceptions und Interrupts wird vom Prozessor eine Rücksprungadresse auf den Stack gelegt, die in der Regel auf den Befehl zeigt, der die Exception ausgelöst hat. Dies bedeutet, daß außer im Falle des Vorliegens einiger weniger Exception-Ursachen die Programmausführung an der betreffenden Stelle wieder aufgenommen werden kann, sobald der Handler die Ursache der Exception behoben hat. Wie (und wo) der Prozessor die Rücksprungadresse und einen eventuellen Error-Code auf den Stack legt, zeigt folgendes Schaubild:



39.2 CPU-Exceptions

Divide-Error-Exception (#DE)

Interruptvektor: 0

Beschreibung: Diese Exception zeigt an, daß der Divisor bei den Befehlen DIV und IDIV den Wert 0 hat oder das Ergebnis der Division nicht in dem spezifizierten Register dargestellt werden kann. So führt z.B. die Division von $FFFF (*word*) durch 2 zu einem Ergebnis ($7FFF), das nicht wie erforderlich in einem Byte dargestellt werden kann.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault, contributory

Error-Code: keiner

Rücksprung: CS:EIP auf dem Stack deuten auf die Anweisung, die die Exception verursacht hat.

Programmzustand: keine Veränderung, da die Exception vor der Ausführung des Befehls generiert wird.

Debug-Exception(#DB)

Interruptvektor: 1

Beschreibung: Diese Exception zeigt an, daß eine oder mehrere Bedingungen erfüllt sind, die im Rahmen des Debuggens eine Exception auslösen sollen. Ob es sich dabei um eine Fault oder Trap-Exception handelt, entscheidet die Ursache:

ein Breakpoint wurde gefunden	Fault
ein überwachtes Datum wurde verändert	Trap
eine Ein- oder Ausgabe erfolgte	Trap
General-Detection-Condition	Fault
Einzelschrittausführung	Trap
Task-Switch	Trap

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Trap oder Fault; die Unterscheidung erfolgt unter anderem durch die Analyse des Inhalts von Debug-Register #6 und anderen; benign

Error-Code: keiner; der Exception-Handler kann anhand der Debug-Register feststellen, wodurch die Exception entstanden ist.

Rücksprung: Fault:
CS:EIP auf dem Stack deuten auf die Instruktion, die die Exception ausgelöst hat.

Trap:
CS:EIP auf dem Stack deuten auf die Instruktion, die auf den die Exception auslösenden Befehl folgt.

Programmzustand: Weder bei Trap noch bei Fault-Exceptions ändert sich der Zustand des Programmes, da entweder der Befehl, der zur Exception führte, nicht ausgeführt wurde (Fault) oder die Veränderungen bereits zuvor stattgefunden haben (Trap).

NMI-Interrupt

Interruptvektor: 2

Beschreibung: NMIs werden durch externe Interruptquellen generiert, indem der dafür zuständige NMI#-Pin des Prozessors angesprochen wird.

Interruptquelle: Hardware (externe Bausteine; Aktivierung des NMI# Pins des Prozessors)

Exception-Klasse: Trap; benign

Error-Code: keiner

Rücksprung: CS:EIP auf dem Stack zeigen auf die nächste auszuführende Instruktion.

Programmzustand: Bevor der NMI-Interrupt generiert wird, wird in jedem Falle die Instruktion beendet, während der der Interrupt empfangen wird. Daher kann das Programm nach Rückkehr aus dem Interrupt-Handler ohne Probleme fortgeführt werden, vorausgesetzt, daß die Prozessorumgebung durch den Handler zunächst gesichert und vor Rückkehr wieder hergestellt wird.

Breakpoint-Exception (# BP)

Interruptvektor: 3

Beschreibung: Debugger tauschen üblicherweise an sogenannten Breakpoints das dort stehende Byte gegen den INT3-Befehl aus. Der Prozessor generiert dann beim Programmlauf an dieser Stelle die #BP-Exception. Der Handler kann dann entsprechend das Programm anhalten, Registerinhalte anzeigen usw., bevor er nach Ersetzen der INT3-Instruktion durch das ursprüngliche Byte die Programmausführung fortsetzt.

Interruptquelle: Software: INT3-Befehl

Exception-Klasse: Trap; benign

Error-Code: keiner

Rücksprung: CS:EIP auf dem Stack zeigen auf den Befehl, der der INT3-Instruktion unmittelbar folgt.

Programmzustand: Obwohl EIP auf den folgenden Befehl zeigt, verändert INT3 keinerlei Register- oder Speicherinhalte. Daher kann der Debugger das Programm fortsetzen, indem er die INT3-Instruktion durch den ursprünglich dort vorhandenen Befehl ersetzt und EIP auf dem Stack dekrementiert. Nach Rückkehr aus dem Debugger-Handler wird die Programmausführung mit dem ersetzten Befehl fortgeführt.

Hinweis: Bei Prozessoren ab dem 80386 werden Breakpoints am besten über die Debug-Register verwaltet und erstellt.

Der INT3-Befehl kann auf zwei Arten erzeugt werden: Durch Verwendung des 1-Byte-OpCodes $CC (= INT3) oder durch Aufruf des INT-Befehls mit einem Operanden vom Wert »3« (Zwei-Byte-Opcode $CD 03). Beide Befehle unterscheiden sich leicht voneinander. Auf Details soll hier aber nicht eingegangen werden. Alle mir bekannten Assembler übersetzen INT 3 in den Ein-Byte-Opcode $CC, der Zwei-Byte-Opcode muß über eingestreute Datenworte erzeugt werden.

Overflow-Exception (#OF)

Interruptvektor: 4

Beschreibung: Der INTO-Befehl prüft den Zustand des Overflow-Flags OF im EFlags-Register. Ist es gesetzt, wird die Exception ausgelöst, andernfalls nicht.

Interruptquelle: Software: INTO-Befehl

Exception-Klasse: Trap; benign

Error-Code: keiner

Rücksprung: CS:EIP zeigen auf den Befehl, der der INTO-Instruktion folgt.

Programmzustand: Da der INTO-Befehl keine Registerinhalte verändert, hat sich am Programmzustand durch die Exception nichts verändert. Der Handler kann die Programmausführung somit mit dem über CS:EIP angegebenen Befehl fortsetzen.

Hinweis: Der INTO-Befehl kann auf zwei Arten erzeugt werden: Durch Verwendung des 1-Byte-OpCodes $CE (= INTO) oder durch Aufruf des INT-Befehls mit einem Operanden vom Wert »4« (Zwei-Byte-Opcode $CD 04).

BOUND-Range-Exceeded-Exception (#BR)

Interruptvektor: 5

Beschreibung: Siehe die Referenz zu BOUND bei der Beschreibung der Befehle im Referenzteil.

Interruptquelle: Software (BOUND-Befehl)

Exception-Klasse: Fault; benign

Error-Code: keiner

Rücksprung: CS:EIP auf dem Stack zeigen auf den BOUND-Befehl, der die Exception ausgelöst hat.

Programmzustand: Die Operanden des BOUND-Befehls werden durch die Exception nicht verändert. Damit ändert sich nichts am Programmzustand.

Invalid-OpCode-Exception (#UD)

Interruptvektor: 6

Beschreibung: #UD wird ausgelöst, wenn der Prozessor einen ungültigen oder reservierten Befehl ausführen soll oder wenn der Operand eines Befehls nicht dem geforderten Typ entspricht. Darüber hinaus kann auch mit dem Befehl UD2 gezielt eine Exception ausgelöst werden. Ferner kann auch eine Befehlsfolge mit dem Präfix LOCK eine #DU auslösen, z.B. wenn der folgende Befehl nicht mit LOCK verwendet werden kann oder der Zieloperand keine Speicherstelle ist. Schließlich wird im Real-Mode oder im Virtual-8086 mode diese Exception ausgelöst, wenn versucht wird, LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW oder ARPL auszuführen, oder wenn RSM ausgeführt werden soll und der Prozessor sich nicht im SMM befindet.

Interruptquelle: Hardware (Prozessor) / Software (UD2-Befehl)

Exception-Klasse: Fault; benign

Error-Code: kein

Rücksprung: CS:EIP auf dem Stack zeigen auf den Befehl, der die Exception ausgelöst hat.

Programmzustand: Es erfolgt keine Zustandsänderung, da der die Exception auslösende Befehl nicht ausgeführt wird.

Hinweis: Die Ein-Byte-OpCodes D6 und F1 sind reservierte, undefinierte OpCodes, die jedoch keine #UD auslösen! Sie sollten dennoch nicht benutzt werden.

Device-Not-Available-Exception (#NM)

Interruptvektor: 7

Beschreibung: #NM wird ausgelöst, sobald der Prozessor einen FPU-Befehl auszuführen versucht und das Emulation-Flag EM oder das Task-Switched-Flag TS im Kontrollregister CR0 gesetzt sind, oder wenn der Prozessor ein WAIT/FWAIT auszuführen versucht, wenn das Task-Switched-Flag TS *und* das Monitor-Processor-Flag MP in CR0 gesetzt sind.

Das EM-Flag im Kontrollregister CR0 zeigt an, daß ein Prozessor *keine* FPU besitzt, FPU-Befehle also emuliert werden müssen. Der Versuch, bei gesetztem EM-Flag einen FPU-Befehl auszuführen, führt somit zur #NM-Exception, deren Handler dann dafür benutzt werden kann, die entsprechenden Funktionen zu emulieren.

Ein gesetztes TS-Flag zeigt an, daß seit dem Ausführen des letzten FPU-Befehls ein Task-Switch stattgefunden hat, die FPU-Umgebung aber nicht gesichert wurde. Auf diese Weise kann der Handler die ursprüngliche FPU-Umgebung sichern und die neue laden, bevor der aktuelle FPU-Befehl ausgeführt wird.

Das MP-Flag erweitert die Funktion des TS-Flags, das nur bei FPU-Befehlen wirksam ist, auf die Befehle WAIT/FWAIT. Auch in diesem Fall kann der Prozessor die alte FPU-Umgebung sichern und eine neue laden, wenn ein WAIT ausgeführt werden soll, das ja für die Koordination von FPU- mit CPU-Befehlen sorgt.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault; benign

Error-Code: keiner

Rücksprung: CS:EIP auf dem Stack zeigen auf den FPU-Befehl bzw. den WAIT/FWAIT-Befehl, der die Exception ausgelöst hat.

Programmzustand: Da die Instruktion, die zur Exception führt, nicht ausgeführt wird, ändert sich nichts am Zustand des Programms.

Hinweis: Das MP-Flag spielt lediglich bei 80286 / 80386 DX-Prozessoren eine Rolle, da nur bei diesen Prozessortypen CPU und FPU eigenständige Prozessoren sind und ein WAIT/FWAIT und die damit ggf. verbundene FPU-Umgebungssicherung nur im Falle der Existenz des Coprozessors Sinn macht. Ab dem 80486 DX befindet sich die FPU auf der CPU-Platine (außer beim 80486 SX; dem fehlt eine FPU, weshalb bei diesen Prozessoren auch MP grundsätzlich gelöscht sein sollte.). Daher sollte bei allen Prozessoren ab dem 80486 DX das MP-Flag grundsätzlich gesetzt sein.

Double-Fault-Exception (#DF)

Interruptvektor: 8

Beschreibung: #DF wird immer dann ausgelöst, wenn eine weitere Exception auftritt, während gerade eine andere Exception im Rahmen des Exception-Handlers behandelt wird. Zwar können üblicherweise mehrere Exceptions, die unmittelbar hintereinander auftreten, auf Halde gelegt und nacheinander abgearbeitet werden. Doch gibt es auch Ausnahmen, und in diesen Fällen wird eine #DF ausgelöst.

Interruptquelle: verschiedene

Exception-Klasse: Abort

Error-Code: Der Prozessor legt einen Error-Code mit »0« auf den Stack.

Rücksprung: Die Rücksprungadressen sind undefiniert, ein Rücksprung kann damit nicht durchgeführt werden.

Programmzustand: Der Programmzustand nach einem Double Fault ist undefiniert. Auch aus diesem Grund kann das Programm oder der Task nicht weiter ausgeführt werden. Die einzig sinnvolle Aktion nach einem Double Fault ist, so viele Informationen wie möglich über die aktuelle Umgebung zu sammeln, auszugeben und den Prozessor dann herunterzufahren.

Hinweis: Die folgende Tabelle zeigt, unter welchen Umständen ein #DF ausgelöst wird und wann die Exceptions hintereinander erfolgen (X):

		zweite Exception		
		benign	contributory	Page-Fault
erste Exception	benign	X	X	X
	contributory	X	#DF	X
	Page-Fault	X	#DF	#DF

Falls eine dritte Exception, egal welchen Typs, eintritt, während der Prozessor die #DF-Exception zu behandeln versucht, wird der Prozessor heruntergefahren.

Coprozessor-Segment-Overrun

Interruptvektor: 9

Beschreibung: 80386/80387-Kombinationen signalisieren mit dieser Exception, daß während des Übertragens des mittleren Teils eines Operanden für einen FPU-Befehl eine Page-oder Segment-Violation aufgetreten ist.

Interruptquelle: Hardware (Prozessor/Coprozessor)

Exception-Klasse: Abort; benign

Error-Code: kein

Rücksprung: Die Rücksprungadresse zeigt auf den Befehl, der die Exception auslöste.

Programmzustand: Der Zustand des Prozessors ist undefiniert. Das Programm bzw. der Task kann nicht wieder aufgenommen werden.

Hinweis: Diese Exception ist reserviert; sie wird vom Prozessor niemals erzeugt. Nicht verwenden. Bei allen Prozessortypen ab dem 80486 wird statt dieser Exception eine #GP ausgelöst.

Invalid-TSS-Exception (#TS)

Interruptvektor: 10 ($0A)

Beschreibung: Diese Exception wird ausgelöst, sobald bei einem Task-Switch Unregelmäßigkeiten auftreten. Das können sein:

- Ungültiges Task-State-Segment (TSS)
- Der Stack-Segment-Selector im TSS liegt außerhalb der Grenzen der GDT oder LDT, zeigt nicht auf ein beschreibbares Segment, besitzt im RPL-Feld nicht die notwendigen Privilegien oder zeigt nicht auf ein Segment mit dem richtigen DPL.
- Der Code-Segment-Selector im TSS liegt außerhalb der Grenzen der GDT oder LDT, zeigt nicht auf ein ausführbares Segment oder weist nicht die richtigen Privilegien auf.
- Ein Data-Segment-Selector im TSS liegt außerhalb der Grenzen der GDT oder LDT, zeigt nicht auf ein lesbares Segment oder weist nicht die notwendigen Privilegien auf.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault; contributory

Error-Code: Der Prozessor legt den Selektor für den Segmentdeskriptor im Error-Code ab, der die Exception verursachte. Der Error-Code wird dann auf den Stack gePUSHt.

Rücksprung: Wenn der Task-Switch bereits erfolgt ist, der zu dem Ausnahmezustand führte, zeigt CS:EIP auf dem Stack auf die erste Instruktion des neuen Tasks, andernfalls auf die Instruktion im alten Task, die den Task-Switch verursachte.

Programm-zustand: Der Zustand des Programmes hängt davon ab, zu welchem Zeitpunkt während des Task-Switches die Exception ausgelöst wurde. Ein Task-Switch ist keine »Hau-Ruck«-Maßnahme; vielmehr ist es ein zeitlich genau definierbarer Prozeß, in dem mehrere Aktionen erfolgen: Prüfung der Validität des TSS sowie des Selektors darauf, Laden der für die neue Task-Umgebung notwendigen Register, Umschalten auf die neue Umgebung und Laden der übrigen Register samt Validitätsprüfung. Daraus wird klar, daß z.B. die Validitätsprüfung des TSS vor einem eigentlichen switch erfolgt, die Validitätsprüfung des GS-Registers beispielsweise als für den eigentlichen Task-Switch unbedeutendes Register erst ganz am Ende.

Es gibt daher einen sog. Commit-to-New-Task-Point. Diesseits dieses Punktes ist der switch noch nicht erfolgt, es wurden keine Registerinhalte verändert. Eine Exception vor diesem Punkt zieht keine Veränderungen des Programmzustandes nach sich. Jenseits dieses Punktes dagegen hat sich der Prozessor zum Task-Switch »verpflichtet« (committed), da er z.B. aufgrund bislang unverdächtiger Überprüfungsergebnisse das Task-Register mit dem

neuen Selektor auf das neue TSS geladen hat. Egal, was passiert: er kann nun nicht zurück, er muß den Task-Switch endgültig vollziehen.

Das aber bedeutet, daß der Zustand des Programms weiterhin abhängig davon ist, welche tatsächliche Ursache die Exception hat. Der Prozessor lädt nämlich nach dem Point-of-no-Return zunächst die Segmentregister. Während des Ladevorgangs überprüft er die Inhalte auf Validität. Führt einer dieser Tests zu einer Exception, werden alle restlichen Register zwar ebenfalls geladen; aber es wird nicht mehr validiert. Daher ist zu dem Zeitpunkt, an dem der Exception-Handler die Verantwortung übernimmt, nicht klar, welche eigentliche Ursache die Exception hatte. Der Handler sollte daher, bevor er die Verantwortung zurückgibt, versuchen, die einzelnen Segmentregister-Inhalte im neuen TSS auf ihre Validität zu prüfen. Andernfalls resultiert nach der Rückkehr ggf. eine #GP-Exception, die äußerst schwer nachvollziehbar sein kann, da nicht vorhersehbar ist, wann auf das mit dem Exception auslösenden, fehlerhaften Selektor geladenen Segmentregister zugegriffen wird.

Segment-Not-Present (#NP)

Interruptvektor: 11 ($0B)

Beschreibung: Die #NP-Exception signalisiert, daß das Present-Bit in einem Segment- oder Gate-Deskriptor gelöscht ist und damit anzeigt, daß sich das spezifizierte Segment zur Zeit nicht im Speicher befindet. Gründe zum Auslösen einer #NP können sein:

- Während eines Task-Switches wird versucht, CS, DS, ES, FS oder GS zu laden. Beim Versuch, SS zu laden, wird im Falle der Abwesenheit eines Segments eine #SS ausgelöst.
- Der Versuch, mittels LLDT das LDTR anzusprechen. Während eines Task-Switches dagegen würde das Laden des LDTR zu einer #TS führen.
- Das Laden des TR, wobei im TSS-Deskriptor das Segment als not present markiert ist.
- Der Versuch, ein Gate oder TSS zu benutzen, das als not present markiert ist.

Das Betriebssystem verwendet das Present-Bit und die #NP-Exception üblicherweise im Rahmen der virtuellen Speicherverwaltung. Sobald der Handler das fehlende Segment nachgeladen hat, kann die Programmausführung ohne Fehler fortgeführt werden.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault; contributory

Error-Code: Der auf den Stack gelegte Error-Code beinhaltet den Segment-Selektor für den Deskriptor, der die #NP ausgelöst hat. Auf diese Weise kann der Handler feststellen, welches Segment nachzuladen ist.

Rücksprung: Üblicherweise zeigt die Rücksprungadresse auf die Adresse des Befehls, der die Exception verursachte. Falls die Exception ausgelöst wurde, während in einem neuen TSS die Einträge für die Segmentregister ausgelesen wurden, zeigen CS:EIP auf die erste Instruktion des neuen Tasks. Falls die Exception ausgelöst wurde, als auf einen Gate-Deskriptor zugegriffen wurde, zeigt CS:EIP auf den Befehl, der diesen Zugriff veranlaßte (z.B. auf ein CALL).

Programm-zustand: Falls die Exception beim Versuch ausgelöst wurde, CS, DS, ES, FG, GS oder das LDTR zu beladen, ändert sich der Zustand des Programms, da die Register nicht, wie erwartet, geladen werden. Die Wiederaufnahme des Programms ist dann einfach dadurch zu erreichen, daß das betreffende Segment nachgeladen wird und das Present-Bit im Deskriptor gesetzt wird.

Falls bei einem Zugriff auf ein Gate die Exception ausgelöst wurde, so heißt das, wie gesagt, gar nichts! Es hat sich am Zustand des Programms nichts geändert. Die Programmaufnahme kann einfach dadurch erfolgen, daß das Present-Bit gesetzt und zu der Rücksprungadresse verzweigt wird.

Hat sich die Exception während eines Task-Switchs ereignet, hängt der Zustand des Programms davon ab, zu welchem Zeitpunkt während des Task-Switches die Exception ausgelöst wurde. Ein Task-Switch ist keine »Hau-Ruck«-Maßnahme; vielmehr ist es ein zeitlich genau definierbarer Prozeß, in dem mehrere Aktionen erfolgen: Prüfung der Validität des TSS sowie des Selektors darauf, Laden der für die neue Task-Umgebung notwendigen Register, Umschalten auf die neue Umgebung und Laden der übrigen Register samt Validitätsprüfung. Daraus wird klar, daß z.B. die Validitätsprüfung des TSS vor einem eigentlichen switch erfolgt, die Validitätsprüfung des GS-Registers beispielsweise als für den eigentlichen Task-Switch unbedeutendes Register erst ganz am Ende.

Es gibt daher einen sog. Commit-to-New-Task-Point. Diesseits dieses Punktes ist der switch noch nicht erfolgt, es wurden keine Registerinhalte verändert. Eine Exception vor diesem Punkt zieht keine Veränderungen des Programmzustandes nach sich. Jenseits dieses Punktes dagegen hat sich der Prozessor zum Task-Switch »verpflichtet« (committed), da er z.B. aufgrund bislang unverdächtiger Überprüfungsergebnisse das Task-Register mit dem neuen Selektor auf das neue TSS geladen hat. Egal, was passiert: er kann nun nicht zurück, er muß den Task-Switch endgültig vollziehen.

Der Prozessor lädt nach dem Point-of-no-Return den neuen Task-State aus dem TSS, ohne jedoch die dabei üblichen Prüfungen auf Validität durchzuführen. Der Handler sollte sich daher nicht darauf verlassen, daß die Registerinhalte tatsächlich valide sind.

Hinweis: Gates korrespondieren nicht mit Segmenten. Deshalb zeigt eine #NP in diesem Falle auch nicht an, daß ein Segment fehlt. Vielmehr kann das Betriebssystem dieses Flag nutzen, um spezifische Dinge durchzuführen, wenn ein Gate angesprochen werden soll.

Stack-Fault-Exception (#SS)

Interruptvektor: 12 ($0C)

Beschreibung: Diese Exception zeigt an, daß der Stack in irgendeiner Weise korrupt ist. Als Gründe für die Exception kommen in Betracht:

- Eine Überschreitung des Limits des Stacks, wenn bei einer Operation der SS involviert ist und daher eine Stack-Überprüfung erfolgt. Das können folgende Befehle verursachen: POP, PUSH, CALL, RET, IRET, ENTER, LEAVE sowie alle Befehle, die implizit oder explizit auf das SS zugreifen, wie z.B. MOV-Befehle mit indizierten Adressen. ENTER erzeugt diese Exception ebenfalls, wenn kein ausreichender Platz für lokale Variablen mehr verfügbar ist.
- Das Present-Bit im Deskriptor für das Stacksegment ist gelöscht (not present). Die Prüfung dieses Flags kann im Rahmen eines Task-Switches erfolgen, bei CALLs an Ziele mit anderen Privilegstufen oder bei deren Rückkehr oder bei einer LSS- oder einer MOV- oder POP-Instruktion, bei der das SS-Register eine Rolle spielt.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault; contributory

Error-Code: Der Error-Code auf dem Stack ist 0, wenn eine »normale« Verletzung der Grenzen des Segments stattgefunden hat. Andernfalls enthält der Error-Code den Selektor, der auf den Deskriptor des Segments zeigt, das die Exception verursachte. In diesem Fall kann der Handler das Present-Flag überprüfen, um die Ursache für die Exception zu eruieren. Ist es gelöscht, so muß lediglich das Stacksegment nachgeladen werden, um den Fehler zu korrigieren. Andernfalls braucht lediglich das Limit des Stacks verändert zu werden.

Rücksprung: Die Rücksprungadresse auf dem Stack zeigt üblicherweise auf den Befehl, der die Exception verursacht hat. Falls aber die Exception im Rahmen eines Task-Switch erfolgte, zeigt CS:EIP auf den ersten Befehl des neuen Tasks.

Programm-zustand: Da die Anweisung, die die Exception verursachte, nicht ausgeführt wird, ändert sich üblicherweise auch nicht der Zustand des Programms. Daher kann das Programm nach Rückkehr aus dem Handler an der unterbrochenen Stelle wieder aufgenommen werden.

Hat sich die Exception während eines Task-Switchs ereignet, hängt der Zustand des Programms davon ab, zu welchem Zeitpunkt während des Task-Switchs die Exception ausgelöst wurde. Ein Task-Switch ist keine »Hau-Ruck«-Maßnahme; vielmehr ist es ein zeitlich genau definierbarer Prozeß, in dem mehrere Aktionen erfolgen: Prüfung der Validität des TSS sowie des Selektors darauf, Laden der für die neue Task-Umgebung notwendigen Register, Umschalten auf die neue Umgebung und Laden der übrigen Register samt Validitätsprüfung. Daraus wird klar, daß z.B. die Validitätsprüfung des TSS vor einem eigentlichen switch erfolgt, die Validitätsprüfung des GS-Registers beispielsweise als für den eigentlichen Task-Switch unbedeutendes Register erst ganz am Ende.

Es gibt daher einen sog. Commit-to-New-Task-Point. Diesseits dieses Punktes ist der switch noch nicht erfolgt, es wurden keine Registerinhalte verändert. Eine Exception vor diesem Punkt zieht keine Veränderungen des Programmzustandes nach sich. Jenseits dieses Punktes dagegen hat sich der Prozessor zum Task-Switch »verpflichtet« (committed), da er z.B. aufgrund bislang unverdächtiger Überprüfungsergebnisse das Task-Register mit dem neuen Selektor auf das neue TSS geladen hat. Egal, was passiert: er kann nun nicht zurück, er muß den Task-Switch endgültig vollziehen.

Der Prozessor lädt nach dem Point-of-no-Return den neuen Task-State aus dem TSS, ohne jedoch die dabei üblichen Prüfungen auf Validität durchzuführen. Der Handler sollte sich daher nicht darauf verlassen, daß die Registerinhalte tatsächlich valide sind.

General-Protection-Exception (#GP)

Interruptvektor: 13 ($0D)

Beschreibung: Diese Exception ist der »Lumpensammler«: Sie wird immer dann ausgelöst, wenn die Gründe für die Exception nicht zu anderen Exceptions »passen«. Als Ursachen für eine #GP gelten:

- Überschreitung von Segmentgrenzen beim Laden von Segmentregistern oder beim Zugriff auf Deskriptortabellen.
- Programmverzweigung in ein Segment, das nicht als »executable« markiert ist.

- Ein Schreibversuch in ein »Read-Only«-Datensegment oder das Auslesen eines »Execute-Only«-Codesegments.
- Das Laden des SS-Registers mit einem Selektor für ein Read-Only-Segment, ein Executable-Segment oder ein Nullsegment; das Laden von SS, DS, ES, FS oder GS mit Selektoren für Systemsegmente; das Laden von DS, ES, FS oder GS mit Selektoren für Execute-Only-Segmente oder das Laden von CS mit Selektoren auf Datensegmente oder ein Nullsegment.
- Der Zugriff auf Speicher, wenn DS, ES, FS oder GS einen Selektor mit dem Wert »0« beinhalten.
- Das Umschalten auf einen Busy-Task während eines Calls oder Jumps zu einem TSS oder während der Rückkehr zu einem TSS im Rahmen eines IRETs.
- Das Benutzen eines Segment-Selektors bei einem Task-Switch, der auf einen TSS-Deskriptor in der aktuellen LDT zeigt. (TSS-Deskriptoren können nur in der GDT angesiedelt werden!)
- Jegliche Verletzung der Privilegien.
- Das Überschreiten der maximalen Länge von 15 *Bytes* für Instruktionen (was nur passieren kann, wenn redundante Angaben zu Präfixen gemacht werden).
- Das Laden des Kontrollregisters CR0 mit einem gesetzten PG- und einem gelöschten PE-Flag (*paging* enabled, protection disabled; unmögliche Kombination) oder mit einem gesetzten NW- und einem gelöschten CD-Flag (not write-through enabled, cache disabled; ebenfalls unmöglich).
- Der Zugriff auf einen IDT-Eintrag, der nicht ein Interrupt- Trap- oder Task-Gate ist.
- Der Versuch, über ein Interrupt- oder Trap-Gate aus dem Virtual-8086 mode auf einen Interrupt-Handler zuzugreifen, wenn der DPL größer als 0 ist.
- Der Versuch, ein Bit im Kontrollregister CR4 zu setzen.
- Der Versuch, einen privilegierten Befehl auszuführen, wenn der CPL nicht 0 ist.
- Das Setzen eines reservierten Bits in einem MSR.
- Zugriff auf ein Gate, dessen Selektor »0« ist.
- Aufruf eines Interrupts, wenn der CPL größer als der DPL des betreffenden Interrupts, Traps oder Task-Gates ist.
- Wenn der Segment-Selektor in einem Call-, Interrupt- oder Trap-Gate nicht auf ein Codesegment zeigt.
- Wenn der Segment-Selektor im Operanden eines LLDT-Befehls lokal ist (TI-Flag gesetzt) oder nicht auf ein Segment vom Typ LDT zeigt oder wenn der Segment-Selektor im Operanden eines LTR-Befehls lokal ist oder auf ein nicht verfügbares TSS zeigt.

- Wenn der Zielselektor bei einem Call, Jump oder Return »0« ist.
- Wenn das PAE-Flag in Kontrollregister 4 (Physical Address Extension) gesetzt ist und der Prozessor irgendein reserviertes Bit in einem Page-Directory-Entry oder Page-Table-Entry als gesetzt vorfindet.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault; contributory

Error-Code: Der Prozessor legt einen Fehlercode auf den Stack, der entweder »0« ist, wenn bei dem die Exception verursachenden Befehl kein Selektor beteiligt war. Andernfalls wird der betroffene Selektor übergeben.

Rücksprung: Die Rücksprungadresse zeigt auf die Instruktion, die die Exception ausgelöst hat.

Programmzustand: Üblicherweise verursacht die Exception keine Veränderung des Programmzustandes, da der auslösende Befehl nicht ausgeführt wird. Daher kann ein Handler die Ursachen der Exception beseitigen und die Programmausführung fortsetzen.

Hat sich die Exception während eines Task-Switchs ereignet, hängt der Zustand des Programms davon ab, zu welchem Zeitpunkt während des Task-Switchs die Exception ausgelöst wurde. Ein Task-Switch ist keine »Hau-Ruck«-Maßnahme; vielmehr ist es ein zeitlich genau definierbarer Prozeß, in dem mehrere Aktionen erfolgen: Prüfung der Validität des TSS sowie des Selektors darauf, Laden der für die neue Task-Umgebung notwendigen Register, Umschalten auf die neue Umgebung und Laden der übrigen Register samt Validitätsprüfung. Daraus wird klar, daß z.B. die Validitätsprüfung des TSS vor einem eigentlichen switch erfolgt, die Validitätsprüfung des GS-Registers beispielsweise als für den eigentlichen Task-Switch unbedeutendes Register erst ganz am Ende.

Es gibt daher einen sog. Commit-to-New-Task-Point. Diesseits dieses Punktes ist der switch noch nicht erfolgt, es wurden keine Registerinhalte verändert. Eine Exception vor diesem Punkt zieht keine Veränderungen des Programmzustandes nach sich. Jenseits dieses Punktes dagegen hat sich der Prozessor zum Task-Switch »verpflichtet« (committed), da er z.B. aufgrund bislang unverdächtiger Überprüfungsergebnisse das Task-Register mit dem neuen Selektor auf das neue TSS geladen hat. Egal, was passiert: er kann nun nicht zurück, er muß den Task-Switch endgültig vollziehen.

Der Prozessor lädt nach dem Point-of-no-Return den neuen Task-State aus dem TSS, ohne jedoch die dabei üblichen Prüfungen auf Validität durchzuführen. Der Handler sollte sich daher nicht darauf verlassen, daß die Registerinhalte tatsächlich valide sind.

Page-Fault-Exception (#PF)

Interruptvektor: 14 ($0E)

Beschreibung: Diese Exception zeigt bei gesetztem PG-Flag in Kontrollregister CR0 an, daß

- das Present-Bit in einer Page-Directory-Entry oder Page-Table-Entry gelöscht ist und die referenzierte Page-Table oder Page daher physikalisch nicht verfügbar ist (was im Rahmen des Paging-Mechanismus kein Fehler ist, sondern lediglich veranlassen soll, daß der Handler die benötigte Table/Page nachlädt),
- daß der rufende Programmteil nicht die benötigten Rechte hat, auf die betreffende Seite zuzugreifen (weil er im User-Mode läuft, die Page aber Supervisor-Privilegien benötigt) oder
- daß entweder versucht wurde, von einem im User-Mode laufenden Programm auf eine als read-only markierte Page schreibend zuzugreifen (diese Prüfung wird nicht durchgeführt, wenn das laufende Programm im Supervisor-Mode läuft!) oder daß ein im Supervisor-Mode laufendes Programm auf Prozessoren ab dem 80486 versuchte, bei gesetztem WP-Flag in Kontrollregister CR0 schreibend auf eine Read-Only-Page im User-Mode zuzugreifen.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault; Page-Fault

Error-Code: Der Prozessor legt einen Error-Code im besonderen Format auf den Stack. Zusätzlich wird in Kontrollregister CR2 eine lineare 32-Bit-Adresse des Befehls abgelegt, der die Exception verursachte. Der Handler kann diese Adresse für die Berechnung der Page-Directory-Entry und Page-Table-Entry benutzen.

Rücksprung: Die Rücksprungadresse auf dem Stack zeigt üblicherweise auf den Befehl, der die Exception verursacht hat. Falls aber die Exception im Rahmen eines Task-Switchs erfolgte, zeigt CS:EIP auf den ersten Befehl des neuen Tasks.

Programmzustand: Üblicherweise verursacht die Exception keine Veränderung des Programmzustandes, da der auslösende Befehl nicht ausgeführt wird. Daher kann ein Handler die Ursachen der Exception beseitigen und die Programmausführung fortsetzen.

Hat sich die Exception während eines Task-Switchs ereignet, hängt der Zustand des Programms davon ab, zu welchem Zeitpunkt während des Task-Switchs die Exception ausgelöst wurde. Ein Task-Switch ist keine »Hau-Ruck«-Maßnahme; vielmehr ist es ein zeitlich genau definierbarer Prozeß, in dem mehrere Aktionen erfolgen: Prüfung der Validität des TSS sowie des

Selektors darauf, Laden der für die neue Task-Umgebung notwendigen Register, Umschalten auf die neue Umgebung und Laden der übrigen Register samt Validitätsprüfung. Daraus wird klar, daß z.B. die Validitätsprüfung des TSS vor einem eigentlichen switch erfolgt, die Validitätsprüfung des GS-Registers beispielsweise als für den eigentlichen Task-Switch unbedeutendes Register erst ganz am Ende.

Es gibt daher einen sog. Commit-to-New-Task-Point. Diesseits dieses Punktes ist der switch noch nicht erfolgt, es wurden keine Registerinhalte verändert. Eine Exception vor diesem Punkt zieht keine Veränderungen des Programmzustandes nach sich. Jenseits dieses Punktes dagegen hat sich der Prozessor zum Task-Switch »verpflichtet« (committed), da er z.B. aufgrund bislang unverdächtiger Überprüfungsergebnisse das Task-Register mit dem neuen Selektor auf das neue TSS geladen hat. Egal, was passiert: er kann nun nicht zurück, er muß den Task-Switch endgültig vollziehen.

Der Prozessor lädt nach dem Point-of-no-Return den neuen Task-State aus dem TSS, ohne jedoch die dabei üblichen Prüfungen auf Validität durchzuführen. Der Handler sollte sich daher nicht darauf verlassen, daß die Registerinhalte tatsächlich valide sind.

Reserved-Exception

Interruptvektor: 15 ($0F)

Hinweis: Diese Exception ist nicht dokumentiert.

Floating-Point-Error-Exception (#MF)

Interruptvektor: 16 ($10)

Beschreibung: Die #MF-Exception signalisiert, daß die FPU eine Exception bei einem FPU-Befehl entdeckt hat.

Interruptquelle: Hardware (FPU)

Exception-Klasse: Fault; benign

Error-Code: keiner; die FPU gibt die Ursache für die Exception in ihrem Statuswort an.

Rücksprung: Die Rücksprungadresse auf dem Stack zeigt auf die WAIT/FWAIT-Instruction oder den FPU-Befehl, der als nächstes ausgeführt worden wäre, wäre die Exception nicht ausgelöst worden. Dies ist nicht der Befehl, der zu der Exception führte; dieser kann der FPU-Umgebung entnommen werden (Instruction-Pointer, Data-Pointer und Opcode!).

Programmzustand: Da die FPU und die CPU weitgehend unabhängig voneinander agieren können und lediglich über WAITs/FWAITs synchronisiert werden, wartet die FPU damit, der CPU eine Exception zu signalisieren, bis entweder ein WAIT/FWAIT oder ein anderer FPU-Befehl ausgeführt werden soll. Dies kann aber dazu führen, daß die Adresse, auf die der Rücksprungzeiger auf dem Stack für den Exception-Handler verweist, eine ganz andere ist als die, an der der FPU-Befehl steht, der zur Exception führte. Daher hat mit großer Wahrscheinlichkeit eine Programmzustandsänderung stattgefunden.

Hinweis: Falls ein CPU-Befehl vom Ergebnis eines FPU-Befehls abhängig ist oder wenn mögliche FPU-Exceptions an definierten Programmstellen oder zu bestimmten Zeiten erfolgen sollen, können WAITs/FWAITs eingestreut werden, um mögliche FPU-Exceptions zu entdecken und zu behandeln.

Bemerkung: Einzelheiten siehe Kapitel FPU-Exceptions

39.3 FPU-Exceptions

Alignment-Check-Exception (#AC)

Interruptvektor: 17 ($11)

Beschreibung: Die #AC-Exception signalisiert, daß der Prozessor einen nicht ausgerichteten Operanden bei einem Alignment-Check gefunden hat. Diese Prüfungen finden nur an Datensegmenten statt, nicht an Code- oder Systemsegmenten. Als Regeln für die Ausrichtung von Daten gelten: Die Adressen von Worten müssen durch zwei ohne Rest teilbar sein, die von Doppelworten durch vier, von Singlereals durch vier, von Doublereals und Extendedreals (Tempreals) durch 8. Auch Zeiger können ausgerichtet werden: Selektoren und 32-Bit-far-Pointer müssen an ganzzahligen Vielfachen von zwei beginnen, 32-Bit-Pointer und 48-Bit-far-Pointer an Vielfachen von vier. Auch die Adressen, die in die Register GDTR, IDTR, LDTR und TR geladen werden, müssen selbst an Adressen liegen, die durch vier ohne Restbildung teilbar sind. Die Operanden für FSTENV/FLDENV und FSAVE/FRSTOR bzw. FXSAVE/FXRSTOR müssen wie die Adressen für Bitfelder, je nach Operandengröße, an Wort- oder Doppelwortgrenzen liegen (durch zwei oder vier vollständig teilbar sein).

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Fault; benign

Error-Code: Es wird der Wert »0« als Error-Code übergeben.

Rücksprung: Die Rücksprungadresse zeigt auf den Befehl, der die Exception ausgelöst hat.

Programmzustand: Da die Instruktion nicht ausgeführt wird, die zu einer #AC-Exception führte, ändert sich am Zustand des Programms nichts.

Hinweis: Ein Alignment-Check wird nur durchgeführt, wenn das AM-Flag in Kontrollregister CR0 sowie das AC-Flag in EFlags gesetzt sind. Ferner erfolgt er nur im Protected-Mode oder im Virtual-8086-Mode bei CPL = 3 (User-Mode).

Machine-Check-Exception (#MC)

Interruptvektor: 18 ($12)

Beschreibung: Der Prozessor hat einen internen Fehler entdeckt und signalisiert mit #MC einen nicht behebbaren Grund zum Programmabbruch.

Interruptquelle: Hardware (Prozessor)

Exception-Klasse: Abort; benign

Error-Code: keiner; Der Fehler wird in den Machine-check-Model-Specific-Registers übergeben.

Rücksprung: Wenn das (hier nicht weiter erklärte) EIPV-Flag im (ebenfalls nicht besprochenen) MCG-STATUS MSR (Model-Specific-Register) gesetzt ist, zeigt CS:EIP auf den Befehl, der den Machine-Check verursacht hat, der zur Exception führte. Andernfalls muß der durch CS:EIP referenzierte Zeiger nichts mit dem Fehler zu tun haben.

Programmzustand: Wenn das MCE-Flag im Kontrollregister CR4 gesetzt ist, verursacht eine #MC einen Programmabbruch. Das heißt, daß zwar das Machine-Check-MSR ausgelesen werden kann, das Programm aber nicht wieder gestartet wird. Ist das MCE-Flag gelöscht, führt eine #MC zum Herunterfahren des Prozessors.

Hinweis: Die #MC ist modellspezifisch! Die Implementation bei Pentium, Pentium Pro oder folgenden Prozessoren ist unterschiedlich und ggf. nicht kompatibel!

Reserved-Exceptions

Interruptvektor: 19 ($13) bis 31 ($1F)

Hinweis: Diese Exceptions sind reserviert und sollten nicht verwendet werden.

Interrupts

Interruptvektor: 32 ($20) bis 255 ($FF)

Beschreibung:

Interruptquelle: Software (INT nn-Befehl) oder Hardware (INTR# -Signal)

Exception-Klasse: keine

Error-Code: keiner

Rücksprung: Die Rücksprungadresse zeigt auf den Befehl nach dem INT nn-Befehl oder auf den Befehl nach der letzten ausgeführten Instruktion, bevor der Prozessor ein Signal am INTR#- Eingang erhalten hat.

Programmzustand: Der Prozessor schließt alle Aktionen eines in Ausführung befindlichen Befehls ab, bevor auf den INT nn-Befehl oder das INTR#-Signal reagiert wird. Damit findet durch den Interrupt keine Veränderung des Programmzustandes statt. Der Prozessor kann nach Verlassen des Interrupt-Handlers ohne Einschränkungen mit der eigentlichen Programmausführung fortfahren.

Da die FPU und die CPU weitgehend unabhängig voneinander agieren können und lediglich über WAITs/FWAITs synchronisiert werden, wartet die FPU damit, der CPU eine Exception zu signalisieren, bis entweder ein WAIT/FWAIT oder ein anderer FPU-Befehl ausgeführt werden soll. Dies kann aber dazu führen, daß die Adresse, auf die der Rücksprungzeiger auf dem Stack für den Exception-Handler verweist, eine ganz andere ist als die, an der der FPU-Befehl steht, der zur Exception führte. Daher hat mit großer Wahrscheinlichkeit eine Programmzustandsänderung stattgefunden.

Unter bestimmten Umständen kann es dazu kommen, daß eine Operation zwei oder mehrere Exceptions auslöst. Beispielsweise kann eine Operation zum einen ein unpräzises Ergebnis hervorrufen, andererseits aber auch zu einem Über- oder Unterlauf führen. Oder es könnte eine sNaN oder eine qNaN durch 0 dividiert werden. Im ersten Fall fallen somit eine #P und eine #O oder #U an, im zweiten eine #IA

(wegen der sNaN als Operand) und eine #Z. Der dritte Fall scheint klar: qNaNs lösen keine Exceptions aus, weshalb hier vermeintlicherweise nur ein #Z in Frage kommt. Das bedeutet, daß die anstehenden Exceptions nach bestimmten Regeln ausgelöst werden und werden müssen.

Diese sehen vor, daß zunächst nach #IAs bzw. #ISs gesucht wird, die folgende Ursachen haben: Stack-Overflow, Stack-Underflow, nicht unterstütztes Format, sNAN als Operand. Liegt eine solche Exception vor, so wird sie behandelt. Zusätzlich anstehende Exceptions werden dann unterdrückt. Wurde keine dieser #IAs gefunden, wird nach qNaNs als Operatoren gesucht. Obwohl sie keine Exceptions auslösen, »löschen« sie quasi anstehende Exceptions, verhindern deren Auslösung. Liegt auch keine qNaN vor, so wird nach anderen, bislang nicht genannten #IA oder #Z gesucht. Dann folgen #D, das nur dann eine Exception mit niedrigerer Priorität erlaubt, wenn es selbst maskiert wurde, und dann #O und #U in Verbindung mit #P. Schließlich wird noch nach einer #P gesucht.

Das bedeutet, daß bei den drei Beispielen von oben folgendes geschieht: Bei einem unpräzisen Ergebnis mit Über- oder Unterlauf wird #O bzw. #U in Verbindung mit #P ausgelöst, da keine vorangegangene Bedingung der Liste erfüllt ist. Die Division einer sNaN führt nur zu einer #IA, da mit Behandlung der #IAs weitere Exceptions unterdrückt werden. Und die qNaN liefert lediglich eine qNaN zurück, da auch sie weitere Exceptions, hier die #Z, unterdrückt.

#IA, #IS, #Z und #D sind Exceptions, die entdeckt werden können (und werden), bevor die Operation durchgeführt wird, die zur Exception führt. Das bedeutet, daß die Operation unterdrückt wird, sich also an der herrschenden FPU-Umgebung nichts ändert. #O, #U und #P dagegen können erst ausgelöst werden, nachdem die Operation durchgeführt wurde. Sie beeinflussen damit ggf. Registerinhalte sowie das Statuswort (z.B. TOS und Condition Code!).

Invalid-FPU-Operation-Exception (#IS, #IA)

Flag: IE (Bit 0 des *Statuswort*)

Maske: IM (Bit 0 des *Kontrollwort*)

Beschreibung: Ein gesetztes IE-Flag zeigt eine von zwei möglichen, generellen *Invalid-Operations an*:

- *Stack-Overflow* oder *Stack-Underflow* (#IS)
- *Invalid Arithmetic Operand* (#IA)

Falls also gleichzeitig zu IE auch SF gesetzt sein sollte, so hat ein Stack-*Fault* stattgefunden, andernfalls wurde ein für die arithmetische Operation fehlerhafter Operand gefunden.

Details: #IS, #IA

Bemerkungen: Die Flags der FPU sind »sticky«, d.h. sie werden nicht automatisch zurückgesetzt. Sollte daher eine #IA auftreten, das SF-Flag einer weiter zurückliegenden #IS jedoch noch nicht zurückgesetzt worden sein (durch den Exception-Handler), so ist es noch gesetzt und läßt nun fälschlicherweise auf eine weitere #IS schließen! Nach Prüfung der FPU-Flags sollten diese daher immer zurückgesetzt werden.

Stack-Overflow-or-Underflow-Exception (#IS)

Flag: SF (Bit 6 des Statusworts) gesetzt bei gesetztem IE-Flag.

Maske: keine

Beschreibung: Ein gesetztes SF-Flag zeigt einen von zwei Stack-Faults an:

- Stack-Overflow bei einem Versuch, ein Register zu beschreiben, das nicht als *empty* markiert ist
- Stack-Underflow bei einem Versuch, ein als *empty* markiertes Register auszulesen.

Die Unterscheidung, ob ein Stack-Overflow oder ein Stack-Underflow stattgefunden hat, ist über das Flag C1 des Condition Codes möglich. Bei einem Stack-Overflow ist es gesetzt, bei einem Stack-Underflow gelöscht.

Aktion bei Maskierung: Das IE- und das SF-Flag werden in jedem Fall gesetzt. Hat ein Stack-Overflow stattgefunden, so wird auch C1 im Condition Code gesetzt, andernfalls gelöscht. Die FPU erzeugt dann je nach Instruktion, die die Exception generiert hat, eine *real infinite*, *integer infinite* oder *BCD infinite* und überschreibt mit diesem Wert das Zielregister bzw. die Ziel-Speicherstelle.

Aktion bei fehlen der Maskierung: Es erfolgt das Setzen der IE- und SF-Flags sowie des Flags C1 im Condition Code, wenn ein Stack-Overflow stattgefunden hat. Andernfalls wird C1 gelöscht. Anschließend wird der Exception-Handler aufgerufen. Der TOS und die Operanden bleiben unverändert.

Bemerkungen: Der Begriff Stack-Overflow ergibt sich ursprünglich aus Situationen, in denen versucht wird, nach dem Pushen von acht Werten auf den FPU-Stack einen weiteren Wert zu pushen, der nun ein nicht leeres Register überschreiben würde. Analog kann, nachdem acht Werte aus den acht Registern gePOPpt wurden, diese also leer sind, kein weiterer Wert aus dem Stack geholt werden, es entsteht also ein Stack-Underflow.

Eine mögliche Aktion des Handlers könnte sein, einen sog. Virtual-Stack zu implementieren, der den realen Stack erweitert.

Invalid-Arithmetic-Operand-Exception (#IA)

Flag: SF (Bit 0 des *Statuswort*) gelöscht bei gesetztem IE-Flag.

Maske: keine

Beschreibung: Ein gelöschtes SF-Flag zeigt eine Fülle verschiedener Ursachen für einen arithmetischen Fehler an.

- Jede arithmetische Operation mit Operanden, die nicht das unterstütze Format aufweisen.
- Jede arithmetische Operation mit einer Signaling-NaN (sNaN).
- Die Einbeziehung von NaNs in Test- oder Vergleichsbefehle.
- Addition von Unendlichkeiten mit verschiedenen Vorzeichen oder Subtraktion von Unendlichkeiten mit gleichen Vorzeichen.
- Multiplikation von 0 mit ∞ oder ∞ mit 0.
- Division von ∞ mit ∞ oder 0 mit 0.
- Der Divisor bei FPREM oder FPREM1 ist 0 oder der Dividend ∞.
- FCOS, FTAN, FSIN und FSINCOS mit ∞ als Operanden.
- Negative Operanden bei FSQRT oder FYL2X oder ein negativerer Operand als –1 bei FYL2XP1 (was gleichbedeutend ist mit FYL2X<0!).
- Der Quelloperand von FBSTP ist ein leeres Register, enthält eine NaN, ∞ oder einen Wert, der nicht mit 18 Dezimalen dargestellt werden kann.
- Ein oder zwei leere Register bei FXCH.

Aktion bei Maskierung: Zunächst wird das IE-Flag gesetzt. (ACHTUNG! Das SF-Flag wird nicht explizit gelöscht!) In Abhängigkeit von dem oben genannten Quellen der Exception erfolgt dann

1. die Übergabe der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«)
2. die Übergabe einer qNaN an den Zielparameter
3. das Setzen von C0, C2 und C3 im Statuswort (»nicht vergleichbar«)
4. die Übergabe der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«)
5. die Übergabe der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«)
6. die Übergabe der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«)

7. die Übergabe der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«); C0 im Statuswort wird gelöscht
8. die Übergabe der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«). C0 im Statuswort wird gelöscht
9. die Übergabe der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«)
10. die Übergabe der *BCD-Integer-Indefinite* (Das 10. Byte der BCD, das das Vorzeichen enthält, besitzt den »Wert« $FF; ebenso die *Bytes* 9 und 8. Alle anderen *Bytes* können unterschiedliche Werte annehmen; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«)
11. das Belegen des/der leeren Register mit der *Real-Indefinite* (-1,10...0 E1...1; siehe »Darstellung der Zahlen im Coprozessor- und MMX-Format«), danach Austausch.

Aktion bei fehlen der Maskierung: Es wird das IE-Flag gesetzt und der Exception-Handler aufgerufen. Der Zeiger auf den TOS bleibt unverändert, ebenso wie die Inhalte der Register und Quelloperanden.

Bemerkungen: Bei einer #IA verändert die FPU das SF-Flag **nicht**! Es ist ein sog. Sticky-Flag und bleibt gesetzt, bis es explizit gelöscht wird. Daher kann es zu falschen Interpretationen der Exception-Ursache kommen, falls der Exception-Handler bei einer vorangegangenen #IS das SF-Flag nicht gelöscht hat.

Denormalized-Operand-Exception (#D)

Flag: DE (Bit 1 des *Statusworts*)

Maske: DM (Bit 1 des *Kontrollworts)*

Beschreibung: Eine #D-Exception wird ausgelöst, wenn

- versucht wird, eine arithmetische Operation mit denormalisierten Operanden durchzuführen oder
- versucht wird, eine denormalisierte SINGLEREAL oder DOUBLEREAL in ein Register einzulesen (denormalisierte TEMPREALs erzeugen diese Exception in diesem Fall nicht!).

Aktion bei Maskierung: Bei arithmetischen Operationen mit denormalisierten Operanden erfolgt außer dem Setzen des DE-Flags gar nichts: Die Operation wird ausgeführt. Die Ergebnisse von Berechnungen sind im Gegenteil sogar mindestens genauso gut, wenn nicht besser, als hätte man eine denormalisierte Zahl durch die kleinste normale Zahl oder gar »0« ersetzt. Vielmehr profitieren viele Berechnungen davon, daß im TEMPRealformat die zusätzlichen Möglichkeiten sehr kleiner Zahlen existieren, weshalb häufig eine Rechenkette mit denormalisierten Operanden zu Ende geführt wird und anschließend eine Betrachtung der Genauigkeit des Ergebnisses durchgeführt wird.

Beim Laden einer denormalisierten SINGLEREAL oder DOUBLEREAL wird ebenfalls das DE-Flag gesetzt, dann die Zahl ins TEMPRealformat überführt und dabei normalisiert. (Da die TEMPREAL eine höhere Genauigkeit mit mehr Nachkommastellen besitzt, kann jede denormalisierte SINGLEREAL oder DOUBLEREAL in eine normalisierte TEMPREAL überführt werden!)

Aktion bei fehlender Maskierung: Es wird das DE-Flag gesetzt und der Exception-Handler aufgerufen. Der Zeiger auf den TOS bleibt unverändert, ebenso wie die Inhalte der Register und Quelloperanden.

Bemerkungen: Es kann sinnvoll sein, denormalisierte Operanden von einer Berechnung auszuschließen, wenn der Verlust von signifikanten Stellen durch die Denormalisierung zu einem Verlust an Genauigkeit führt. Der Ausschluß von denormalisierten Operanden kann durch den Exception-Handler erfolgen, wenn eine #D nicht maskiert wird!

Division-By-Zero-Exception (#Z)

Flag: ZE (Bit 2 des *Statusworts*)

Maske: ZM (Bit 2 des *Kontrollworts*)

Beschreibung: Alle Befehle, die eine Division durchführen (FDIV, FDIVP, FDIVR, FIDVRP, FIDIV, FIDIVR), aber auch diejenigen, bei denen lediglich intern eine Division erfolgt (FYL2X, FXTRACT) lösen diese Exception aus, wenn versucht wird, einen Nicht-Null-Operanden durch 0 zu dividieren.

Aktion bei Maskierung: Die FPU setzt zunächst das ZE-Flag. Bei den Divisionsbefehlen wird dann eine vorzeichenbehaftete Infinite (∞) zurückgegeben. Das Vorzeichen wird durch ein exklusives OR der Vorzeichen der Operanden ermittelt. So ist das Vorzeichen der Infiniten negativ, wenn beide Vorzeichen unterschiedlich sind, und positiv, wenn beide Vorzeichen gleich sind. (Beachten Sie bitte, daß auch der Wert »0« ein Vorzeichen besitzen kann!).

Bei der FYL2X-Instruktion wird ebenfalls eine Infinite (∞) zurückgegeben, deren Vorzeichen das entgegengesetzte Vorzeichen des Operanden besitzt, der nicht 0 ist.

Bei der FXTRACT-Anweisung wird ST(1) mit (-∞) belegt und ST(0) mit dem Wert »0«, der das gleiche Vorzeichen wie der Quelloperand besitzt.

Aktion bei fehlender Maskierung: Es wird das ZE-Flag gesetzt und der Exception-Handler aufgerufen. Der Zeiger auf den TOS bleibt unverändert, ebenso wie die Inhalte der Register und Quelloperanden.

Numeric-Overflow-Exception (#O)

Flag: OE (Bit 3 des *Statusworts*)

Maske: OM (Bit 3 des *Kontrollworts*)

Beschreibung: Eine #O-Exception wird immer dann ausgelöst, wenn das gerundete Ergebnis einer Operation den maximal darstellbaren Bereich des Zieloperanden überschreitet. So hat z.B. eine TEMPREAL einen maximalen Bereich von $-1.11..11 \cdot 2^{16383}$ bis $+1.11..11 \cdot 2^{16383}$ ($\cong \pm 1{,}18 \cdot 10^{4932}$). Führt nun eine Berechnung zu der »nächsthöheren« Zahl $\pm 1.00..00 \cdot 2^{16384}$, so ist diese nicht mehr darstellbar, und eine #O-Exception wird ausgelöst.

Dies kann immer dann erfolgen, wenn eine arithmetische Operation erfolgte oder aber eine TEMPREAL als SINGLEREAL oder DOUBLEREAL abgespeichert werden soll.

Aktion bei Maskierung: Das Ergebnis der Aktion bei maskierten Exceptions hängt davon ab, welcher Rundungsmodus eingestellt ist. In jedem Falle wird das OE-Flag gesetzt.

- Rundung zur nächsten oder ganzen Zahl (Bit 11 und 10 im Control-Register = 00): hat das Ergebnis ein positives Vorzeichen, so wird +∞ zurückgegeben, bei negativem Vorzeichen -∞.
- Rundung in Richtung »minus unendlich« (01): bei positivem Vorzeichen des Ergebnisses wird die größte darstellbare positive Zahl zurückgegeben, bei negativem Vorzeichen -∞.

- Rundung in Richtung »plus unendlich« (10): bei positivem Vorzeichen des Ergebnisses wird +∞ zurückgegeben, bei negativem Vorzeichen die größte darstellbare negative Zahl.
- Abschneiden der Nachkommastellen (11): Bei positivem Vorzeichen des Ergebnisses wird die größte positive, bei negativem Vorzeichen die größte negative Zahl zurückgegeben, die im Operanden darstellbar ist.

Aktion bei fehlender Maskierung: Falls der Zieloperand eine Speicherstelle ist, wird das OE-Flag gesetzt und der Exception-Handler aufgerufen. Der TOS und die Quelloperanden bleiben unverändert.

Falls der Zieloperand ein FPU-Register ist, wird das Ergebnis gerundet und der Exponent durch $3 \cdot 2^{13} = 24.576$ dividiert (skaliert) und mit der Mantisse im Zieloperanden gespeichert. Anschließend wird C1 (in dieser Situation »Round-up«-Bit genannt) im Statuswort gesetzt, wenn die Rundung »nach oben« erfolgte, und gelöscht, wenn sie »nach unten« erfolgte. Anschließend wird das OE-Flag gesetzt und der Exception-Handler aufgerufen.

Bei der FSCALE-Instruktion kann es zu einem massiven Overflow kommen, der selbst mit der Skalierung noch die darstellbaren Bereiche überschreitet. In diesem Fall wird eine Infinite im Zieloperanden abgelegt, deren Vorzeichen den Regeln entspricht.

Bemerkungen: #O-Exceptions werden nicht ausgelöst, wenn Daten im Integerformat oder als BCD abgespeichert werden, selbst wenn der Wertebereich überschritten wird. In diesem Fall erfolgt die Auslösung einer #IA-Exception!

Die Skalierung mit dem Faktor $3 \cdot 2^{13}$ im Falle eines unmaskierten OM-Flags als Antwort auf die Exception soll den Exponenten so weit wie möglich »in die Mitte« des Exponentenbereiches bringen. Auf diese Weise können folgende Berechnungen, so sie alle mit dem Faktor skaliert werden, weiterhin durchgeführt werden – wobei das Risiko, daß es zu einem weiteren Überlauf kommt, entsprechend geringer ist.

Numeric-Underflow-Exception (#U)

Flag: UE (Bit 4 des *Statusworts*)

Maske: UM (Bit 4 des *Kontrollworts*)

Beschreibung: Eine #U-Exception wird immer dann ausgelöst, wenn das gerundete Ergebnis einer Operation den minimal darstellbaren Bereich des Zieloperanden unterschreitet. Hierbei ist zu beachten, daß als minimaler Wert der Wert bezeichnet wird, der noch ohne Denormalisierung darstellbar ist! So

hat z.B. eine TEMPREAL einen minimalen Grenzwert von $\pm 1.00..00 \cdot 2^{-16382}$ ($\cong \pm 3{,}37 \cdot 10^{-4931}$). Führt nun eine Berechnung zu der »nächstniedrigeren« Zahl $\pm 1.11..11 \cdot 2^{-16383}$, so ist diese nicht mehr darstellbar, und eine #U-Exception wird ausgelöst.

Dies kann immer dann erfolgen, wenn eine arithmetische Operation erfolgte oder aber eine TEMPREAL als SINGLEREAL oder DOUBLEREAL abgespeichert werden soll.

Aktion bei Maskierung:

Die nicht mehr korrekt darstellbare Zahl wird denormalisiert. Ist das Ergebnis der Denormalisierung korrekt, was bedeutet, daß die Zahl als Denormale dargestellt werden kann, wird es im Zieloperanden abgelegt. Das UE-Flag wird *nicht* gesetzt!

Andernfalls kann die Zahl so klein geworden sein, daß sie selbst durch Denormalisierung nicht mehr korrekt darstellbar ist. In diesem Fall wird das UE-Flag gesetzt und eine #P-Exception ausgelöst.

Aktion bei fehlender Maskierung:

Falls der Zieloperand eine Speicherstelle ist, wird das UE-Flag gesetzt und der Exception-Handler aufgerufen. Der TOS und die Quelloperanden bleiben unverändert.

Falls der Zieloperand ein FPU-Register ist, wird das Ergebnis gerundet und der Exponent mit $3 \cdot 2^{13} = 24.576$ multipliziert (skaliert) und mit der Mantisse im Zieloperanden gespeichert. Anschließend wird C1 (in dieser Situation »Round-up«-Bit genannt) im Statuswort gesetzt, wenn die Rundung »nach oben« erfolgte, und gelöscht, wenn sie »nach unten« erfolgte. Anschließend wird das UE-Flag gesetzt und der Exception-Handler aufgerufen.

Bei der FSCALE-Instruktion kann es zu einem massiven Underflow kommen, der selbst mit der Skalierung noch die darstellbaren Bereiche unterschreitet. In diesem Fall wird der Wert »0« im Zieloperanden abgelegt, dessen Vorzeichen den Regeln entspricht.

Bemerkungen:

Wenn die Behandlung von Underflow-Exceptions maskiert ist, so wird eine Exception nur in dem Fall ausgelöst, in dem eine Zahl nicht denormalisiert werden kann! Sobald sie als Denormale darstellbar ist, wird weder ein Flag gesetzt noch ein Exception-Handler aufgerufen.

Die Skalierung mit dem Faktor $3 \cdot 2^{13}$ im Falle eines unmaskierten UM-Flags als Antwort auf die Exception soll den Exponenten so weit wie möglich »in die Mitte« des Exponentenbereichs bringen. Auf diese Weise können folgende Berechnungen, so sie alle mit dem Faktor skaliert werden, weiterhin durchgeführt werden – wobei das Risiko, daß es zu einem weiteren Unterlauf kommt, entsprechend geringer ist.

Precision-Exception (#P)

Flag: PE (Bit 5 des *Statusworts*)

Maske: PM (Bit 5 des *Kontrollworts*)

Beschreibung: Die Precision-Exception, auch Inexact-Result-Exception genannt, wird immer dann ausgelöst, wenn das Ergebnis einer Operation nicht korrekt im Zielformat dargestellt werden kann. So kann beispielsweise der Bruch 1/3 binär nicht korrekt dargestellt werden.

Aktion bei Maskierung: Zunächst prüft die FPU, ob der Grund für die Precision-Exception ein Overflow oder Underflow war. Ist dies nicht der Fall, so wird das PE-Flag gesetzt, das Ergebnis gerundet und im Zieloperanden abgelegt. Die im Kontrollwort in den Bits 11 und 10 vorgegebene Rundungsart kommt dabei zum Einsatz. Wurde »nach oben« gerundet, so wird C1 im Statuswort (in diesem Fall »Round-up«-Bit genannt) gesetzt, andernfalls gelöscht. Wurde »nach unten« gerundet, heißt das, daß die letzten Ziffern des Nachkommaanteils so abgeschnitten wurden, daß das Ergebnis darstellbar wird.

Hat eine Precision-Exception in Verbindung mit einem Overflow oder Underflow stattgefunden, so werden das PE-Flag und das OE- oder UE-Flag gesetzt. Das Ergebnis wird dann, wie unter #O oder #U beschrieben, in Abhängigkeit von der Maskierung dieser Exceptions bearbeitet.

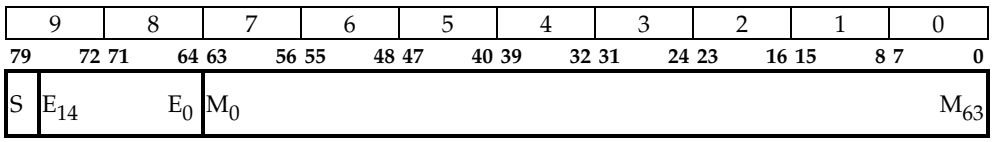
Aktion bei fehlender Maskierung: Auch in diesem Fall prüft die FPU zunächst, ob der Grund für die Precision-Exception ein Overflow oder Underflow war. Ist dies nicht der Fall, so wird wie im Falle einer Maskierung weitergemacht. Abschließend wird der Exception-Handler für #P-Exceptions aufgerufen.

Hat dagegen eine Precision-Exception in Verbindung mit einem Overflow oder Underflow stattgefunden, so werden das PE-Flag und das OE- oder UE-Flag gesetzt. Das Ergebnis wird dann, wie unter #O oder #U beschrieben, in Abhängigkeit von der Maskierung dieser Exceptions bearbeitet. Abschließend wird auch in diesem Fall der Exception-Handler für #P-Exceptions aufgerufen.

Bemerkungen: Diese Exception hat häufig anzutreffende Ursachen; die nicht vollständig darstellbaren Ergebnisse von Brüchen sind nur ein Beispiel. So können aufgrund ihrer Natur alle transzendentalen Funktionen (FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, FYL2XP1) keine vollständig darstellbaren Ergebnisse erzeugen.

40 Darstellung von Zahlen im Coprozessor- und MMX-Format

40.1 Internes Format und TEMPREAL-Format (FPU)



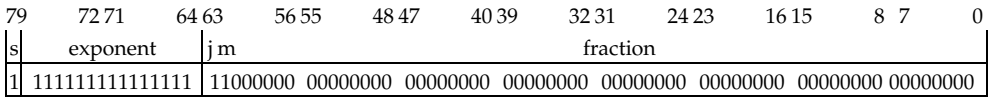
Eine TEMPREAL besitzt einen Mantissenteil mit 64 Bits, einen Exponenten mit 15 Bits sowie ein Vorzeichen mit 1 Bit. Das Vorzeichen *S* steht dabei an Bitposition 79 (Bit 7 des höchstwertigen Bytes der 10-Byte-TEMPREAL) und gilt nur für die Mantisse. Die Mantisse erstreckt sich von Bitposition 63 bis 0, wobei sie umgekehrt dargestellt wird: Das niedrigstwertige Bit der Manitsse steht an Bitposition 63, ihr höchstwertiges an Position 0. Der Exponent füllt den verbleibenden Raum zwischen Position 78 und 64. Er wird »richtig herum« codiert, also das niedrigstwertige Bit des Exponenten an Position 64, sein höchstwertiges an Position 78.

Die Realzahl läßt sich wie folgt berechnen:

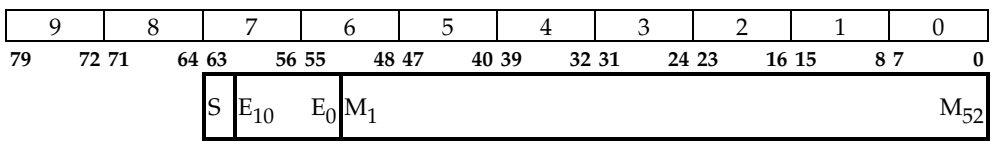
$R = (-1)^S \cdot M \cdot 2^{E-\$3FFF}$

Da in das Exponentenfeld also für die Exponenten Werte zwischen 0 und $7FFF eingetragen werden können, ist der mögliche Bereich für den Exponenten 2^{-16383} bis 2^{16384} bzw. 10^{-4931} bis 10^{4932}.

Als Code-Zahl für eine ungültige Operation wird die sog. *Real-Indefinite* verwendet. Bei ihr ist das Vorzeichenbit gesetzt, der Exponent hat den Inhalt $7FFF (alle 15 Bits gesetzt), und die Mantisse hat den Wert 1,1 (Bits 63 und 62 gesetzt):



40.2 LONGREAL-Format (FPU)



Eine LONGREAL besitzt einen Mantissenteil mit 52 Bits, einen Exponenten mit 10 Bits sowie ein Vorzeichen mit 1 Bit. Das Vorzeichen *S* steht dabei an Bitposition 63 (Bit 7 des höchstwertigen Bytes der 8-Byte-LONGREAL) und gilt nur die für Mantisse. Die Mantisse erstreckt sich von Bitposition 51 bis 0, wobei sie umgekehrt dargestellt wird: Das niedrigstwertige Bit der Manitsse steht an Bitposition 51, ihr höchstwertiges an Position 0. Wichtig ist hierbei noch, daß die Bitposition 0 der Mantisse nicht gespeichert wird, da sie implizit vorliegt). Der Exponent füllt den verbleibenden Raum zwischen Position 62 und 52. Er wird »richtig herum« codiert, also das niedrigstwertige Bit des Exponenten an Position 52, sein höchstwertiges an Position 62. Die Realzahl läßt sich wie folgt berechnen:

$R = (-1)^S \cdot M \cdot 2^{E-\$3FF}$

Da in das Exponentenfeld also für die Exponenten Werte zwischen 0 und $7FF eingetragen werden können, ist der mögliche Bereich für den Exponenten 2^{-1023} bis 2^{1024} bzw. 10^{-307} bis 10^{308}.

Als Code-Zahl für eine ungültige Operation wird die sog. *Real-Indefinite* verwendet. Bei ihr ist das Vorzeichenbit gesetzt, der Exponent hat den Inhalt $7FF (alle 11 Bits gesetzt), und die Mantisse hat den Wert 1,1 (Bit 51 gesetzt, das Vorkomma-(J-) Bit ist impliziert!):

79	72 71	64 63	56 55	48 47	40 39	32 31	24 23	16 15	8 7	0
		s	exponent	m			fraction			
		1	11111111111	1000000000000	00000000	00000000	00000000	00000000	00000000	

40.3 SHORTREAL-Format (FPU)

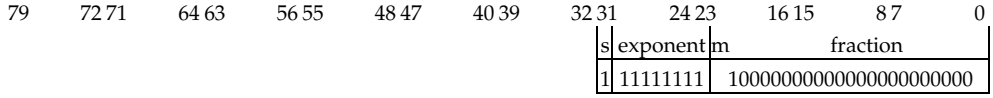
9	8	7	6	5	4	3	2	1	0
79 72 71	**64 63**	**56 55**	**48 47**	**40 39**	**32 31**	**24 23**	**16 15**	**8 7**	**0**
					S	E_7 E_0	M_1		M_{23}

Eine SHORTREAL besitzt einen Mantissenteil mit 23 Bits, einen Exponenten mit 8 Bits sowie ein Vorzeichen mit 1 Bit. Das Vorzeichen *S* steht dabei an Bitposition 31 (Bit 7 des höchstwertigen Bytes der 4-Byte-SHORTREAL) und gilt nur für die Mantisse. Die Mantisse erstreckt sich von Bitposition 22 bis 0, wobei sie umgekehrt dargestellt wird: Das niedrigstwertige Bit der Manitsse steht an Bitposition 22, ihr höchstwertiges an Position 0. Wichtig ist hierbei noch, daß die Bitposition 0 der Mantisse nicht gespeichert wird, da sie implizit vorliegt. Der Exponent füllt den verbleibenden Raum zwischen Position 30 und 23. Er wird »richtig herum« codiert, also das niedrigstwertige Bit des Exponenten an Position 23, sein höchstwertiges an Position 30. Die Realzahl läßt sich wie folgt berechnen:

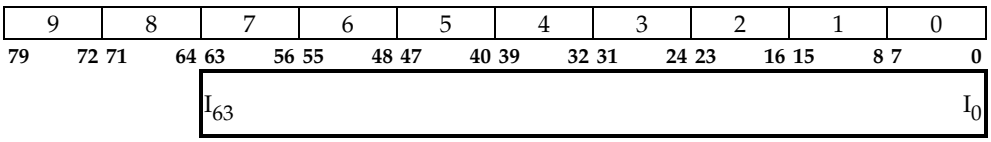
$R = (-1)^S \cdot M \cdot 2^{E-\$7F}$

Da in das Exponentenfeld also für die Exponenten Werte zwischen 0 und $FF eingetragen werden können, ist der mögliche Bereich für den Exponenten 2^{-63} bis 2^{64} bzw. 10^{-18} bis 10^{19}.

Als Code-Zahl für eine ungültige Operation wird die sog. *Real-Indefinite* verwendet. Bei ihr ist das Vorzeichenbit gesetzt, der Exponent hat den Inhalt $FF (alle 8 Bits gesetzt), und die Mantisse hat den Wert 1,1 (Bit 22 gesetzt, das Vorkomma-(J-) Bit ist impliziert!):

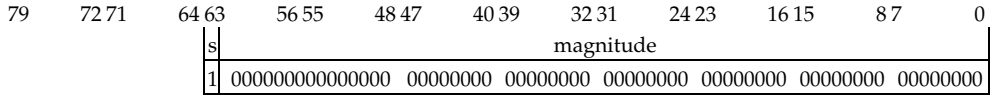


40.4 LONGINT-Format (FPU)

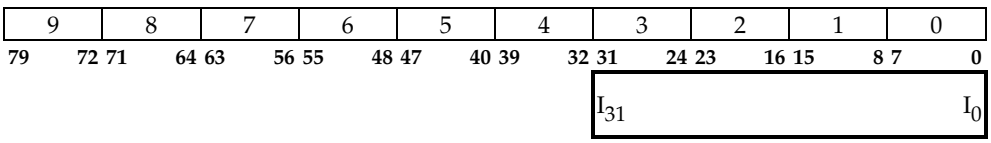


Eine LONGINT wird wie auch beim Prozessor üblich im Zweierkomplement dargestellt. Das höchstwertige Bit steht wie üblich an der höchstwertigen Position 63 der 8-Byte-LONGINT.

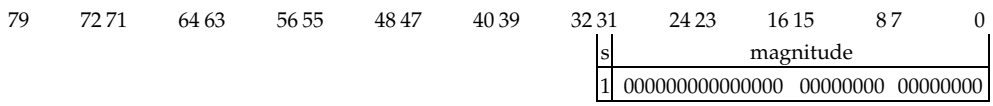
Als Code-Zahl für eine ungültige Operation wird die sog. *Integer-Indefinite* verwendet. Bei ihr ist das Vorzeichenbit (Bit 63) gesetzt und alle anderen Bits gelöscht:



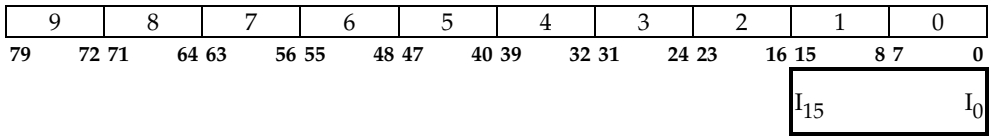
40.5 SHORTINT-Format (FPU)



Eine SHORTINT wird wie auch beim Prozessor üblich im Zweierkomplement dargestellt. Als Code-Zahl für eine ungültige Operation wird die sog. *Integer-Indefinite* verwendet. Bei ihr ist das Vorzeichenbit (Bit 31) gesetzt und alle anderen Bits gelöscht:

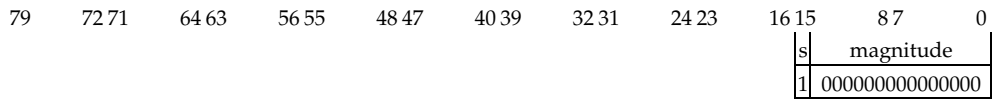


40.6 WORDINT-Format (FPU)

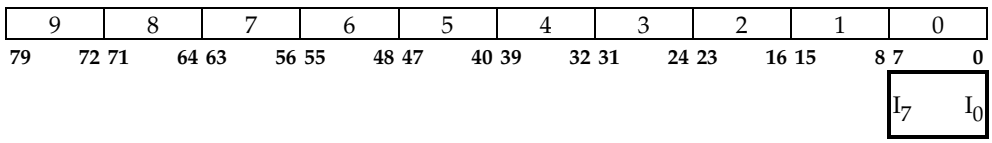


Eine WORDINT wird wie auch beim Prozessor üblich im Zweierkomplement dargestellt. Das höchstwertige Bit steht wie üblich an der höchstwertigen Position 15 der 2-Byte-WORDINT.

Als Code-Zahl für eine ungültige Operation wird die sog. *Integer-Indefinite* verwendet. Bei ihr ist das Vorzeichenbit (Bit 15) gesetzt und alle anderen Bits gelöscht:

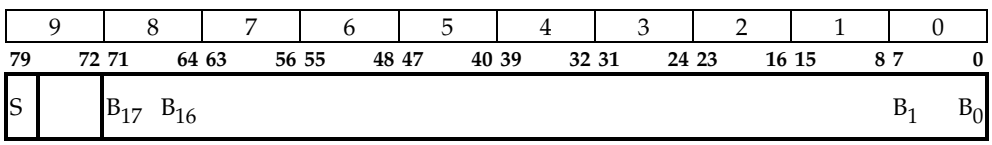


40.7 BYTEINT-Format (FPU)



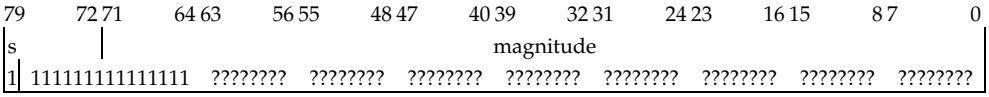
Eine BYTEINT wird wie auch beim Prozessor üblich im Zweierkomplement dargestellt. Das höchstwertige Bit steht wie üblich an der höchstwertigen Position 7 der 1-Byte-Integer.

40.8 BCD-Format (FPU)

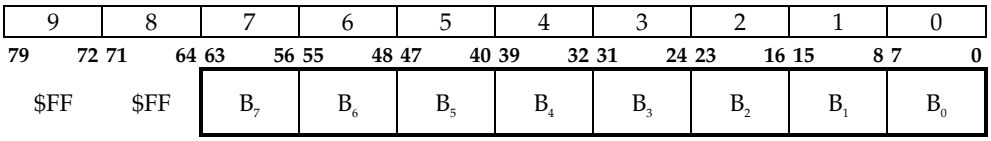


Eine gepackte BCD besitzt zwei Dezimalzeichen pro Byte, wodurch in einer 10-Byte-Struktur 18 Ziffern Platz finden. Das verbleibende Byte dient zur Speicherung des Vorzeichens der Zahl und benutzt nur das obere Nibble des höchstwertigen Bytes. Das untere Nibble wird nicht verwendet.

Als Code-Zahl für eine ungültige Operation wird die sog. *Decimal-Integer-Indefinite* verwendet. Bei ihr ist das Vorzeichenbit (Bit 79) sowie die ansonsten undefinierten Bits 78 bis 72 gesetzt. Die binär codierten Bytes B17 und B16 enthalten $FF (was bei einer BCD-codierten Zahl normalerweise nicht möglich ist!). Alle anderen Bits sind undefiniert.

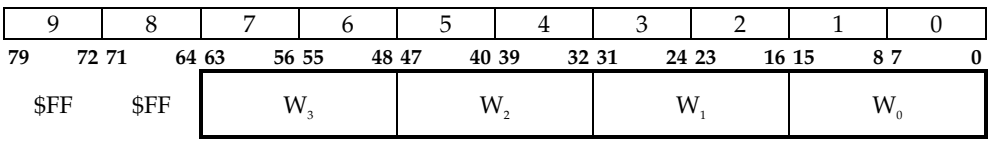


40.9 PACKED BYTES-Format (MMX)



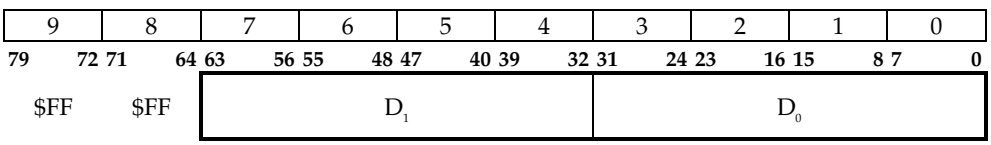
Packed Bytes ist ein Feld aus acht Bytes, die die Bits 63 bis 0 des Registers belegen. Wie bei Intel üblich, liegt dabei das Byte mit der niedrigsten Rangziffer am »unteren« Ende des Registers (B_0), also bei Bitposition 7 bis 0. Die Bits 79 bis 64 können nicht benutzt werden und sind auf »1« gesetzt.

40.10 PACKED WORDS-Format (MMX)



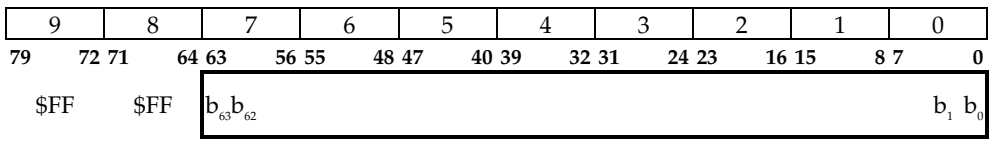
Packed Words ist ein Feld aus vier *Worten*, die die Bits 63 bis 0 des Registers belegen. Wie bei Intel üblich, liegt dabei das *Wort* mit der niedrigsten Rangziffer am »unteren« Ende des Registers (W_0), also bei Bitposition 15 bis 0. Die Bits 79 bis 64 können nicht benutzt werden und sind auf »1« gesetzt.

40.11 PACKED DOUBLEWORDS-Format (MMX)



Packed DoubleWords ist ein Feld aus zwei Doppelworten, die die Bits 63 bis 0 des Registers belegen. Wie bei Intel üblich, liegt dabei das Doppelwort mit der niedrigsten Rangziffer am »unteren« Ende des Registers (D_0), also bei Bitposition 31 bis 0. Die Bits 79 bis 64 können nicht benutzt werden und sind auf »1« gesetzt.

40.12 QUADWORD-Format (MMX)



Ein Quadwort ist ein Feld aus 64 voneinander unabhängigen Bits, die die Bits 63 bis 0 des Registers belegen. Wie bei Intel üblich, liegt dabei das Bit mit der niedrigsten Rangziffer am »unteren« Ende des Registers (b_0), also bei Bitposition 31 bis 0. Die Bits 79 bis 64 können nicht benutzt werden und sind auf »1« gesetzt.

40.13 Normale, nicht normale und unnormale Zahlen

Normalisierte (oder normale) Zahlen sind die üblicherweise anzutreffenden Realzahlen. Sie besitzen ein Vorzeichen, eine Mantisse und einen (vorzeichenbehafteten) Exponenten. Da die Zahlen mit Exponenten dargestellt werden, liegt der Wert der Mantisse immer zwischen 1 und 2.

Warum ist das so? Dazu müssen wir uns daran erinnern, daß die Zahlen binär dargestellt sind. So wird die Zahl 1 als $1 \cdot 2^0$ dargestellt, wobei die Mantisse 1,0000 und der Exponent 0 ist. 2 liegt als $1 \cdot 2^1$ mit der Mantisse 1,000 und dem Exponenten 1 vor, 4 als $1 \cdot 2^2$ mit der Mantisse 1,000 und dem Exponenten 2. Auf diese Weise lassen sich alle Zahlen, die direkte Potenzen von 2 sind, mit einer Mantisse mit dem Wert 1 und einem entsprechenden Exponenten darstellen. Alle Zahlen zwischen zwei aufeinanderfolgenden Potenzen von 2 haben dann eine gebrochene Mantisse.

Da nun aber der Unterschied zwischen zwei aufeinanderfolgenden Potenzen von 2 gerade der Faktor 2 ist, kann die Mantisse niemals einen Wert 2 oder darüber annehmen. Denn man kann ja die Realzahl $R = 3{,}8 \cdot 2^4$ auch umschreiben als $R = 1{,}9 \cdot 2 \cdot 2^4$, was aber gleichbedeutend mit $1{,}9 \cdot 2^5$ ist. Logische Konsequenz: Die obere Schranke für die Mantisse ist 1,999999… . Aber man kann auch verhindern, daß die Mantisse unter einen Wert von 1 sinkt! Denn analog zu dem eben Gesagten kann die Realzahl $R = 0{,}7 \cdot 2^{27}$ umgeschrieben werden als $R = 1{,}4 \cdot 2^{-1} \cdot 2^{27}$, was zu $1{,}4 \cdot 2^{26}$ führt.

Wenn aber die Mantisse immer zwischen 1,000 und 1,999… liegt, so kann man die 1 vor dem Komma vergessen. Dies ist der Grund, weshalb im LONGREAL- und SHORTREAL-Format die Ziffer mit dem Index 0 implizit vorgegeben ist: Sie ist immer 1! Normalisierte Zahlen zeichnen sich also dadurch aus, daß die Bits der Mantisse als Nachkommaanteil einer Mantisse mit der Vorkommaziffer 1 zu interpretieren sind.

Warum aber hat die TEMPREAL nicht ebenfalls eine implizite 1 vor der Mantisse? Antwort: Man kann wesentlich einfacher und schneller rechnen, wenn man die Mantisse vollständig angibt! TEMPREALS sind ja das interne Format, mit dem der Coprozessor arbeitet. Daher verwendet die TEMPREAL eben eine explizit anzugebende 1 als Vorkommateil! Das aber hat Konsequenzen. Denn wer bestimmt, daß nun tatsächlich eine 1 an dieser expliziten Stelle zu stehen hat?

Niemand! Und genau dann, wenn dort eben keine 1 steht, liegt eine sogenannte nicht normalisierte oder denormalisierte Zahl (manchmal auch als nicht normale oder denormale Zahl bezeichnet) vor! Diese Zahlen sind gültige Zahlen, die tatsächlich verarbeitet werden können!

Denn schließlich verbietet niemand dem Prozessor, etwa die beiden Zahlen

+ 1,20345628437594810594 E+234
+ 0,00000345729438394854 E+000

zu addieren! In diesem Beispiel ist die obere Zahl eine normalisierte, die untere eine denormalisierte Zahl. Das Ergebnis ist korrekt berechenbar und wird daher auch berechnet.

Das bedeutet aber, daß der minimal darstellbare Wert einer Realzahl deutlich unter dem Wert liegen kann, der durch Normalisierung erreicht wird. Denn bei gleichem Exponenten kann eine normalisierte Zahl wegen der impliziten 1 keinen kleineren Mantissenwert als 1 haben. Wohl aber nicht normalisierte Zahlen. Der kleinste Mantissenwert ist hier durch die Anzahl von Bits gegeben, die die Mantisse bilden. Bei TEMPREALS sind dies 64 Bits, so daß als Mantisse eine 1 mit 63 führenden Nullen denkbar ist. Dies bedeutet aber einen Wert von 2^{-64} oder $5{,}42 \cdot 10^{-20}$. Eine denormalisierte Zahl kann also um diesen Faktor kleiner sein als ihr normalisierter Zwilling.

Solange im TEMPREAL-Format gearbeitet wird, funktioniert das gut. Es existieren in diesem Fall lediglich dann Probleme, wenn eine denormalisierte Zahl mit einer ebenfalls denormalisierten Zahl z.B. multipliziert wird, wobei nun auch der Bereich unterschritten wird, der mit denormalisierten Zahlen dargestellt werden kann. Dieses Problem wird, wie bei allen anderen Zahlen auch, durch eine Ausnahmebedingung angezeigt; das *Underflow-Exception-Bit* wird gesetzt.

Aber was passiert, wenn eine denormalisierte Zahl in ein anderes Format gebracht werden soll? Also wenn z.B. mit LONGREALs gearbeitet wird? Dann wird ja z.B. durch den Ladebefehl zunächst die LONGREAL in das interne TEMPREAL-Format überführt. Durch die Berechnungen, die dann folgen, können denormalisierte Zahlen entstehen. Falls nun z.B. mit FST die Zahl gespeichert wird, muß es zwangsläufig zu Problemen kommen! Genau das ist auch der Fall, denn LONGREALs erwarten ja eine implizit vorgegebene 1 als Vorkommazahl der Mantisse, die nicht existiert. Der Coprozessor macht auf dieses Problem ebenfalls durch eine Exception aufmerksam: Er setzt das *Denormalized-Operand-Exception-Flag*!

Wird dagegen eine TEMPREAL abgespeichert, so versucht der Prozessor auch hier zunächst, eine eventuell entstandene denormalisierte Zahl dadurch zu normalisieren, daß er den Exponenten verringert, bis die erste von 0 verschiedene Ziffer der Mantisse an die Mantissenspitze gelangt ist. Gelingt dies – was manchmal durchaus sein kann, weil in der Zwischenzeit der Exponent durch Berechnungen erhöht werden konnte –, so wird die normalisierte Zahl gespeichert. Wenn nicht, dann wird die denormalisierte Zahl gespeichert, ohne daß das *Denormalized-Operand-Exception-Flag* gesetzt wird. Dieser Versuch zur Normalisierung erfolgt übrigens auch bei jeder mathematischen Operation!

Die entscheidende Voraussetzung für denormalisierte Zahlen ist also ein Exponent von 0. Was aber passiert, wenn einerseits das höchstwertige (explizit angegebene) Bit der Mantisse 0 ist (eine Voraussetzung für eine Denormale), der Exponent aber andererseits nicht 0 ist (Verletzung der zweiten Bedingung einer Denormalen)?

Dann liegt eine sogenannte unnormalisierte oder unnormale Zahl vor, die von vielen Prozessoren wegen ihrer Unnormalität erst gar nicht unterstützt wird, so beim Pentium oder 80486!

40.14 Null und NANs

Wie wir eben gesehen haben, ist es also möglich, den Darstellungsbereich von Zahlen deutlich zu vergrößern, indem die Mantisse denormalisiert wird. Wie schon gesagt, ist der kleinste denormalisierte Wert eine Zahl, bei der bis auf Bit 0 keines der 63 weiteren Bits der Mantisse gesetzt ist und deren Exponent 0 ist. Nun ist auch offensichtlich, daß eine denormalisierte Zahl, die so klein wird, daß auch das Bit 0 der Mantisse 0 ist, dann wirklich 0 ist! Summa: Ist der Exponent 0 und die Mantisse auch, so ist die ganze Zahl 0.

Wir müssen jedoch noch eine Ausnahme beim Exponenten berücksichtigen. Trägt man nämlich in das Exponentenfeld der TEMPREAL den Wert $3FFF ein, ergibt sich dann nach der Formel zur Realzahldarstellung ein Exponent von 0. Dann ist alles 0, und der Coprozessor geht davon aus, daß die gesamte Zahl 0 ist. Interessanterweise unterscheidet er hierbei je nach Stellung des Vorzeichenbits einen »positiven« und »negativen« Nullwert für die Mantisse. Nimmt man dagegen den Wert $7FFF, so resultiert hieraus ebenfalls nach der Formel ein Exponent mit dem Betrag 0. Bei diesem aber ist das Exponentenvorzeichen gesetzt, der Exponent also negativ!

Dies aber ist für den Coprozessor das Zeichen, daß mit der Zahl etwas nicht stimmt, sie also keine Zahl ist, *Not a Number*, wie man sagt, also eine *NaN*. *NaNs* signalisieren dem Coprozessor, daß der Inhalt des betreffenden Registers nicht definiert ist. Ansonsten wird eine *NaN* ganz normal behandelt: Man kann sie abspeichern, das Vorzeichen umdrehen und den Mantissenwert verändern. Man kann nur eines nicht mit ihr, nämlich rechnen! Wann entstehen *NaNs*? Zum Beispiel, wenn versucht wird, mit leeren Registern zu rechnen. Falls Sie auf die Idee kommen, mittels *FADD ST, ST(2)* z.B. das leere Register ST(2) zum TOS zu addieren, entsteht eine *NaN*, und das *Invalid-Operation-Flag* wird gesetzt.

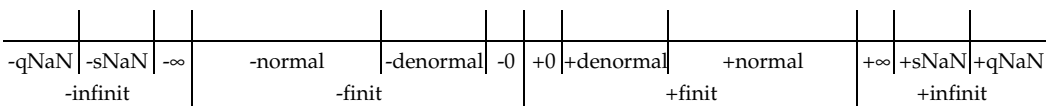
40.15 Unendlichkeiten

Was aber passiert, wenn man z.B. durch 0 dividiert? Dann wird ebenfalls ein *Exception-Flag* gesetzt, das *Zero-Division-Flag*. Auch in diesem Fall entsteht eine *NaN*, allerdings eine andere als bei der Einbeziehung von leeren Registern in Berechnungen. Diese *NaN*, sie heißt auch Unendlichkeit, unterscheidet sich von den anderen dadurch, daß bei ihr anstelle von beliebigen Werten der Wert 1.000 in der Mantisse steht. Unendlichkeiten können ebenfalls je nach den verwendeten Operatoren positiv oder negativ sein, zumindest ab dem 80387, wenn im Kontrollwort das entsprechende Bit gesetzt ist. Der 8087 und 80287 machen hier sowieso keinen Unterschied.

40.16 Zusammenfassung

Stellen wir daher noch einmal die verschiedenen Möglichkeiten zusammen. Das *msb* der Mantisse, also das Bit der Mantisse mit der höchsten Nummer, wird auch als J-Bit bezeichnet. Dieses Bit ist, wie bereits gesagt, in TEMPREALs explizit vorhanden, wird aber in DOUBLEREALs oder SINGLEREALs nur implizit angegeben. Alle weiteren Bits werden als Fraktion, Nachkommateil, bezeichnet. Das dem J-Bit folgende Bit, also das *msb* des Nachkommateils, heißt im Intel-Jargon auch M-Bit. Diese beiden Bits sind bei der Unterteilung der Zahlen wichtig.

Im folgenden wird das Vorzeichenbit S außer Betracht gelassen: Es gibt lediglich an, ob wir uns auf dem rechten oder linken Teil des bei »0« gespiegelten Zahlenstrahls im negativen oder positiven Bereich bewegen. Gehen wir bei der folgenden Betrachtung vom Extrem aus. Die betragsmäßig größte darstellbare »Zahl« ist »∞«. Sie wird durch eine 1,0...0 E1...1 repräsentiert, d.h. das J-Bit ist gesetzt, die Fraktion ist »0« und der Exponent enthält den »Code« für »infinit« (alle Bits gesetzt). Die nächstkleinere Zahl ist eine Zahl, an der die gesamte Mantisse aus »1« besteht und der Exponent einen »echten« Wert darstellt: 1,1...1 E 1...0. Dies ist der Beginn der normalen Zahlen. Sie gehen bis zu einem Wert von 1,0...0 E0..0, also einer »echten« Zahl mit größtem negativem Exponenten. Soll es noch kleiner werden, kann man mit dem Exponenten nichts mehr machen, er hat seinen gesamten Bereich abgedeckt. Also muß die Mantisse ran. Bislang war das J-Bit, der Vorkommaanteil, gesetzt, also vorhanden. Ab jetzt wird er gelöscht. Das hat zwei Konsequenzen: Diese Zahlen sind nur im TEMPRealformat darstellbar, da alle anderen Realformate ein implizit gesetztes J-Bit haben. Die nächstkleinere Zahl hat die Bitfolge 0,1...1 E0..0 – das ist der Beginn der denormalen Zahlen. Sie gehen bis zum kleinsten möglichen Wert: 0,0...0 E 0...0. Das ist »0«.



Bleiben noch zwei Ausnahmen: Was ist, wenn die Mantisse denormal ist, wie bei den Denormalen, der Exponent aber nicht 0 (die andere Bedingung für Denormale) – also der Bereich von 0,1...1 E1...1 bis 0,0...0 E0...1? Und was ist, wenn der Exponent »Infinite« anzeigt, die Mantisse aber nicht 1,0...0 ist?

Beginnen wir mit dem letzten Fall: Der Exponent signalisiert »unendlich«, die Mantisse aber nicht. Dann haben wir die NaNs. Nun gibt es zwei Möglichkeiten. Entweder, das J-Bit ist gesetzt. Dann haben wir wiederum zwei Möglichkeiten: das M-Bit ist gesetzt, dann kann die Mantisse von 1,11..1 bis 1,10..0 variieren. Diese NaNs nennt man Quite-NaNs, qNaNs. Oder, es ist gelöscht, die Mantisse variiert zwischen 1,01..1 bis 1,0...1 (1,0...0 ist ja für die Unendlichkeit reserviert!). Dann liegen Signaling-NaNs oder sNaNs vor. Diese Unterscheidung ist nicht ganz unwichtig! Denn qNaNs heißen quiet, weil sie bei arithmetischen Operationen in der Regel keine Exception auslösen, also ziemlich »leise« sind. Signaling-NaNs dagegen melden in solchen Fällen den Fehler durch Auslösung einer Exception, signalisieren ihn! Noch ein Hinweis: Analog der Definition der Unendlichkeit durch einen Spezialfall einer sNaN (1,0...0 E1...1) definiert man einen weiteren Sonderfall: die *Real-Indefinite*, eine qNaN mit dem Wert -1,10..0 E1..1. Sie wird erzeugt und übergeben, wenn eine Invalid-Operation-Exception erfolgt ist.

Aber es kann auch die zweite der ersten beiden Möglichkeiten eintreten: Das J-Bit ist gelöscht. Dann haben wir NaNs, die von den meisten Prozessoren gar nicht unterstützt werden. Sie können auch nur in TEMPREALs dargestellt werden, da ja alle anderen Realformate ein implizit gesetztes J-Bit besitzen, das immer auf »1« gesetzt wird, es sei denn, der Exponent und der Nachkommaanteil sind beide »0«. Dann wird eine »0« angenommen und auch das J-Bit gelöscht. Diese Pseudo-NaNs, wie sie heißen, folgen ansonsten absolut dem, was eben über sNaNs und qNaNs gesagt wurde: psNaNs decken den Bereich von 0,11..1 bis 0,10..0 ab und pqNaNs von 0,01..1 bis 0,00..1. Es gibt sogar eine Pseudo-Infinite: 0,0...0 E1...1 ist pseudo-∞.

Kommen wir noch kurz zur zweiten Ausnahme: Die Mantisse signalisiert Denormale, der Exponent aber nicht. Auch diese Ausnahme kann nur bei TEMPREALs auftreten und wird daher von den meisten Prozessoren gar nicht erst unterstützt. Wenn also die vermeintliche Denormale nicht denormal ist, nicht »0« und auch keine NaN, wie uns der Exponent sagt, dann ist sie eben unnormal! Unnormale umfassen die Bereiche ± 0,1...1 bis ±0,0...1 bei einem Exponenten von E11..0 bis E00..1.

In der folgenden Tabelle sind zu den verschiedenen möglichen Bitstellungen die korrespondierenden Daten angegeben. Hierbei werden für Mantisse und Exponenten jeweils die Bereiche angegeben. Sie können also beispielsweise aus der Tabelle entnehmen, daß z.B. eine positive, normale TEMPREAL ein Vorzeichen 0, eine Mantisse im Bereich von 1,111…1111 bis 1,000…000 (64 Bits!) und einen Exponenten im Bereich von 111…111 bis 000…000 (15 Bits) haben kann. Dementsprechend besitzt eine negative, unnormale TEMPREAL ein Vorzeichen 1, eine Mantisse im Bereich von 0,11…111 bis 0,00…001 sowie einen Exponenten im Bereich von 111…110 bis 000…001.

Typ	Datum	S	Mant JM	Exp
TempReal	positive NaN	0	11……1 10……1	1…11 1…11
	positive Unendlichkeit	0	10……0	1…11
	positive Normale	0	11……1 10……0	1…10 0…00
	positive Denormale	0	01……1 00……1	0…00 0…00
	Null	0	00……0	0…00
	positive Pseudo-NaN	0	01……1 00……1	1…11 1…11
	positive Unnormale	0	01……1 00……0	1…10 0…01
	negative Unnormale	1	01……1 00……0	1…10 0…01
	negative Pseudo-NaN	1	01……1 00……1	1…11 1…11
	Null	1	00……0	0…00
	negative Denormale	1	01……1 00……1	0…00 0…00
	negative Normale	1	11……1 10……0	1…10 0…00
	negative Unendlichkeit	1	10……0	1…11
	negative NaN	1	11……1 10……1	1…11 1…11
SingleReal **DoubleReal**	positive NaN	0	X1……1 X0……1	1…11 1…11
	positive Unendlichkeit	0	X0……0	1…11
	positive Normale	0	X1……1 X0……0	1…10 0…00
	Null	0 1	X0……0 X0……0	0…00 0…00
	negative Normale	1	X1……1 X0……0	1…10 0…00
	negative Unendlichkeit	1	X0……0	1…11
	negative NaN	1	X1……1 X0……1	1…11 1…11

41 Assembleranweisungen

Die folgende Liste erhebt keinen Anspruch auf Vollständigkeit! Es werden hier lediglich die Assembleranweisungen wiedergegeben, die für den Einstieg in die Assemblerprogrammierung notwendig und sinnvoll sind. Dem geübten Assemblerprogrammierer sei an dieser Stelle weiterführende Literatur empfohlen, falls er die z.T sehr mächtigen Assembleranweisungen der verschiedenen Assembler benutzen möchte.

.186 *MASM, TASM*

Funktion: Aktivierung der ab dem 80186 eingeführten Befehle.

Verwendung: .186

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, bei der Assemblierung die spezifischen Befehle des 80186 zu berücksichtigen. Allerdings werden die Befehle der Prozessoren 8086/8088 ebenfalls unterstützt.

Bemerkungen: *Code, der mit dieser Anweisung erzeugt wurde, ist ggf. nicht auf 8086/8088-Prozessoren lauffähig!*

.286 *MASM, TASM*

Funktion: Aktivierung der ab dem 80286 eingeführten Befehle.

Verwendung: .286

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, bei der Assemblierung die spezifischen Befehle des 80286 zu berücksichtigen. Allerdings werden die Befehle der Prozessoren bis 80186 ebenfalls unterstützt.

Bemerkungen: *Code, der mit dieser Anweisung erzeugt wurde, ist ggf. nicht auf Prozessoren bis zu Typ 80186 lauffähig!*

.287 MASM, TASM

Funktion: Aktivierung der ab dem 80287 eingeführten Coprozessorbefehle.

Verwendung: .287

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, bei der Assemblierung die spezifischen Befehle des 80287 zu berücksichtigen. Allerdings werden ebenfalls die Befehle des Coprozessors bis 8087 unterstützt.

Bemerkungen: *Code, der mit dieser Anweisung erzeugt wurde, ist nicht mit Coprozessoren vom Typ 8087 lauffähig!*

.386 MASM, TASM

Funktion: Aktivierung der ab dem 80386 eingeführten Befehle.

Verwendung: .386

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, bei der Assemblierung die spezifischen Befehle des 80386 zu berücksichtigen. Allerdings werden ebenfalls die Befehle der Prozessoren bis 80286 unterstützt.

Bemerkungen: *Code, der mit dieser Anweisung erzeugt wurde, ist ggf. nicht auf Prozessoren bis zu Typ 80286 lauffähig!*

Falls diese Anweisung *vor* der Definition des Codesegments erfolgt, wird außerdem das sogenannte Flat-Model als Adressierungsart gewählt. In diesem Speichermodell besteht die Adresse einer Speicherstelle aus einem 16-Bit-Selektor und einem 32-Bit-Offset. Somit umfaßt die vollständige Adresse 48 Bits, die linear interpretiert werden. Alle Speicherzugriffe arbeiten dann mit dieser Adressierungsart, was dazu führt, daß jedem Befehl, der auf Speicher zugreift, das *Address Size Prefix* $67 vorangestellt wird.

Nach einer Codesegmentdefinition dagegen wird am gewählten Speichermodell nichts verändert. Das bedeutet, daß u.U. die Speichersegmentierung des Real-Mode mit einem 16-Bit-Segment und einem 16-Bit-Offset (was zu einer 20-Bit-Adresse führt) beibehalten wird. Lediglich die erweiterten Befehle stehen dann zur Verfügung.

Beschreibung: Seite 295

.387 MASM, TASM

Funktion: Aktivierung der ab dem 80387 eingeführten Coprozessorbefehle.

Verwendung: .387

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, bei der Assemblierung die spezifischen Befehle des 80387 zu berücksichtigen. Allerdings werden ebenfalls die Befehle der Coprozessoren bis 80287 unterstützt.

Bemerkungen: *Code, der mit dieser Anweisung erzeugt wurde, ist nicht mit Coprozessoren bis zum Typ 80287 lauffähig!*

.486 MASM, TASM

Funktion: Aktivierung der ab dem 80486 eingeführten Befehle.

Verwendung: .486

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, bei der Assemblierung die spezifischen Befehle des 80486 zu berücksichtigen. Allerdings werden die Befehle der Prozessoren bis 80386 ebenfalls unterstützt.

Bemerkungen: *Code, der mit dieser Anweisung erzeugt wurde, ist ggf. nicht auf Prozessoren bis zu Typ 80386 lauffähig!*

Falls diese Anweisung *vor* der Definition des Codesegments erfolgt, wird außerdem das sogenannte Flat-Model als Adressierungsart gewählt. In diesem Speichermodell besteht die Adresse einer Speicherstelle aus einem 16-Bit-Selektor und einem 32-Bit-Offset. Somit umfaßt die vollständige Adresse 48 Bits, die linear interpretiert werden. Alle Speicherzugriffe arbeiten dann mit dieser Adressierungsart, was dazu führt, daß jedem Befehl, der auf Speicher zugreift, das *Address Size Prefix* $67 vorangestellt wird.

Nach einer Codesegmentdefinition dagegen wird am gewählten Speichermodell nichts verändert. Das bedeutet, daß u.U. die Speichersegmentierung des Real-Mode mit einem 16-Bit-Segment und einem 16-Bit-Offset (was zu einer 20-Bit-Adresse führt) beibehalten wird. Lediglich die erweiterten Befehle stehen dann zur Verfügung.

.8086 MASM, TASM

Funktion: Aktivierung der 8086-Befehle.

Verwendung: .8086

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, den Befehlssatz des 8086/8088 zu aktivieren.

Bemerkungen: Mit diesem Befehl kann auf die »niedrigste« Ebene der Assemblierung zurückgesetzt werden, falls ein *.x86*-Befehl eingestreut wurde und wieder 8086-kompatibler Code erzeugt werden soll.

Beschreibung: Seite 298

.8087 MASM, TASM

Funktion: Aktivierung der 8087-Befehle.

Verwendung: .8087

Arbeitsweise: Diese Anweisung veranlaßt den Assembler, den Befehlssatz des 8087 zu aktivieren.

Bemerkungen: Mit diesem Befehl kann auf die »niedrigste« Ebene der Assemblierung zurückgesetzt werden, falls ein *.x87*-Befehl eingestreut wurde und wieder 8087-kompatibler Code erzeugt werden soll.

: MASM, TASM

Funktion: Definition eines Labels.

Verwendung: *labelname***:**

Arbeitsweise: Der Doppelpunkt weist den Assembler an, die Adresse des Befehls, der unmittelbar folgt, dem Label *labelname* zuzuordnen. Verwendet wird dies z.B. zur Programmierung von Einsprungpunkten bei bedingten oder unbedingten Sprüngen.

Bemerkungen: *TASM und MASM arbeiten hier unterschiedlich!* Unter MASM sind grundsätzlich alle »:«-Labels *lokal* definiert, können also nur innerhalb der aktuellen Routine verwendet und angesprungen werden! TASM dagegen erlaubt die *globale* Verwendung von »:«-Labels, was heißt, daß solchermaßen deklarierte *Labels* grundsätzlich von jedem Punkt innerhalb des aktuellen Assemblermoduls aus angesprungen und angesprochen werden können!

Falls Sie unter MASM global verfügbare *Labels* generieren wollen, so müssen Sie die .MODEL-Anweisung verwenden und auf die »::«-Deklaration ausweichen! Mit zwei hintereinander stehenden Doppelpunkten markierte *Labels* sind unter MASM global definiert. Alternativ können Sie durch OPTION NOSCOPED den Gültigkeitsbereich standardmäßig globalisieren!

Dagegen bewirkt das Voranstellen zweier *at*-Zeichen »@@« bei TASM eine Reduktion der Verfügbarkeit auf den lokalen Bereich. Auch unter TASM gibt es einen Schalter, der eine globale Reduktion der Gültigkeit bewirkt: die Anweisung LOCALS.

Siehe auch *LABEL*.

Beschreibung: Seite 462

; *MASM, TASM*

Funktion: Einleitung eines Kommentars.

Verwendung: ; *Text*

Arbeitsweise: Der Assembler ignoriert den hinter einem Semikolon stehenden Text einer Zeile. Dies kann man nutzen, um in einer Zeile Kommentare einzufügen.

Bemerkungen: Der Assembler ignoriert lediglich den Text zwischen dem Semikolon und dem nächsten Zeilenabschluß (*Carriage Return – Line Feed*). Daher kann dieses Zeichen auch dazu verwendet werden, probehalber Befehle und Anweisungen »auszukommentieren«, indem man an den Anfang der Zeile mit dem Befehl oder der Anweisung ein Semikolon setzt.

Siehe auch *COMMENT*.

= MASM, TASM

Funktion: Definition einer untypisierten Konstanten.

Verwendung: *Const* = *value*

Arbeitsweise: Das Gleichheitszeichen weist dem Symbol mit Namen *Const* den Wert *value* zu. Der Assembler ersetzt während des Assemblierens jedes Vorkommen des Symbols *Const* durch den zugeordneten Wert. Somit sind Konstanten definierbar, die über symbolische Namen ansprechbar sind.

Bemerkungen: *Value* selbst kann ein Wert, eine Adresse oder ein Ausdruck sein, der allerdings durch den Assembler berechenbar sein muß. Das Symbol kann auch durch folgende »=« mehrfach umdefiniert werden. Mit »=« können keine Strings, Schlüsselwörter oder Befehle umdefiniert werden. Hierzu dient *EQU*.

Beschreibung: Seite 447

ASSUME MASM, TASM

Funktion: Zuordnung eines Segmentregisters des Prozessors zu einem Segment oder einer Segmentgruppe.

Verwendung: ASSUME CS:Code, DS:Data

Arbeitsweise: ASSUME teilt dem Assembler mit, daß der Inhalt des spezifizierten Segmentregisters den Segmentteil des zugeordneten Segments beinhaltet.

Bemerkungen: ACHTUNG: ASSUME belegt *nicht* die Segmentregister selbst mit den Segmentadressen! Dies muß, wenn es sich um DS, ES (und in manchen Fällen auch SS sowie ab dem 80386 auch FS und GS) handelt, durch den Programmierer selbst erfolgen! ASSUME stellt lediglich *für den Assembler* die Verbindung zwischen dem Registerinhalt und dem betreffenden Segment her, um die Interpretation des Segmentregisterinhalts während der Assemblierung zu ermöglichen. Dies erfolgt, um ggf. Segmentpräfixe einstreuen zu können, wenn nicht auf die Standardvorgaben zurückgegriffen werden kann.

Beschreibung: Seite 256

.CODE MASM, TASM

Funktion: Definition eines Codesegments.

Verwendung: .CODE

Arbeitsweise: .CODE erleichtert die Erstellung von Segmenten mit ausführbarem Code. Anstelle der etwas umständlichen Deklaration mittels der SEGMENT-Anweisung und der Anmeldung mit ASSUME braucht lediglich unmittelbar vor der ersten Codezeile .CODE zu stehen. Der Assembler erstellt dann automatisch ein Codesegment CODE und weist dem CS-Register dieses Segment zu. Mit .CODE erstellte Segmente dürfen nicht mit ENDS beendet werden! Sie werden entweder durch die END-Anweisung oder durch die Deklaration eines anderen Segments (z.B. .DATA) automatisch beendet.

Bemerkungen: Vor der Verwendung von .CODE muß mittels .MODEL ein Speichermodell eingestellt werden! Dies ist notwendig, damit der Assembler dem Codesegment einen Namen geben kann, der mit den in Hochsprachen verwendeten Definitionen konform ist.

ASSUME und SEGMENT sind weiterhin deklariert und können verwendet werden! Sie sollten hierbei jedoch Vorsicht walten lassen, um die Konventionen nicht zu verletzen!

Beschreibung: Seite 453

COMMENT MASM, TASM

Funktion: Definition von Kommentaren im Quelltext.

Verwendung: COMMENT *Zeichen*
Text
Zeichen

Arbeitsweise: COMMENT leitet einen Block ein, der zur Kommentierung des Quelltexts gedacht ist. Statt jede Kommentarzeile mit einem Semikolon (»;«) zu beginnen, kann mit COMMENT ein Zeichen definiert werden, das Beginn und Ende eines Kommentars anzeigt. Alle dazwischenstehenden Zeichen werden vom Assembler ignoriert!

Bemerkungen: Der Block muß mit dem gleichen Zeichen beendet werden, mit dem er auch begonnen wurde. Verwenden Sie hierzu Zeichen, die im Kommentar nicht vorkommen! So ist z.B. ein (weil selten im Text gebrauchtes und vom Assembler nicht benutztes) häufig verwendetes Kommentarzeichen das Doppelkreuz »#«.

ACHTUNG! Der Assembler verwendet das unmittelbar hinter COMMENT stehende Zeichen als Blockmarkierung! Falls Sie, wie häufig zu beobachten ist, den Kommentar mit einem Rahmen versehen wollen, so darf *Zeichen* auf keinen Fall ein Zeichen sein, das für den Rahmen verwendet wird, weil ansonsten ab dem ersten Auftreten des Blockbegrenzers der Assembler die Assemblierung wieder aufnimmt! Beachten Sie bitte hierzu die ASM-Dateien auf der beiliegenden CD-ROM.

Siehe auch »**;**«

.CONST MASM, TASM

Funktion: Deklaration eines Segments für schreibgeschützte Daten.

Verwendung: .CONST

Arbeitsweise: .CONST deklariert wie .CODE ein Segment mit Hilfe der vereinfachten Segmentanweisungen. Dies bedeutet, daß mittels .MODEL ein Speichermodell gewählt sein muß. Der Anweisung .CODE können dann Deklarationen von Daten folgen.

Bemerkungen: Die Verwendung von .CONST macht nur in den Fällen Sinn, in denen Assemblerroutinen für eine Hochsprache erzeugt werden sollen, die selbst Konstanten in eigenen Segmenten verwaltet, wie z.B. C. Pascal beispielsweise kennt die Unterscheidung Variable – Konstanten in eigenen Segmenten nicht! Aus diesem Grunde kann in diesem Fall .CONST nicht verwendet werden, da ein Zugriff von der Hochsprache aus nicht möglich ist.

Beschreibung: vgl. .CODE

.DATA MASM, TASM

Funktion: Definition eines Datensegments.

Verwendung: .DATA

Arbeitsweise: .DATA arbeitet vollständig analog zu .CODE, deklariert jedoch ein Segment, in dem Daten definiert werden bzw. sind. Wie bei .CODE auch muß mittels .MODEL ein Speichermodell gewählt worden sein.

Bemerkungen: .DATA dient zur Eröffnung eines allgemeinen Datensegments, wie es von Pascal, aber auch von C für initialisierte Daten verwendet wird.

ACHTUNG! *Unter Pascal können Daten nicht aus Assemblermodulen heraus initialisiert werden!* Obwohl Pascal das mit .DATA erzeugte Segment als Datensegment für alle Daten verwendet, dürfen diese nicht initialisiert werden. Zwar erzeugt der Assembler hierbei keinen Fehler, und Speicher für die Daten ist auch reserviert. Dennoch sind alle in diesem Segment stehenden Daten beim Eintritt ins Hochsprachenmodul nicht initialisiert. Sie müssen von Pascal-Teilen (z.B. dem Startcode einer Unit) mit Werten beladen werden.

Beschreibung: vgl. .CODE

.DATA? *MASM, TASM*

Funktion: Deklaration eines Datensegments mit uninitialisierten Daten.

Verwendung: .DATA?

Arbeitsweise: .DATA? arbeitet wie .DATA, ermöglicht jedoch die Unterscheidung zwischen initialisierten und uninitialisierten Daten, die einige Hochsprachen, wie z.B. C ermöglichen.

Bemerkungen: .DATA? kann nicht bei allen Hochsprachen eingesetzt werden.

Beschreibung: vgl. .CODE

DB *MASM, TASM*

Funktion: Reservierung eines Speicherbereichs von einem Byte Größe.

Verwendung: [*label*] DB *value* [, *value* …]

Arbeitsweise: DB (*Define Byte*) reserviert an der aktuellen Position im Speicher genau ein Byte Platz und verbindet diese Speicherstelle, wenn vorhanden, mit dem Namen *Label*. Die Speicherstelle wird dann mit dem in *value* übergebenen Wert des Bytes initialisiert.

Bemerkungen: DB wird üblicherweise verwendet, um Speicherplatz für Daten zu reservieren. Allerdings ist es mit DB auch möglich, Opcodes von Befehlen in den Quelltext einzustreuen.

VORSICHT! Hier unterscheiden sich nicht nur externer und eingebauter Assembler, sondern auch die Assembler unterschiedlicher Hersteller erheblich!

Bei *value* kann es sich auch um Ausdrücke handeln, die den Wert erst bei der Assemblierung bestimmen, solange der Wertebereich für Bytes nicht überschritten wird:

```
Five EQU 5
MyByte DB 3*Five
```

Mit DB können auch mehrere, aufeinanderfolgende Bytes definiert werden. Die Anzahl richtet sich dann nach der Anzahl der übergebenen *values*, die durch Kommata getrennt sein müssen.

Soll ein Byte zwar reserviert, aber nicht initialisiert werden, so wird für *value* ein Fragezeichen verwendet:

```
UnInitialized DB ?
```

Mittels des Operators DUP können mehrere Bytes generiert werden. ACHTUNG! Die Verwendung von DUP ist etwas unglücklich realisiert! So bezeichnet der hinter DB und vor DUP stehende Wert in diesem Fall nicht den Wert der Bytes, sondern die Anzahl der Bytes, die DUP generieren soll. *value* selbst muß dann in Klammern dem DUP-Operator folgen und gilt für alle generierten Bytes:

```
ByteArray DB 20 DUP (1)
```

Hier werden 20 Bytes Platz im Speicher reserviert, die alle mit dem Wert 1 initialisiert werden. Das Label *ByteArray* zeigt hierbei auf die Adresse des ersten definierten Bytes.

Vgl. DD, DF, DP, DQ, DT, DW

Beschreibung: Seite 253

DD — *MASM, TASM*

Funktion: Reservierung eines Speicherbereichs von der Größe eines Doppelworts (4 Bytes).

Verwendung: [*label*] DD *value* [, *value* …]

Arbeitsweise: DD (*Define Doubleword*) reserviert an der aktuellen Position im Speicher vier Bytes Platz und verbindet diese Speicherstelle, wenn vorhanden, mit dem

Namen *Label*. Die Speicherstelle wird dann mit dem in *value* übergebenen Wert initialisiert. Mit DD kann somit Speicherplatz für alle Daten angefordert werden, die genau vier Bytes zur Darstellung benötigen: LONGINTS (vorzeichenbehaftet oder auch nicht) und einfach genaue, coprozessorkompatible Realzahlen (FLOAT, SINGLE, SHORTREAL), aber auch vollständige (*far*-) Zeiger mit Offset- und Segmentanteil.

WICHTIG! Der Assembler nimmt keinen Einfluß darauf, ob die in DDs verzeichneten Zahlen korrekt verwendet werden. So erlaubt er die Verwendung von mit DD deklarierten Daten als Coprozessordaten vom Typ SHORTREAL genauso wie die Angabe des gleichen Datums als Parameter für ein 32-Bit-Register ab den 80386ern. Dies kann zu Problemen führen!

Bemerkungen: Siehe DB

Beschreibung: Seite 283

DF *MASM, TASM*

Funktion: Reservierung eines Speicherbereichs von 6 Bytes.

Verwendung: [*label*] DF *value* [, *value* …]

Arbeitsweise: DF (*Define Float*) reserviert an der aktuellen Position im Speicher sechs Bytes Platz und verbindet diese Speicherstelle, wenn vorhanden, mit dem Namen *Label*. Die Speicherstelle wird dann mit dem in *value* übergebenen Wert initialisiert.

Bemerkungen: Siehe DB

DF dient zum einen dazu, Realzahlen zu definieren, die 6 Bytes verwenden. Diese Realzahlen sind *nicht* coprozessor-kompatibel und werden nur noch aus Kompatibilitätsgründen auch heute noch bereitgestellt, da einige frühere Versionen von Hochsprachen 6-Byte-Reals verwendet haben (vgl. Datentyp REAL in Turbo Pascal).

Neue Anwendung finden DFs heutzutage bei Rechnern ab 80386, bei denen die Speicheradressierung mittels 32-Bit-Offsets erfolgen kann. Ein mit DF deklariertes Datum kann dann eine vollständige Adresse aufnehmen: einen 16-Bit-Selektor mit 32-Bit-Offset.

Vgl. DB, DD, DP, DQ, DT, DW

DP — TASM

Funktion: Reservierung eines Speicherbereichs von 6 Bytes.

Verwendung: [*label*] DP *value* [, *value* …]

Arbeitsweise: DP (*Define Pointer*) reserviert an der aktuellen Position im Speicher sechs Bytes Platz und verbindet diese Speicherstelle, wenn vorhanden, mit dem Namen *Label*. Die Speicherstelle wird dann mit dem in *value* übergebenen Wert initialisiert.

Bemerkungen: Alias für DF

Vgl. DB, DD, DF, DQ, DT, DW

DQ — MASM, TASM

Funktion: Reservierung eines Speicherbereichs von der Größe eines Quadworts (8 Bytes).

Verwendung: [*label*] DQ *value* [, *value* …]

Arbeitsweise: DQ (*Define Quadword*) reserviert an der aktuellen Position im Speicher acht Bytes Platz und verbindet diese Speicherstelle, wenn vorhanden, mit dem Namen *Label*. Die Speicherstelle wird dann mit dem in *value* übergebenen Wert initialisiert.

Bemerkungen: Siehe DB

DQs werden zur Definition von den coprozessor-kompatiblen Realzahlen doppelter Genauigkeit (DOUBLE, LONGREAL) verwendet.

Vgl. DB, DD, DF, DP, DT, DW

DT — MASM, TASM

Funktion: Reservierung eines Speicherbereichs von 10 Bytes Größe.

Verwendung: [*label*] DT *value* [, *value* …]

Arbeitsweise: DT (*Define Ten bytes*) reserviert an der aktuellen Position im Speicher zehn Bytes Platz und verbindet diese Speicherstelle, wenn vorhanden, mit dem Namen *Label*. Die Speicherstelle wird dann mit dem in *value* übergebenen Wert initialisiert.

Bemerkungen: Siehe DB

DTs werden zur Definition von den coprozessor-kompatiblen Realzahlen höchster Genauigkeit (LONG DOUBLE, EXTENDED, TEMPREAL) verwendet. Einige Programmiersprachen definieren darüber hinaus auch BCDs mit 10 Bytes Umfang, die mittels DT definiert werden können (Turbo Pascal: COMP).

Vgl. DB, DD, DF, DP, DQ, DW

DW *MASM, TASM*

Funktion: Reservierung eines Speicherbereichs von der Größe eines Worts (2 Bytes).

Verwendung: [*label*] DW *value* [, *value* …]

Arbeitsweise: DW (*Define Word*) reserviert an der aktuellen Position im Speicher zwei Bytes Platz und verbindet diese Speicherstelle, wenn vorhanden, mit dem Namen *Label*. Die Speicherstelle wird dann mit dem in *value* übergebenen Wert initialisiert.

Bemerkungen: Siehe DB

DWs werden zur Definition von allen Daten verwendet, deren Umfang genau zwei Bytes beträgt. Dies sind alle vorzeichenlosen oder -behafteten Integer, aber auch Offsets von (*near*-)Zeigern.

Vgl. DB, DD, DF, DP, DQ, DT

Beschreibung: Seite 274

ELSE MASM, TASM

Funktion: Anfang des alternativen Zweiges einer Bedingung bei bedingter Assemblierung.

Verwendung:
IF *Bedingung*
Anweisung(en)
ELSE
Anweisung(en)
ENDIF

Arbeitsweise: Falls die nach IF stehende Bedingung nicht erfüllt ist, werden die zwischen ELSE und ENDIF stehenden Anweisungen assembliert.

Bemerkungen: Mit IF, ELSEIF, ELSE und ENDIF sind komplexe Strukturen realisierbar, die die Assemblierung steuern. Beachten Sie bitte, daß es hier (wie bei den vergleichbaren Befehlen in Hochsprachen auch) zu Fehlerquellen kommen kann, wenn mehrere IF-Bedingungen ineinander verschachtelt sind.

ELSEIF MASM, TASM

Funktion: Anfang einer verschachtelten Bedingung im alternativen Zweig einer Bedingung bei bedingter Assemblierung.

Verwendung:
IF *Bedingung1*
Anweisung(en)
ELSEIF *Bedingung2*
Anweisung(en)
ELSE
Anweisung(en)
ENDIF

Arbeitsweise: Falls die nach IF stehende *Bedingung1* nicht erfüllt ist, erfolgt die Prüfung auf erfüllte *Bedingung2*. ELSEIF stellt in diesem Fall den alternativen Zweig der IF-Entscheidung dar, der selbst mit einer Entscheidung verknüpft ist. ELSE ist in diesem Fall der alternative Zweig der verschachtelten Struktur.

Bemerkungen: Mit IF, ELSEIF, ELSE und ENDIF sind komplexe Strukturen erstellbar, die die Assemblierung steuern. Beachten Sie bitte, daß es hier (wie bei den vergleichbaren Befehlen in Hochsprachen auch) zu Fehlerquellen kommen kann, wenn mehrere IF-Bedingungen ineinander verschachtelt sind.

EMUL *TASM*

Funktion: Erzeugung von Code, der sowohl Coprozessorbefehle als auch deren Emulation durch Coprozessoremulatoren ermöglicht.

Verwendung: EMUL

Arbeitsweise: EMUL veranlaßt den Assembler, Tabellen zu erstellen, in denen die Adressen von Coprozessorbefehlen stehen, die emuliert werden sollen. So kann dann der Linker oder Compiler einer Hochsprache, die solche Assemblermodule nutzt, an den entsprechenden Adressen Modifikationen vornehmen, wie z.B. den Eintrag von Interruptaufrufen für die Emulatorroutinen.

Bemerkungen: Die entsprechende MASM-Anweisung ist OPTION EMULATOR.

EMUL kann mit der Anweisung NOEMUL wieder ausgeschaltet werden.

Beschreibung: Seite 438

END *MASM, TASM*

Funktion: Abschluß der Datei mit dem Assemblerquelltext.

Verwendung: END [*Startadresse*]

Arbeitsweise: END signalisiert dem Assembler, daß hiermit der Quelltext abgeschlossen ist, den er zur Assemblierung verwenden soll. Alle sich anschließenden Zeilen werden ignoriert.

Falls der END-Anweisung ein Label als Parameter übergeben wird, so generiert der Assembler Linker-Anweisungen, die diesen dazu veranlassen, beim Programmstart dieses Label als Einsprungspunkt zu markieren.

Bemerkungen: Falls mehrere Assemblermodule zusammengelinkt werden, darf nur einmal ein *Label* für den Programmstart definiert und der END-Anweisung übergeben werden!

Als Startadresse sind alle Labels innerhalb des Assemblerquelltextes erlaubt, aber auch mittels EXTRN eingebundene Labels aus anderen Modulen.

Beschreibung: Seite 265

ENDIF MASM, TASM

Funktion: Beenden einer (verschachtelten) Bedingung bei bedingter Assemblierung.

Verwendung:
IF *Bedingung1*
Anweisung(en)
ELSEIF *Bedingung2*
Anweisung(en)
ELSE
Anweisung(en)
ENDIF

Arbeitsweise: ENDIF schließt die mit IF oder ELSEIF eingeleitete Definition eines Blocks ab.

Bemerkungen: Mit IF, ELSEIF, ELSE und ENDIF sind komplexe Strukturen realisierbar, die die Assemblierung steuern. Beachten Sie bitte, daß es hier (wie bei den vergleichbaren Befehlen in Hochsprachen auch) zu Fehlerquellen kommen kann, wenn mehrere IF-Bedingungen ineinander verschachtelt sind.

ENDM MASM, TASM

Funktion: Beenden der Definition eines Makros.

Verwendung: ENDM

Arbeitsweise: ENDM schließt die Definition eines Makros oder einer Wiederholungsanweisung ab.

Bemerkungen: Bei ENDM darf, im Gegensatz zu ENDS, der Name des Makros *nicht* wiederholt werden!

Beschreibung: Seite 318

ENDP *MASM, TASM*

Funktion: Abschluß einer Prozedur.

Verwendung: [*Prozedurname*] ENDP

Arbeitsweise: ENDP zeigt dem Assembler an, daß eine mit PROC eingeleitete Routine hier beendet ist.

Bemerkungen: ENDP macht einen RET-Befehl nicht überflüssig! Während RET den Prozessor veranlaßt, aus der Routine in den rufenden Teil des Programms zurückzuspringen, und somit lebensnotwendig ist, so ist ENDP lediglich ein Hinweis für den Assembler, daß die Programmierung der Routine im Sinne des Programmieres hier zu Ende ist! Der Assembler kann hierdurch verschiedene Prüfungen vornehmen.

Die modernen Assembler verlangen nicht, daß vor ENDP der Prozedurname wiederholt wird. Dennoch ist es ein guter Rat, es zu tun. Auf diese Weise wird der Quelltext strukturierter und somit lesbarer.

Beschreibung: Seite 292

ENDS *MASM, TASM*

Funktion: Abschluß der Definition eines Segments oder einer Struktur.

Verwendung: [*Segmentname*] ENDS
[*Strukturname*] ENDS

Arbeitsweise: ENDS zeigt dem Assembler, daß an dieser Stelle die Definition eines Segments oder einer Struktur beendet ist und hat, analog zu ENDP, für den Assembler rein informativen, aber extrem wichtigen Charakter.

Bemerkungen: Falls die bei modernen Assemblern überflüssige Wiederholung des Segment- oder Strukturnamens gewählt wird (was wie bei ENDP dringend anzuraten ist), so muß der Name mit dem bei der Segment- oder Strukturdefinition angegebenen Namen übereinstimmen.

Beschreibung: Seite 253, 444

EQU MASM, TASM

Funktion: Definiert ein Symbol für einen Stringnamen, Aliasnamen oder Zahlenwert.

Verwendung: *name* EQU *value*

Arbeitsweise: EQU ordnet einem String- oder Alternativnamen oder aber einem Wert einen Namen zu, unter dem dann auf den Wert etc. zugegriffen werden kann:

```
FiftyFive EQU 55
```

Der Assembler ersetzt hierdurch im Quelltext jedes Auftreten von *FiftyFive* durch den Wert 55. Mit EQU können auch Textstrings benannt werden. Der Assembler wird dann bei jedem Auftreten von *name* im Quelltext diesen Namen durch den Textstring ersetzen. Einsatzgebiete sind alle Assembler*anweisungen*, bei denen Strings erlaubt sind.

```
Message EQU <'Dies ist eine Meldung'>
```

EQU erlaubt auch die Umdefinition von Symbolen, die anderweitig schon vergeben wurden, so z.B. von Schlüsselworten.

```
Test DD ?
BYTE EQU DWORD
fld   BYTE PTR Test
```

Wäre das Schlüsselwort BYTE nicht umdefiniert worden, hätte der Assembler eine Fehlermeldung generiert.

Bemerkungen: Anders als bei »=« darf *name* noch nicht verwendet worden sein, wenn mittels EQU ein Zahlenwert oder Alias zugeordnet werden soll! Während Zeile 3 und 5 keinen Fehler generieren, erfolgt dies bei Zeile 4 sehr wohl:

```
Max = 0FFFFh
Min EQU 0
Max = 07FFFh
Min EQU 0FFh
Min = 0FFh
```

ACHTUNG! EQU kann nicht dazu verwendet werden, Strings als Daten zu definieren, auf die der Prozessor direkt zurückgreifen könnte (etwa wie bei der Datendefinition mittels DB)! Es wird lediglich ein Symbol generiert, was immer dann *lege artis* verwendet werden kann, wenn auch der eigentliche Textstring erlaubt ist! Da der Assembler die Anweisung

```
mov  ax, 'Hi'
```

nicht zuläßt, kann sie auch nicht durch Umdefinition von »Hi« zugelassen werden:

```
Message EQU <'Hi'>
mov  ax, Message          ; Fehler!
mov  ax, [Message]                ; hier auch!
```

Bei der Umdefinition von bereits existierenden Symbolen ist äußerste Vorsicht geboten! Im obigen Beispiel würde nämlich die Umdefinition von BYTE zu DWORD jede Verwendung von BYTE im Quelltext überall dort zunichte machen, wo tatsächlich BYTEs benötigt werden! Resultat: schwer lokalisierbare Fehler!

Beschreibung: Seite 326, 333

ERR *MASM, TASM*

Funktion: Erzeugen einer Fehlermeldung durch den Assembler.

Verwendung: ERR [<*string*>]

Arbeitsweise: ERR erzeugt eine Fehlermeldung, bei der der im String übergebene Text ausgegeben wird. ERR kann damit in Verbindung mit der bedingten Assemblierung dazu verwendet werden, den Programmierer auf eine bestimmte Fehlersituation aufmerksam zu machen.

Bemerkungen: ERR gibt es in verschiedenen Versionen als bedingte und unbedingte Fehlermeldung für die verschiedensten Situationen. Bitte konsultieren Sie Ihr Handbuch.

EXTRN *MASM, TASM*

Funktion: Deklaration eines Symbols, das in einem anderen Modul definiert ist.

Verwendung: EXTRN [*Sprache*] *Symbol*

Arbeitsweise: Sie können üblicherweise, wie in Hochsprachen auch, im Quelltext nur die Symbole verwenden, die dem Assembler zum Zeitpunkt der Assemblierung bekannt sind. Extern definierte Symbole wie Datennamen, Labels oder Programmnamen kennt der Assembler nicht. Diese müssen dem Assembler daher durch die EXTRN-Anweisung quasi »bekannt gemacht« werden.

Bemerkungen: Zur EXTRN-Anweisung gehört neben dem Symbolnamen auch eine Beschreibung, worum es sich bei dem externen Symbol handelt. Schließlich muß der Assembler wissen, ob das Symbol *HalloWelt* beispielsweise einen Einsprungspunkt für den Prozessor markiert (ein Label) oder den Platz, an dem der Ausgabestring »Hallo, Welt!« steht, also ein Datum bezeichnet. Ferner ist für die Adressierung wichtig, ob Adressen vollständig (far; Segment und Offset) sind oder nicht (near; nur Offset) oder welcher Datentyp an der spezifizierten Stelle steht.

Um dies zu erreichen, muß die EXTRN-Anweisung um die Typbeschreibung ergänzt werden, also um eines der Schlüsselworte NEAR, FAR, PROC, BYTE, WORD, DWORD, FWORD, QWORD, TBYTE, bei TASM auch PROC und PWORD, bei MASM auch SBYTE, SWORD, SDWORD, sowie REAL4, REAL8 und REAL10. PROC nimmt je nach gewähltem Speichermodell einen der Werte NEAR oder FAR an! Folgende EXTRN-Deklarationen sind korrekt:

```
EXTRN AByte     : BYTE
EXTRN AWord     : WORD
EXTRN ASingle   : DWORD
EXTRN FarLabel  : FAR
EXTRN NearLabel : NEAR
```

Nach dieser Deklaration kann mit den Symbolen wie mit allen anderen im aktuellen Modul definierten Symbolen gearbeitet werden. Falls besondere Namenskonventionen eingehalten werden sollen, kann EXTRN auch mit der Spezifikation einer Sprache versehen werden. Gültige Angaben sind C, PASCAL, BASIC und Fortran, bei TASM auch PROLOG und NOLANGUAGE, bei MASM auch SYSCALL und STDCALL.

Beschreibung: Seite 354, 378

.FARDATA *MASM, TASM*

Funktion: Deklaration eines Far-Datensegments.

Verwendung: .FARDATA [*name*]

Arbeitsweise: .FARDATA arbeitet wie .DATA, ermöglicht jedoch die Benutzung beliebig vieler Datensegmente, denen dann zur Unterscheidung Namen gegeben und die über den optionalen Parameter der Anweisung angemeldet werden müssen.

Bemerkungen: .FARDATA kann nicht bei allen Hochsprachen eingesetzt werden. Datensegmente, die mittels .FARDATA definiert werden, werden nicht zu einer Datengruppe zusammengefaßt! Sie müssen somit einzeln angesprochen und auch bei der Benutzung der vereinfachten Segmentanweisungen mittels ASSUME beim Assembler angemeldet werden.

.FARDATA? MASM, TASM

Funktion: Deklaration eines Far-Datensegments für uninitialisierte Daten.

Verwendung: .FARDATA? [*name*]

Arbeitsweise: .FARDATA? arbeitet wie .FARDATA.

Bemerkungen: siehe .FARDATA.

FOR MASM

Funktion: Blockwiederholung.

Verwendung: FOR *Parameter, <Argument1 [, Argument2 …]>*
Anweisung(en)
ENDM

Arbeitsweise: FOR hat Ähnlichkeiten mit einem Makro. Die im Block eingeschlossenen Anweisungen werden für jedes in der Argumentenliste aufgeführte Argument durchgeführt. Hierzu besitzt FOR einen Pseudoparameter *Parameter*, der nacheinander alle in der Liste aufgeführten Argumente übergeben erhält:

```
PushAll MACRO
   FOR Register, <ax,cx,dx,bx,sp,bp,si,di>
      push  Register
   ENDM
ENDM
```

Bemerkungen: Anstelle von FOR kann auch IRP verwendet werden.

Parameter kann als Parameter bei den Befehlen im Block verwendet werden. Die Argumentliste muß in spitzen Klammern stehen.

GLOBAL TASM

Funktion: Definition eines global verfügbaren Symbols.

Verwendung: GLOBAL [*Sprache*] *Symbol*

Arbeitsweise: Global ist eine Kombination aus PUBLIC und EXTRN. Alle Daten, Prozeduren und sonstigen Labels, die außerhalb des aktuellen Quelltexts verwendet werden sollen und somit bekannt sein müssen, können mit der GLOBAL-Anweisung versehen werden. Die Anweisung erfolgt identisch zur EXTRN-Anweisung.

Auch eine Sprachspezifikation kann angegeben werden, wenn bestimmte Namenskonventionen zu berücksichtigen sind. Mögliche Angaben sind C, PASCAL, BASIC und FORTRAN, bei TASM auch PROLOG und NOLANGUAGE, bei MASM auch SYSCALL und STDCALL.

Bemerkungen: Falls die GLOBAL-Anweisung unmittelbar vor oder nach der Definition des Symbols ausgeführt wird, braucht der Symboltyp nicht spezifiziert zu werden:

```
GLOBAL MyDate
MyDate DD ?
Global MyProc
MyProc PROC FAR
   :
ret
MyProc ENDP
```

GOTO MASM, TASM

Funktion: Unbedingter Sprung zu einer Markierung.

Verwendung: GOTO *Symbol*

Arbeitsweise: GOTO veranlaßt den Assembler, die Assemblierung zu unterbrechen und sie erst an der Stelle *Symbol* wieder aufzunehmen.

Bemerkungen: GOTO wird im Rahmen der bedingten Assemblierung eingesetzt.

GROUP *MASM, TASM*

Funktion: Zusammenfassung einzelner Segmente zu einer Gruppe.

Verwendung: *Name* GROUP *Segement* [, *Segment2* …]

Arbeitsweise: *Segment1, Segment2* etc. sind die Namen von Segmenten, die z.B. mittels der SEGMENT-Direktive erstellt wurden. Diese Segmente können durch GROUP zu einem Gruppensegment vereinigt werden, das dann als Ganzes mit *name* angesprochen werden kann.

Bemerkungen: Die Hochsprache C beispielsweise verwendet verschiedene Segmente für Daten (siehe .CONST, .DATA, .DATA?, .STACK). Je nach gewähltem Speichermodell werden solche Datensegmente zusammengefaßt, beispielsweise gemäß

```
DGroup GROUP DATA1, DATA2, MyStack, OtherData
```

Auf alle Daten dieser Segmentgruppe kann zugegriffen werden, wenn dem Assembler mit der ASSUME-Anweisung *DGroup*, also der Name der Segmentgruppe, angegeben wird.

IF *MASM, TASM*

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit von einer Bedingung.

Verwendung:

IF *Bedingung1*
Anweisung(en)
ELSEIF *Bedingung2*
Anweisung(en)
ELSE
Anweisung(en)
ENDIF

Arbeitsweise: Falls die nach IF stehende *Bedingung1* erfüllt ist, erfolgt die Assemblierung der sich anschließenden Anweisungen. Andernfalls wird der durch IF … ELSEIF, IF… ELSE oder IF … ENDIF eingeschlossene Block *nicht* assembliert.

Bemerkungen: Mit IF, ELSEIF, ELSE und ENDIF sind komplexe Strukturen realisierbar, die die Assemblierung steuern. Beachten Sie bitte, daß es hier (wie bei den vergleichbaren Befehlen in Hochsprachen auch) zu Fehlerquellen kommen kann, wenn mehrere IF-Bedingungen ineinander verschachtelt sind.

IFB MASM, TASM

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit von einem Argument.

Verwendung:
IFB <*Argument*>
Anweisung(en)
ENDIF

Arbeitsweise: IFB arbeitet genau wie IF, mit dem Unterschied, daß die Bedingung, die erfüllt sein muß, ein leeres Argument ist (**IF B**lank). IFB wird häufig bei der Definition von Makros verwendet:

```
Meldung MACRO Msg
IFB<Msg>
   mov   si, DefaultMsg
ELSE
   mov   si, Msg
ENDIF
   call  StringOut
Meldung ENDM
```

Bemerkungen: Das Argument muß in spitzen Klammern stehen.

IFDEF MASM, TASM

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit von der Existenz eines Symbols.

Verwendung:
IFDEF *Symbol*
Anweisung(en)
ENDIF

Arbeitsweise: IFDEF arbeitet genau wie IF, mit dem Unterschied, daß die Bedingung, die erfüllt sein muß, die Existenz des als Parameter übergebenen Symbols ist.

```
IFDEF ??VERSION
   :
ELSE
   :
ENDIF
```

In diesem Beispiel wird der der IFDEF-Anweisung folgende Block nur dann assembliert, wenn das durch Turbo Assembler definierte Symbol *??VERSION* deklariert ist.

IFDIF[I] MASM, TASM

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit von der Gleicheit zweier Argumente.

Verwendung:
IFDIF <*Argument1*>, <*Argument2*>
Anweisung(en)
ENDIF

Arbeitsweise: IFDIF arbeitet genau wie IF, mit dem Unterschied, daß sich die beiden Argumente voneinander unterscheiden müssen (*IF DIF*ferent). IFDIFI unterscheidet sich von IFDIF dadurch, daß letzteres beim Vergleich die Groß-/Kleinschreibung berücksichtigt, also AX nicht gleich ax setzt, während IFDIFI nicht zwischen Groß- und Kleinschreibung unterscheidet.

```
CopyByte MACRO Date
IFDIF    <ax>, <Date>
   mov   ax, Date
ENDIF
```

Bemerkungen: Die Argumente müssen in spitzen Klammern stehen.

IFE MASM, TASM

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit vom Wert eines Datums.

Verwendung:
IFE *value*
Anweisung(en)
ENDIF

Arbeitsweise: IFE arbeitet genau wie IF, mit dem Unterschied, daß der Block genau dann assembliert wird, wenn *value* 0 ist (*IF E*qual 0):

```
MyDate DW 4711

IFE MyDate
   ERR 'Dieser Text wird niemals angezeigt!'
ELSEIF
   ERR 'Aber dieser, weil MyDate = 4711 > 0!'
ENDIF
```

Bemerkungen: *Value* kann auch ein Ausdruck sein, der jedoch einen genau bestimmbaren Wert ergeben muß!

IFIDN[I] MASM, TASM

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit von der Gleicheit zweier Argumente.

Verwendung:
```
IFIDN <Argument1>, <Argument2>
    Anweisung(en)
ENDIF
```

Arbeitsweise: IFIDN arbeitet genau wie IF, mit dem Unterschied, daß sich die beiden Argumente nicht voneinander unterscheiden dürfen (*IF IDeN*tical). IFIDNI unterscheidet sich von IFIDN dadurch, daß letzteres beim Vergleich die Groß-/Kleinschreibung berücksichtigt, also AX nicht gleich ax setzt, während IFIDNI nicht zwischen Groß- und Kleinschreibung unterscheidet. Vgl. IFDIF.

Bemerkungen: Die Argumente müssen in spitzen Klammern stehen.

IFNB MASM, TASM

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit von einem Argument.

Verwendung:
```
IFNB <Argument>
    Anweisung(en)
ENDIF
```

Arbeitsweise: IFNB arbeitet genau wie IF, mit dem Unterschied, daß die Bedingung, die erfüllt sein muß, ein *nicht-leeres* Argument ist (IF Not Blank). IFNB wird häufig bei der Definition von Makros verwendet. Vgl. IFB.

Bemerkungen: Das Argument muß in spitzen Klammern stehen.

IFNDEF MASM, TASM

Funktion: Einleitung eines Blocks zur bedingten Assemblierung in Abhängigkeit von der Existenz eines Symbols.

Verwendung:
IFNDEF *Symbol*
Anweisung(en)
ENDIF

Arbeitsweise: IFNDEF arbeitet genau wie IF, mit dem Unterschied, daß die Bedingung, die erfüllt sein muß, die Nicht-Existenz des als Parameter übergebenen Symbols ist. Vgl. IFDEF.

INCLUDE MASM, TASM

Funktion: Einfügen von Assemblerquelltext aus anderen Dateien.

Verwendung: INCLUDE *Datei*

Arbeitsweise: INCLUDE unterbricht die Assemblierung des Quelltextes an der aktuellen Position und liest den in *Datei* stehenden Quelltext ein. Dieser wird dann ebenfalls assembliert. Anschließend wird die Assemblierung an der unterbrochenen Stelle des aktuellen Quelltexts fortgesetzt.

Bemerkungen: INCLUDE leistet wertvolle Hilfe bei der Deklaration von Symbolen, die in verschiedenen Quelltexten verwendet werden. Durch Auslagerung aller mit EXTRN einzubindenden Symbole in eine INCLUDE-Datei kann man sicherstellen, daß alle Module, die diese Datei mittels INCLUDE einbinden, über die notwendigen Definitionen verfügen.

IRP MASM, TASM

Funktion: Blockwiederholung.

Verwendung:
IRP *Parameter, <Argument1 [, Argument2 …]>*
Anweisung(en)
ENDM

Arbeitsweise: IRP hat Ähnlichkeit mit einem Makro. Die im Block eingeschlossenen Anweisungen werden für jedes in der Argumentenliste aufgeführte Argument durchgeführt. Hierzu besitzt IRP einen Pseudoparameter *Parameter*, dem nacheinander alle in der Liste aufgeführten Argumente übergeben werden:

```
PushAll MACRO
   IRP Register, <ax,cx,dx,bx,sp,bp,si,di>
      push  Register
   ENDM
ENDM
```

Bemerkungen: Unter MASM kann auch die Anweisung FOR mit identischen Angaben verwendet werden.

Parameter kann als Parameter bei den Befehlen im Block verwendet werden. Die Argumentliste muß in spitzen Klammern stehen.

JUMPS *TASM*

Funktion: Automatische Sprungweitenanpassung bei bedingten Sprüngen.

Verwendung: JUMPS

Arbeitsweise: JUMPS veranlaßt den Assembler, alle bedingten Sprünge daraufhin zu untersuchen, ob das Sprungziel weiter als 127 Bytes von der aktuellen Stelle entfernt ist. Ist dies nicht der Fall, so wird der bedingte Sprung so wie programmiert beibehalten. Andernfalls wird er logisch invertiert. Hierzu wird der bedingte Sprungbefehl in einen unbedingten Sprungbefehl umgewandelt. Sein logisch entgegengesetztes Pendant wird unmittelbar vor diesen (nun) unbedingten Sprungbefehl gesetzt und hat als Ziel die unmittelbar auf den unbedingten Sprungbefehl folgende Stelle.

```
      :
      je   Destination
      :
```

Falls das Sprungziel weiter als 127 Bytes entfernt ist, wird daraus:

```
      :
      jne  Next
      jmp  Destination
Next: :
```

Bemerkungen: JUMPS kann durch NOJUMPS wieder ausgeschaltet werden. Unter MASM kann das gleiche mit OPTION LJMP erreicht werden.

Beschreibung: Seite 462

LABEL *MASM, TASM*

Funktion: Definition eines Labels.

Verwendung: *Name* LABEL *Typ*

Arbeitsweise: Mit LABEL können bestimmte Speicherstellen über symbolische Namen angesprochen werden. Hierbei spielt es keine Rolle, ob es sich um Daten oder ausführbaren Code handelt. Grundsätzlich sind alle Speicherstellen mit Labels versehbar. Dennoch muß dem Assembler kundgetan werden, worum es sich bei der betreffenden Speicherstelle handelt. Hierzu dienen die Operatoren BYTE, WORD, DWORD, FWORD, PWORD, QWORD und TBYTE für Daten und FAR, NEAR und PROC für ausführbaren Code (PROC besitzt je nach gewähltem Speichermodell den Inhalt NEAR oder FAR).

```
EinWort LABEL WORD
ZweiBytes DB ?, ?
```

In diesem Beispiel kann auf das erste Byte von *ZweiBytes* entweder in Byteform direkt (*mov ZweiBytes, al*) oder in Wortform via Label *EinWort* zurückgegriffen werden (*mov EinWort, ax*). Das betreffende Byte hat quasi zwei Labels: das mit der DB-Direktive angegebene Label *ZweiBytes* und das unmittelbar davor stehende und somit auf die gleiche Speicherstelle zeigende Label *EinWort*.

Bemerkungen: Vgl. »:«. Mit LABEL deklarierte Labels sind grundsätzlich global verfügbar (was in diesem Zusammenhang heißt: innerhalb des aktuellen Quelltextes, nicht zu verwechseln mit GLOBAL)!

LOCAL *MASM, TASM*

Funktion: Definition lokaler Variablen in Makros und Prozeduren.

Verwendung:

LOCAL *Symbol* [, *Symbol…*]	in Makros
LOCAL *Variable* [, *Variable…*]	in PROCs

Arbeitsweise: In Makros dient die LOCAL-Anweisung zur Herstellung von Labels, die nur einmal im Quelltext verwendet werden (indem ein eingebauter Zähler des Assemblers bei jedem erneuten Aufruf inkrementiert wird. Diese Zahl wird Teil eines somit exklusiven Labels). Dies wird notwendig, da Makros identische Befehlssequenzen an unterschiedlichen Stellen im Programm generieren, was zu Interferenzen führt.

In Prozeduren führt die Anweisung LOCALS zum Einrichten eines Stackrahmens und zur Reservierung von Speicherplatz auf dem Stack, in dem die lokalen Variablen dann Platz finden. Die Definition solcher lokalen Variablen erfolgt analog der EXTRN-Deklaration, indem ein Labelname, gefolgt von einem Doppelpunkt und einem Datentyp, angegeben wird:

```
Test PROC FAR
LOCAL LocalByte:Byte, LocalWord:WORD
LOCAL LocalPointer:FAR
:
```

Beschreibung: Seite 458

LOCALS *TASM*

Funktion: Aktivieren lokaler Symbole.

Verwendung: LOCALS [*prefix*]

Arbeitsweise: Mit LOCALS kann TASM dazu veranlaßt werden, bestimmte Labels lokal zu definieren. Diese Labels sind nur zwischen den beiden sie »einrahmenden« globalen Labels definiert. Beispiele für solche globalen »Labels« sind PROC und ENDP. Lokale Labels müssen mit zwei bestimmten Zeichen beginnen. TASM verwendet standardmäßig das Präfix @@. Das bedeutet, daß alle Labels, die mit @@ beginnen, nur lokal (z.B. innerhalb der Routine) bekannt sind. Mit dem optional übergebenen Parameter *prefix* können diese beiden Zeichen verändert werden (ACHTUNG: Die Zeichen müssen der Nomenklatur für Labels entsprechen! Führende Ziffern oder Zeichen, die eine andere Bedeutung für den Assembler haben, sind nicht erlaubt!).

Bemerkungen: LOCALS kann durch NOLOCALS wieder aufgehoben werden. MASM verwendet eine andere Art der Gültigkeit (vgl. OPTION SCOPED). Vgl. LABEL.

Beschreibung: Seite 464

MACRO *MASM, TASM*

Funktion: Definition eines Makros.

Verwendung:
Name MACRO [*Parameter1* [, *Parameter2*…]]
Anweisung(en)
ENDM

Arbeitsweise: Mit MACRO können Sie eine bestimmte Folge von Assemblerbefehlen und/oder Assembleranweisungen unter einem gemeinsamen Namen zusammenfassen. Diesem Block können Sie optional Parameter übergeben, die im Makro verfügbar sind. Makros können nach ihrer Definition über ihren Namen im Quelltext aufgerufen werden,

Makros sind keine Unterprogramme! Der Assembler erzeugt bei der Verwendung eines definierten Makros keinen Unterprogrammaufruf, sondern ersetzt statt dessen den Makronamen an der aktuellen Stelle durch die Anweisungen, die im Makro definiert wurden. Dies bedeutet, daß die Befehls- und/oder Anweisungsfolgen genau so oft im Quelltext auftauchen, wie das Makro aufgerufen wurde.

Bemerkungen: Aus diesem Grunde ist Vorsicht bei der Verwendung von Labels in Makros geboten! Je nach Schalterstellung sind nämlich unter MASM oder TASM Labels global gültig. Da in diesem Falle aber bei mehrfacher Makro-Verwendung das Label mehrfach definiert würde, erzeugt der Assembler eine entsprechende Fehlermeldung! Zur Problemlösung siehe LOCAL.

Beschreibung: Seite 319

MASM *TASM*

Funktion: Einschalten des MASM-kompatiblen Modus in TASM.

Verwendung: MASM

Arbeitsweise: MASM veranlaßt TASM, die MASM-kompatible Syntax zu verwenden (Defaulteinstellung).

Bemerkungen: Die anderen Modi von TASM wie QUIRKS und IDEAL werden in diesem Zusammenhang nicht erwähnt!

MASM51 TASM

Funktion: Einschalten des MASM-kompatiblen Modus in TASM unter Berücksichtigung der Erweiterungen von MASM 5.1.

Verwendung: MASM51

Arbeitsweise: MASM51 veranlaßt TASM, die MASM-kompatible Syntax unter Berücksichtigung der Erweiterungen von MASM 5.1 zu verwenden.

Bemerkungen: Die anderen Modi von TASM wie QUIRKS und IDEAL werden in diesem Zusammenhang nicht erwähnt!

.MODEL MASM, TASM

Funktion: Definition des Speichermodells und der verwendeten Sprache für vereinfachte Segmentanweisungen.

Verwendung: .MODEL [*Vorgabe*] *Modell* [*Modul*] [, [*Kontext*] *Sprache*] [, *Vorgabe*] TASM
.MODEL Modell [,Sprache] [,Kontext][,Vorgabe] MASM

Arbeitsweise: .MODEL ist die Voraussetzung für die vereinfachte Segmentanweisung mittels .CODE, .DATA etc. Da hierbei unterschiedliche Deklarationen von Segmentnamen, Gruppenzugehörigkeit etc. notwendig sind, benötigt der Assembler Angaben über das gewünschte Speichermodell sowie ggf. über die Programmiersprache, mit der das Assemblermodul »zusammenarbeiten« soll. Letzteres ist für die korrekte Verwendung von Namens- und Übergabekonventionen entscheidend.

Vorgabe Mit der Angabe von NEARSTACK oder FARSTACK als *Vorgabe* kann bewirkt werden, daß der Assembler das Stacksegment mit dem/den Datensegment(en) zusammenfaßt (NEARSTACK) oder nicht! Im ersten Fall hat dann SS den gleichen Inhalt wie DS. *Vorgabe* kann aus Kompatibilitätsgründen entweder an erster oder letzter Stelle der Liste stehen.

Modell Mit *Modell* definieren Sie das zu wählende Speichermodell. Gültige Angaben sind TINY, SMALL, MEDIUM, COMPACT, LARGE und HUGE. Unter TASM ist auch TPASCAL erlaubt. In diesem Fall darf jedoch kein anderer Parameter aus der Liste verwendet werden (also *nur .MODEL TPASCAL*). MASM erlaubt darüber hinaus noch FLAT, mit dem das Flat-Model gewählt werden kann.

Modul Hier können Sie, falls Sie dies benötigen (z.B. im Modell LARGE), dem aktuellen Assemblermodul einen vom Standardwert abweichenden Modulnamen geben. Dieser Name erfüllt die gleichen Aufgaben wie *name* in der SEGMENT-Anweisung.

Kontext Erlaubte Angaben sind hier bei TASM WINDOWS, NORMAL, ODDNEAR und ODDFAR. Diese Symbole steuern die automatische Codegenerierung für Stackrahmen in WINDOWS-Modulen bzw. bei verschiedenen Overlay-Techniken. Unter MASM sind OS_OS2 und OS_DOS möglich. Vgl. hierzu die entsprechenden Handbücher.

Sprache legt die Programmiersprache fest, mit der das Assemblermodul kompatibel sein soll. Erlaubte Sprachen sind C, PASCAL, BASIC, FORTRAN sowie bei TASM, PROLOG und NOLANGUAGE bei MASM SYSCALL und STDCALL.

Bemerkungen: ACHTUNG! Wenn vor der .MODEL-Anweisung eine .386- oder .486-Anweisung angegeben wird, so werden grundsätzlich 32-Bit-Adressen für die Segmentadressen verwendet. Falls keine Sprache angegeben wird, werden lediglich die Definitionen für die Segmente im gewählten Modell berücksichtigt!

Beschreibung: Seite 453

.NO87 *MASM*

Funktion: Deaktivierung der Coprozessorbefehle.

Verwendung: .NO87

Arbeitsweise: Nach der Angabe von .NO87 erzeugt der Assembler bei der Programmierung von Coprozessorbefehlen Fehlermeldungen.

Bemerkungen: Unter TASM muß PNO87 verwendet werden.

NOEMUL TASM

Funktion: Ausschalten der Erzeugung emulator-fähigen Codes.

Verwendung: NOEMUL

Arbeitsweise: NOEMUL setzt den Assembler in den Zustand zurück, in den er mittels EMUL gesetzt wurde. Nach NOEMUL werden echte Coprozessorbefehle generiert, *die auf Rechnern ohne Coprozessor nicht mehr lauffähig sind!*

Bemerkungen: Unter MASM erfolgt die Zurückschaltung mit OPTION NOEMULATOR.

NOJUMPS TASM

Funktion: Ausschalten der automatischen Sprungzielanpassung.

Verwendung: NOJUMPS

Arbeitsweise: NOJUMPS veranlaßt den Assembler, die mit JUMPS eingeschaltete Korrektur der Sprungbefehle bei bedingten Sprüngen wieder abzuschalten, die automatisch bei Überschreitung der Sprungweiten den Code verändert.

Bemerkungen: Unter MASM heißt der analoge Befehl OPTION NLJMP.

Beschreibung: Seite 462

NOLOCALS TASM

Funktion: Abschalten lokaler Labels.

Verwendung: NOLOCALS

Arbeitsweise: NOLOCALS verbietet die Verwendung lokaler Labels, die mittels LOCALS erlaubt wurde.

Bemerkungen: Unter MASM erfolgt eine ähnliche Aktion mit OPTION NOSCOPED.

Beschreibung: Seite 464

NOMASM51 *TASM*

Funktion: Verbot der MASM-5.1-Erweiterungen.

Verwendung: NOMASM51

Arbeitsweise: NOMASM51 schaltet in den MASM-Modus zurück.

OPTION *MASM*

Funktion: Einstellung verschiedener Optionen.

Verwendung: OPTION *option*

Arbeitsweise: Mittels OPTION kann MASM auf verschiedene Zustände eingestellt werden. Neben vielen anderen möglichen Angaben für *option* werden hier nur diejenigen angesprochen, die Kompatibilität zu Assembleranweisungen unter TASM herstellen:

EMULATOR	vgl. EMUL
NOEMULATOR	vgl. NOEMUL
LJMP	vgl. JUMPS
NOLJMP	vgl. NOJUMPS
M510	vgl. MASM51
NOM510	vgl. NOMASM51
SCOPED	vgl. LOCALS
NOSCOPED	vgl. NOLOCALS

Bemerkungen: SCOPED/LOCALS arbeiten nicht vollständig kompatibel!

Beschreibung: Seite 438, 462, 463

PNO87 *TASM*

Funktion: Deaktivierung der Coprozessorbefehle.

Verwendung: PNO87

Arbeitsweise: Nach der Angabe von PNO87 erzeugt der Assembler bei der Programmierung von Coprozessorbefehlen Fehlermeldungen.

Bemerkungen: Unter MASM muß .NO87 verwendet werden.

PROC MASM, TASM

Funktion: Deklaration einer Prozedur.

Verwendung: *Name* PROC [*Vorgabe*] [*Sprache*] [*Adresse*] [*USES Liste*] [*VarList*]

Arbeitsweise: PROC leitet die Deklaration einer Prozedur namens *Name* ein. Mit *Adresse*, die die Schlüsselworte NEAR oder FAR annehmen kann, kann angegeben werden, ob der anschließende RET-Befehl eine vollständige Rücksprungadresse mit Segment- und Offsetanteil vom Stack nehmen soll oder nur einen Offset (NEAR). Dementsprechend ändern sich auch die Adressen eventuell übergebener Parameter.

Vorgabe dient wie bei .MODEL dazu, ggf. automatisch Änderungen am Stackrahmen für WINDOWS-Module vorzunemen. Gültige Angaben sind WINDOWS und NOWINDOWS. *Sprache* regelt die Namens- und Übergabekonventionen, ebenfalls wie bei .MODEL. Erlaubt sind C, PASCAL, BASIC, FORTRAN sowie bei TASM PROLOG und NOLANGUAGE.

USES leitet eine Liste ein, die die Register angibt, die unmittelbar nach Einrichten des Stackrahmens automatisch auf den Stack kopiert und unmittelbar vor Entfernen des Rahmens restauriert werden. ACHTUNG: die einzelnen Listenelemente müssen durch Leerzeichen, keinesfalls aber dürfen sie durch Kommata getrennt werden. Falls nach *USES* noch Variablen angegeben werden sollen, so muß nach dem letzten Listenelement ein Komma stehen!

VarList spezifiziert eine Liste von Argumenten, die der Routine als Parameter übergeben wird. Sie können hier beliebige Namen angeben, die nicht mit denen im Routinenprototyp im Hochsprachenmodul übereinstimmen müssen. Wichtig ist jedoch, den Typ des Parameters anzugeben. Dies erfolgt durch Anfügen eines Doppelpunktes an den Namen sowie die anschließende Typisierung mittels eines der Schlüsselworte BYTE, WORD, DWORD, FWORD, QWORD, TBYTE und bei TASM auch PWORD. Selbstdefinierte Strukturen sind ebenfalls als Angabe erlaubt! Variablenangaben müssen durch Kommata getrennt werden.

Beispiel: Test PROC PASCAL FAR USES ds es, V1:WORD, V2: DWORD

Bemerkungen: ACHTUNG! Bei Verwendung einer Variablenliste ist die Angabe der verwendeten Sprache unerläßlich, da sie die Reihenfolge auf dem Stack regelt! Wie im Beispiel zu sehen ist, kann dies im Rahmen der PROC-Deklaration erfolgen. Üblicherweise wird dies jedoch global in der .MODEL-Anweisung festgelegt.

Alle Angaben, die auch in der .MODEL-Anweisung erfolgen können (Vorgabe, Sprache), erlauben eine Abweichung der global mit .MODEL vereinbarten Einstellungen lokal für diese eine Prozedur. Damit sind einzelne Prozeduren in Modulen umdefinierbar.

Beschreibung: Seite 285

PUBLIC *MASM, TASM*

Funktion: Symbole als öffentlich deklarieren.

Verwendung: PUBLIC [*Sprache*] *Name*

Arbeitsweise: PUBLIC ist das Gegenstück zu EXTRN. Alle Daten und Prozeduren, die aus anderen Modulen heraus ansprechbar sein sollen, müssen mittels PUBLIC als öffentlich zugänglich deklariert werden.

Name gibt den Namen der öffentlich zugänglichen Prozedur oder des Datums an. Mit *Sprache* kann die Einhaltung der Namenskonvention für eine bestimmte Programmiersprache erzwungen werden. Erlaubt sind C, PASCAL, BASIC, FORTRAN und unter TASM auch PROLOG und NOLANGUAGE, unter MASM auch STDCALL und SYSCALL.

Beschreibung: Seite 350

REPT *MASM, TASM*

Funktion: Deklaration einer Blockwiederholung.

Verwendung:
```
RETP Anzahl
    Anweisung(en)
ENDM
```

Arbeitsweise: Mittels REPT werden die im Block eingeschlossenen Anweisungen genau *Anzahl*-mal wiederholt. *Anzahl* kann auch ein Ausdruck sein, bei dem jedoch der Wert zur Assemblierungszeit bestimmbar sein muß.

SEGMENT MASM, TASM

Funktion: Definition eines Segments mit Attributen.

Verwendung: *Name* SEGMENT [*Ausricht*] [*Komb*] [*Verw*] ['*Klasse*']

Arbeitsweise: SEGMENT definiert ein Segment und teilt ihm verschiedene Attribute zu.

Der Segmentname wird in *Name* übergeben. Auf diese Weise kann ein Segment angesprochen werden. Gibt es mehrere Segmente gleichen Namens, so werden sie (auch wenn es sich um verschiedene Assemblermodule handelt!) zu einem Segment gleichen Namens zusammengefaßt – mit allen hieraus resultierenden Konsequenzen!

Mit *Ausricht* können Sie die Ausrichtung, also den Beginn des Segments im Speicher festlegen. Gültige Angaben sind *BYTE* (Beginn an jeder beliebigen Stelle), *WORD* (Beginn an Adressen, die ohne Restbildung durch 2 teilbar sind), *DWORD* (Beginn an DWORD-Grenzen, also Adressen, die durch 4 teilbar sind), *PARA* (Beginn an Paragraphen: Adressen, die durch 16 teilbar sind) und *PAGE* (eine Seite = 256 Bytes; also sind Seitengrenzen Adressen, die ohne Restbildung durch 256 teilbar sind). Wichtig sind solche Angaben immer dann, wenn eine Ausrichtung Geschwindigkeitsvorteile bringt, wie z.B. die wortorientierte Ausrichtung von Daten bei wortweisem Zugriff oder die DWORD-orientierte Ausrichtung beim Datenaustausch mit den erweiterten (32-Bit-) Registern ab dem 80386. Standardeinstellung: PARA.

Komb steuert die Art, wie Module gleichen Namens zusammengeführt werden, wenn sie gelinkt werden. Mögliche Angaben sind *AT value*, wobei *value* eine Paragraphenadresse darstellen muß. ACHTUNG: *AT* legt die Adresse eindeutig im Speicher fest! Sie kann nicht mehr vom Betriebssystem verändert werden und ist daher nur in Spezialfällen sinnvoll und erlaubt (beispielsweise, wenn ein Datensegment defininert werden soll, das an einer absoluten Adresse liegt, wie z.B. die BIOS-Daten). *COMMON* legt alle Anfangsadressen gleichnamiger Segmente auf die gleiche Adresse, überlagert diese Segmente also! Als Länge wird die Länge des längsten Segments mit dem entsprechenden Namen verwendet. *MEMORY* entspricht der Kombination *PUBLIC* und bewirkt, daß alle Segmente gleichen Namens ein großes zusammenhängendes Segment bilden. Die Länge des resultierenden Segments entspricht der Summe der Länge aller kombinierten Segmente. PRIVATE dagegen wird verwendet, wenn dieses Segment nicht kombiniert werden soll. Dies ist die Standardeinstellung. STACK entspricht PUBLIC, jedoch wird der Linker dazu veranlaßt, das SS-Register mit der Segmentadresse dieses »Spezial-PUBLIC-Segments« zu laden.

Verw spielt nur ab dem 80386 eine Rolle, wenn nämlich neben der üblichen 16-Bit-Adressierung für Segmente auch das Flat-Model mit 32-Bit-Adressen gewählt werden kann. Dementsprechend ist mit *USES16* oder *USES32* für *Verw* in diesem Fall die Angabe des Speichermodells möglich.

Die Angabe für *Klasse*, die mit Anführungszeichen versehen sein muß, steuert die Anordnung der Segmente im Speicher. So werden alle Segmente gleichen Namens aneinandergehängt! Auswirken wird sich dies naturgemäß nur bei Segmenten, die nicht kombiniert werden, wie z.B. bei PRIVATE-Segmenten. Allerdings können auch Segmente mit unterschiedlichen Namen, aber gleicher Klassenzugehörigkeit auf diese Weise aneinandergehängt werden. Beispiel: die Segmente DATA, FARDATA, DATA?, FARDATA? und CONST haben alle unterschiedliche Namen, können aber durch Angabe der Klassenbezeichnung 'Data' aneinandergehängt werden (was *nicht* einer Kombination entspricht! Kombinierte Segmente haben die gleiche Segmentadresse, aneinandergehängte nicht)!

Bemerkungen: Segmentdefinitionen müssen mit ENDS abgeschlossen werden.

Beschreibung: Seite 252

.STACK MASM, TASM

Funktion: Definition eines Stacksegments.

Verwendung: .STACK [*Größe*]

Arbeitsweise: .STACK erzeugt einen Stack im eigenen Segment mit einer durch *Größe* einstellbaren Größe. Defaultwert ist 1024.

Bemerkungen: Die Definition eines Stacksegments macht nur in reinen Assemblerprogrammen Sinn, da jede Hochsprache ein eigenes Stacksegment definiert, das durch die Assemblermodule benutzt werden kann.

STRUC MASM, TASM

Funktion: Definition einer Struktur.

Verwendung:
```
Name STRUC
    Daten
ENDS
```

Arbeitsweise: STRUC faßt verschiedene Daten zu einer Struktur unter einem Namen zusammen. Der verwendete Name kann dann als Strukturmerkmal überall dort verwendet werden, wo Typangaben notwendig sind, wie z.B. bei der Definition von Daten, der Definition von Symbolen, bei Adressierungen usw.

Bei der Datendeklaration einer Struktur gelten alle Richtlinien wie bei jeder anderen Datendeklaration auch, also z.B. die Einzigartigkeit der Namen.

Bemerkungen: STRUC wurde um vielfältige Möglichkeiten erweitert, um auch Objekte bei der objektorientierten Programmierung zu unterstützen. Bitte konsultieren Sie Ihr Handbuch.

Beschreibung: Seite 444

UNION *MASM, TASM*

Funktion: Definition einer Varianten.

Verwendung: *Name* UNION
Daten
ENDS

Arbeitsweise: Mit UNION werden sogenannte variante Datenstrukturen erzeugt.

Beschreibung: Seite 448

USES *TASM*

Funktion: Sicherung der Registerinhalte.

Verwendung: USES *Register* [,*Register*]

Arbeitsweise: USES arbeitet genaus so wie das Feld USES bei der Definition einer Prozedur mit PROC.

Bemerkungen: ACHTUNG! USES muß unter MASM im Prozedurkopf stehen, während TASM diese Anweisung auch unmittelbar nach dem Prozedurkopf erlaubt. In dieser USES-Variante *müssen* die Elemente durch Kommata getrennt werden!

Beschreibung: Seite 460

VERSION *TASM*

Funktion: Einstellen einer speziellen Assemblerversion.

Verwendung: VERSION <ID>

Arbeitsweise: Mittels VERSION können Sie die Besonderheiten verschiedener Assemblerversionen berücksichtigen. Mögliche Angaben für ID sind M400 (MASM 4.0), M500 (MASM 5.0), M510 (MASM 5.1), M520 (MASM 5.2), T100 (TASM 1.0), T101 (TASM 1.01), T200 (TASM 2.0), T250 (TASM 2.5), T300 (TASM 3.0), T310 (TASM 3.1) und T320 (TASM 3.2).

Bemerkungen: Falls Sie mit VERSION die Version eines früheren Assemblers wählen (z.B. TASM 1.01), so stehen Ihnen alle Erweiterungen der nachfolgenden Assemblerversionen nicht mehr zur Verfügung!

WHILE *MASM, TASM*

Funktion: Wiederholung eines Makros.

Verwendung: WHILE *value*
Makro
ENDM

Arbeitsweise: WHILE generiert die durch dieses Schlüsselwort eingeleitete Folge an Assemblerbefehlen und -anweisungen so lange, wie *value* »logisch wahr«, also <> 0 ist. Der Wert des Pseudoparameters *value* muß also innerhalb des Makros verändert werden.

Anhang

A Die verschiedenen Zahlensysteme

Ein Punkt bereitet erfahrungsgemäß jedem, der erste Bekanntschaft mit dem Assembler knüpft, gewaltige Schwierigkeiten: das Binärsystem. Dieses System basiert nicht, wie wir es gewohnt sind, auf zehn unterschiedlichen Zeichen oder Symbolen, eben den uns so vertrauten Ziffern 0 bis 9. Es benutzt vielmehr nur zwei Symbole oder Ziffern. Diese Darstellung resultiert aus der Arbeitsweise der einzelnen elektronischen Bauteile.

A.1 Das Dezimalsystem

Bevor wir andere Zahlensysteme betrachten wollen, sollten wir uns vielleicht noch einmal vergegenwärtigen, wie das »uns innewohnende« Dezimalsystem eigentlich funktioniert.

Da wir mit den Ziffern 0 bis 9 zehn unterschiedliche Symbole haben, können wir durch Aneinanderreihung dieser Symbole beliebige Zahlen erzeugen. Dies erreichen wir, indem wir einfach die Ziffern von rechts nach links aneinanderschreiben, je nachdem, wie groß die Zahl werden soll. Reicht uns nämlich die Ziffer 9 nicht aus, um die gewünschte Zahl (beispielsweise 10) darzustellen, so nutzen wir die nächste, links stehende Stelle.

Wir schreiben einfach die Zahl 4711, ohne uns weiter zu überlegen, daß sie eigentlich eine Summe ist: die Summe aus $4 \cdot 10^3 + 7 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0$. Hier haben wir wieder unsere Basis 10! Das ganze funktioniert natürlich auch mit gebrochenen Zahlen. So ist 0,125 nichts anderes als $0 \cdot 10^0 + 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$.

A.2 Das Binärsystem

Wenn Ihnen das alles klar ist, dann sollte Ihnen auch die Binärzahl 1001001100111 keine Probleme bereiten! Vorausgesetzt, man weiß, daß dies eine Binärzahl ist, steht diese Ziffernfolge für $1 \cdot 2^{12} + 1 \cdot 2^9 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$. Die Voraussetzung ist wichtig, denn es könnte ja auch die Dezimalzahl 1.001.001.100.111 sein! Aus diesem Grunde werden Binärzahlen üblicherweise gekennzeichnet: Man hängt einfach ein kleines *b* an die Binärzahl an und erhält dann 1001001100111b.

Was Sie nun natürlich interessiert, ist die Frage, welche Zahl das in *unserem* Zahlensystem ist! Aber das ist wirklich sehr einfach. Berechnen Sie lediglich die Summe von eben, wobei Sie die Zweierpotenzen als Zahlen zur Basis 10 schreiben: 2^{12} entspricht 4096, 2^9 = 512, 2^6 = 64, 2^5 = 32, 2^2 = 4, 2^1 = 2 und 2^0 = 1. Ergebnis der Rechenoperation: 4711! Damit keine Verwechslung vorkommt: 4711d, wobei *d* bedeutet, daß diese Zahl *dezimal* zu interpretieren ist.

Auch wenn Sie es noch nicht gesehen haben: Wer hindert uns eigentlich daran, auch gebrochene Zahlen binär zu codieren? Natürlich lassen sich auch negative Potenzen zur Basis 2 erheben: 2^{-1}, 2^{-2}, 2^{-4}. So ist also 0,1101b eine erlaubte, gültige und existierende Zahl! Ihr dezimales Pendant ist dann, weil 2^{-1} = 0,5; 2^{-2} = 0,25 und 2^{-4} = 0,0625 ist, 0,8125d.

ACHTUNG*!* Gerade bei gebrochenen Zahlen im Binärsystem ist ein wenig Vorsicht geboten! Sie kennen wahrscheinlich *unsere* Probleme mit Brüchen, die nicht vollständig berechenbar sind: 1/3 oder 1/9 z.B. Diese Brüche liefern gebrochene Zahlen unendlicher Länge. Üblicherweise müssen sie aus offensichtlichen Gründen irgendwann einmal zwangsweise beendet werden, was dann zu Ungenauigkeiten führt. So ist eben 1/3 nicht 0,33333; auch nicht 0,33333333333333; und auch nicht 0,3333333333333333333, so sehr wir dem korrekten Wert auch immer näher kommen.

Nun ist aber das Problem bei Binärzahlen, daß solche »unendlichen« Ziffernfolgen nicht nur dann auftreten, wenn wir es eigentlich ob der bekannten Problematik erwarten! Nein – es kommt schlimmer. Denn eine Zahl wie 0,9, die im Dezimalsystem sehr schnell abbricht und dann genau definiert ist, tut das im Binärsystem nicht: 0,9 = 0,5 + 0,25 + 0,125 + 0,015625 + 0,0078125 + 0,0009765625 + …, was bedeutet, daß sie nicht (bzw. nicht so schnell!) abbricht. Da jeder Chip Zahlen nur mit einer endlichen Anzahl von Ziffern darstellen kann, führt das in unserem Beispiel dazu, daß die im Dezimalsystem genau darstellbare Zahl 0,9 im Binärsystem ungenau repräsentiert ist. Und wenn Sie nun 0,9 quadrieren, so quadrieren Sie auch den Fehler! Das Rechnen mit Binärzahlen ist somit mit mehr Fehlerquellen behaftet als das Rechnen mit Dezimalzahlen! Das führt uns zu folgender Warnung:

ACHTUNG! Falls Sie beim Programmieren (nicht nur in Assembler!) einmal Schwierigkeiten mit Zahlen bekommen, etwa weil Sie z.B. ein Rechenergebnis mit einem Wert vergleichen und das Resultat für Entscheidungen verwenden wollen, denken Sie daran, daß sich hier aufgrund der ungenauen Repräsentation Fehler einschleichen können! Ein Beispiel soll dies demonstrieren: Die Zahl 0,2 wird im Coprozessor im *TempReal*-Format[16] dargestellt als $3FFC CCCC CCCC CCCC CCCD, die Zahl 4,5 als $4001 9000 0000 0000 0000.

Führt man nun die einfache Multiplikation 4,5 · 0,2 mittels FMUL durch, so resultiert hieraus eine Zahl mit der Repräsentation $3FFE E666 6666 6666 6667. Würde nun tatsächlich ein Programm prüfen, ob 4,5 · 0,2 = 0,9 ist und davon einen Sprung abhängig machen, können Sie bis in alle Ewigkeit auf diesen Sprung warten. Denn die Darstellung

---

[16] Vgl. hierzu das Kapitel »Darstellung der Zahlen im Coprozessor«.

der Zahl 0,9 im Coprozessor ist $3FFE E666 6666 6666 6666! Womit Ergebnis und Wert eindeutig unterschiedlich sind! Bitte berücksichtigen Sie dies (so wie es z.B. in der Unit *Mathe* an manchen Stellen durch Rundung berücksichtigt wurde).

A.3 Das Hexadezimalsystem

Zahlen im Binärsystem werden schnell recht unhandlich! Wenn man nämlich 4711 in dezimaler Schreibweise mit vier Ziffern darstellen kann, benötigt man bei binärer Schreibweise schon 13 Stellen! Die gar nicht so große Zahl 4.294.967.296, die mit 10 dezimalen Zeichen auskommt, bringt es auf sage und schreibe 33 Stellen, wenn man sie binär darstellt. Wozu also das Binärsystem, wenn es, verglichen mit dem Dezimalsystem, so unhandlich ist?

Am Binärsystem kommen wir nicht vorbei. Denn in der DV wird digital gearbeitet. So kennen die meisten Bausteine, die in Computern Verwendung finden, nur die beiden Zustände »Strom aus« und »Strom ein«. Das vereinfacht die Realisierung von elektronischen Bauteilen ungemein und sorgt auch für eine gewisse Fehlerunempfindlichkeit. Wollte man nämlich das Dezimalsystem verwenden, so müßten die Zustände »Strom aus«, »Strom zu 10% an«, »Strom zu 20% an«, bis »Strom an« realisiert werden. Dies ist zwar machbar (und wird in bestimmten Fällen auch gemacht!), birgt dann aber die Gefahr größerer Fehler: Was nämlich passiert, wenn ein Chip einen Zustand »Strom zu 15% an« vorfindet? Bei den verwendeten Spannungen und Stromstärken kann es hier leicht zu Komplikationen kommen, die sich dann sehr schnell potenzieren können.

Daher arbeiten auch heute noch praktisch alle Computer digital. Dies ist auch der Grund, warum die Programmierung in Assembler digital erfolgt: Wir arbeiten mit der *Hardware* direkt zusammen und programmieren sie! Zwar lassen die meisten Assembler auch die Angabe dezimaler Werte zu, dennoch sollten Sie sich das Arbeiten mit Binärzahlen angewöhnen. Denn die Assembler übersetzen Ihre Dezimalangaben in Binärwerte. Und wenn Sie mit Debuggern arbeiten, so sehen Sie nur noch die erzeugten Binärwerte.

Aber das Problem der Unhandlichkeit bleibt! Daher machte man sich auf die Suche nach einem Zahlensystem, das eng mit dem Binärsystem verwandt ist. Man fand dann ziemlich schnell eines: das Hexadezimalsystem. Dieses System hat als Basis die Zahl 16. Es müssen also 16 unterschiedliche Symbole vorhanden sein. Nun definierte man, ziemlich unglücklich, wie ich finde, die ersten zehn Symbole analog zum Dezimalsystem. Es sind die Ziffern *0* bis *9*. Daran schließen sich, noch unglücklicher, die Buchstaben *A* bis *F* an – sehr einfallslos! Das bedeutet also, daß bis zur Ziffer 9 Dezimal- und Hexadezimalsystem parallel verlaufen. Dann trennen sich die Wege: 10d = Ah, 11d = Bh bis 15d = Fh.

Das bedeutet nun für unsere Testzahl 4711d, daß sie als Summe aus 4096 + 2·256 + 6·16 + 7 dargstellt werden kann. Nun ist aber $4096 = 16^3$, $256 = 16^2$, $16 = 16^1$ und $1 = 16^0$ und somit 4711d = 1267h!

Die Markierung einer Zahl als Hexadezimalzahl folgt hier unterschiedlichen Regeln. Nachdem das ja sowieso nur Übereinkünfte sind, entschlossen sich die Assembler-Väter, Hexadezimalzahlen mit nachgestelltem *h* zu versehen. Dies war aber nichts für die Entwickler von PASCAL. Diese meinten nämlich, daß eine Hexadezimalzahl durch ein vorangestelltes *$* kenntlich gemacht werden sollte. Da es schon seit grauer Vorzeit eine gewisse Konkurrenz zwischen PASCAL und C gibt, konnten sich die C-Designer nicht durchringen, diese Art zu übernehmen. Sie verwendeten ein vorangestelltes *0x*! Daher sind die Schreibweisen 4711**h**, **$**4711 und **0x**4711 identisch. Sie lassen meistens Rückschlüsse darauf zu, aus welchem Quelltext sie kommen. Ich habe versucht, Hexadezimalzahlen ihrem Kontext entsprechend mit dem Suffix *h* zu versehen, falls *Assembler-Hex-Zahlen* gemeint sind. In PASCAL-Texten habe ich das Präfix *$* verwendet und in C-Programmen eben *0x*.

Doch wieso konnte ich weiter oben behaupten, daß das Hexadezimalsystem dem Binärsystem so ähnlich ist, daß man das eine durch das andere ersetzen kann? Nun – auch hier ist die Antwort einfach: 16 ist eine Potenz von 2! Somit ist jede Zahl, die eine Potenz von 16 ist, automatisch auch eine Potenz von 2. Dies bewirkt die Kompatibilität!

Ein Beispiel: 4096 ist, wie wir gesehen haben, 16^3. Da $16 = 2^4$ ist, kann 4096 auch dargestellt werden als $(2^4)^3$ oder 2^{12}. Letztendlich ist die Darstellung einer Binärzahl als Hexadezimalwert nichts anderes als die Gruppierung der Binärziffern in Vierergruppen, wobei jede Vierergruppe ein neues Hexadezimalzeichen entsprechend ihrem Wert erhält:

1001001100111b = 0001 0010 0110 0111 = 1 2 6 7 = 1267h

Sie haben sicherlich erkannt, daß wir drei Nullen voranstellen mußten, um eine gültige Hex-Ziffer zu erhalten. Aber das ist ja erlaubt und ändert nichts am Ergebnis! Wie Sie sehen, wurden die binären Vierergruppen lediglich durch die korrespondierenden *Hex*-Zeichen ersetzt, gemäß folgender Tabelle:

d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
h	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

ACHTUNG! Die Zahl 1267 z.B. enthält keinerlei hexadezimalspezifische Ziffern (A – F)! Sie kann daher sehr leicht als Dezimalzahl mißinterpretiert werden, falls sie nicht eindeutig als *Hex-Zahl* markiert wird. Ein häufiger Fehler beim Programmieren mit Assembler ist, daß die *Suffixe* vergessen werden. Da dies im Quelltext nicht besonders auffällt, sind hieraus resultierende Fehler schwer auszumachen.

Gewöhnen Sie sich daher an, unter Assembler *immer* mit *Hex-Zahlen* zu programmieren. Einzige Ausnahme: Coprozessordaten dürfen Sie dezimal eingeben, da hier zum einen die Umrechnung äußerst schwierig ist und das eine potentielle und potente Fehlerquelle ist.

Im gesamten Buch wurde versucht, dieser Regel zu folgen. So wurde selbst dann konsequent eine Hexadezimaldarstellung gewählt, wenn dies nicht notwendig war, wie z.B. bei Zahlen wie 3, 5 oder 9. Doch Disziplin in solch banal erscheinenden

Dingen wird belohnt! Daher auch der Rat an Sie: Je disziplinierter Sie sich beim Assemblerprogrammieren verhalten, desto weniger Probleme erzeugen *Sie*!

Noch ein Tip: Alle Assembler geben eine Fehlermeldung aus, wenn eine Zahl mit einem Buchstaben beginnt, da alles, was so anfängt, für sie *Labels* oder *Namen* sind. Daher können Sie die *Hex* $A nicht durch *Ah* angeben. Dies würde der Assembler als *Label Ah* interpretieren, was in einer Datendeklaration natürlich den Unmut des Assemblers heraufbeschwört! Ausweg: Alle Zeichenfolgen, die mit einer Ziffer 0 bis 9 beginnen, interpretiert der Assembler als Zahl. Somit ist die korrekte Eingabe der *Hex 0xA* im Assembler: *0Ah*! Wenn Sie dies auch bei Zahlen durchhalten, bei denen die führende 0 unnötig ist, weil sie mit einer Ziffer beginnen, so erfolgt das irgendwann einmal automatisch. Gewöhnen Sie sich an, immer genau so viele *Hex*-Zeichen anzugeben, wie zur Darstellung eines Datums des gewählten Typs notwendig sind, um die gute Lesbarkeit des Textes zu gewährleisten. Also bei DBs zwei *Hex*zeichen (maximal 255d = 0FFh!), bei DWs vier *Hex*zeichen (maximal 65535d = 0FFFFh) usw. Konsequent *und immer mit führender 0* auch in den Fällen angewendet, in denen es nicht notwendig ist, können Sie jedoch schon anhand der programmierten Konstanten sehen, von welchem Typ sie sind und daß im Beispiel #3 etwas nicht stimmen kann:

```
mov  ax,00001h     ; führende 0 + vier Hex-Ziffern = word!
cmp  al,002h       ; führende 0 + zwei Ziffern = byte!
cmp  bl,123h       ; das kann wohl nicht sein!
```

Merken Sie sich, daß Hexadezimalzahlen im Gegensatz zu Dezimalzahlen immer Binärzahlen sind – nur in anderer, kompakterer Darstellung. Diese Definition wird mancher *Prediger der reinen Lehre* nicht akzeptieren können; denn immerhin unterscheidet sich die Basis der Zahlen. Dennoch macht es meines Erachtens in der DV und beim Assembler Sinn. Wenn also von Binärzahlen gesprochen wird, sind, zumindest in diesem Buch, immer Zahlen gemeint, die auf dem binären Zahlensystem basieren: sowohl Binärzahlen im engeren Sinne als auch die hiervon abgeleiteten Oktal- (Basis: 8) und Hexadezimalzahlen!

A.4 Rechnen mit Binärzahlen

Mit Binär- und Hexadezimalzahlen läßt sich vortrefflich und genauso einfach und korrekt rechnen wie mit Dezimalzahlen (von oben genannten und begründeten Ausnahmen abgesehen!). So ist selbstverständlich, wie jeder intuitiv einsieht, 0Ah · 0Fh = 096h und 1001110110b : 11110b = 10101b. Der Assembler erlaubt jedoch in den meisten Fällen auch die Angabe von Ausdrücken, etwa

```
ANumber  DB 0Ah · 0Fh
ASecond  DW 1001110110b/11110b
```

Wenn alle Stricke reißen, rechnen Sie eben erst dezimal und dann um – so, wie ich das meistens tue!

A.5 Negative Zahlen

Sie können im Binärsystem auch negative Werte darstellen. Sehr viele Datentypen im Assembler besitzen mit dem höchstwertigen Bit zwar ein Vorzeichen-Bit, das ja auch in bestimmten Situationen abgefragt werden kann. Dennoch ist das Setzen dieses Bits nicht der einzige Unterschied, den eine positive und eine negative Zahl haben!

Machen wir uns dies einmal klar. Wenn Sie z.B. die Zahl 2 binär darstellen wollen, so können Sie dies bei Integer-Werten mit sieben Bits (Bit 8 ist ja das Vorzeichen) folgendermaßen tun:

S	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0

Subtrahieren Sie nun, ganz analog zur Subtraktion im dezimalen System, 1. Bedenken Sie, daß es maximal zwei Ziffern gibt, also 0 und 1. Dann erhalten Sie

S	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1

Ein weiteres Mal:

S	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

Und noch einmal!

S	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1

Ist dies klar? Wenn nicht, hier die Gegenprobe. Addieren Sie 1 + 1 zu 0 plus 1 Überlauf!

	S	6	5	4	3	2	1	0
	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	1
←	0	0	0	0	0	0	0	0

Was aber heißt das? Negative Werte haben nicht nur ein gesetztes höchstwertiges Bit! Die Bits, die zur Darstellung der Zahl herangezogen werden, müssen negiert werden. Diese Art der Negation nennt man 2er-Komplement.

B Interrupts

Wenn ein Programm abläuft, so ist der Prozessor ausschließlich damit beschäftigt, dessen Befehle abzuarbeiten. Dies tut er so lange, bis ihn das Programm anweist, damit aufzuhören. Im allgemeinen ist dies das Ende des Programms und führt dazu, daß wieder in das Betriebssystem zurückgesprungen wird.

Doch auch das Betriebssystem ist nur ein Programm! Also wird auch in diesem Fall nur die Befehlsfolge abgearbeitet, die das Betriebssystem für den Fall vorsieht, daß kein anderes Programm ausgeführt wird. Und dies ist: Nichts! Der Prozessor ist also in der Regel damit beschäftigt, eine Endlosschleife auszuführen, in der nichts passiert!

Nun gibt es aber durchaus Gründe, den Prozessor zu bestimmten Zeitpunkten in seinem Tun zu unterbrechen! Denn schließlich weiß weder ein von Ihnen geschriebenes Programm noch DOS beispielsweise, wann Sie eine Eingabe machen. Oder denken Sie an die Uhr Ihres PC, sprich die Systemzeit, die ja laufend aktualisiert werden muß. Oder vielleicht haben Sie Ihren Computer vernetzt? Dann sollte er auch hier ein Ohr am Netzwerk haben! Kurz: es gibt Hunderte von Gründen, warum ein Prozessor im Nichtstun oder im Abarbeiten von erstellten Programmen unterbrochen werden muß.

Da die Beispiele von eben nicht vorhersehbar sind, zumindest für ein Programm, muß eine Möglichkeit gefunden werden, auf solche Ereignisse zu reagieren, ja sie überhaupt wahrzunehmen. Da gäbe es nun die Möglichkeit, routinemäßig in bestimmten Intervallen abzufragen, ob die eine oder andere Bedingung erfüllt ist, soll heißen, ob irgend jemand die Aufmerksamkeit des Prozessors benötigt. Dies wäre z.B. als Abfrage in einer Schleife des Betriebssystems realisierbar.

Diese Möglichkeit ist nicht besonders wirkungsvoll. Denn dies würde bedeuten, daß für alle Eventualitäten vorgesorgt und alle nur erdenklichen »Störer« routinemäßig abgefragt werden. Wenn Sie an alle möglichen Peripherie-Geräte denken, die an PCs anschließbar sind, ist das ein durchaus aufwendiges und zeitraubendes Unterfangen. Denn in 99,9% der Zeit stört ja niemand! Unter diesen Umständen ist abzusehen, daß die *Performance* nicht besonders gut wird.

Daher entschloß man sich, Peripheriegeräte, die die Aufmerksamkeit des Prozessors benötigen, mit der Fähigkeit auszustatten kundzutun, daß der Prozessor benötigt wird. Der Sekretär des Prozessors, der Interrupt-Controller, sammelt nun alle Unterbrechungswünsche, verleiht ihnen je nach Wichtigkeit eine Priorität und klopft dann seinerseits beim Prozessor an. Dieser wird nun gestört. Er läßt sich die Liste mit den Interruptanforderungen zeigen und sucht sich im Alarmplan »Was muß ich tun, wenn jemand etwas von mir will?« die Vorschrift heraus, in der steht, was nun zu tun ist.

Da er in seiner eigentlichen Arbeit unterbrochen wurde, später aber weitermachen möchte, muß er sich jedoch vorher merken, an welchem Problem er gerade gearbeitet hatte. Dazu sichert er die Adresse des Befehls, der eigentlich als nächstes aus-

geführt werden sollte, in seinen privaten Datenspeicher, den Stack. Doch nicht nur das! Er sichert auch den Zustand der Flags. Und das ist sinnvoll! Denn es könnte ja sein, daß er just in dem Moment unterbrochen wird, in dem er nach einem Testbefehl anhand der durch diese gesetzen Flags einen bedingten Sprung ausführen muß. Würde der Flagzustand »vergessen«, so resultierten hieraus Fehler!

ACHTUNG! Was er jedoch nicht sichert, ist alles andere! Also z.B. den Inhalt von Registern. Das muß als allererstes die Routine vornehmen, die anschließend abgearbeitet wird. Vor ihrer Beendigung muß sie den alten Zustand restaurieren! Dies liegt also in der Verantwortung der Programmierer von Interruptroutinen!

Da er nun von einer Ausnahmesituation hoher Priorität unterbrochen wurde, will sich der Prozessor nicht bei ihrer Bearbeitung nochmals unterbrechen lassen und löscht deshalb das Interrupt-Enable-Flag. Der Interrupt-Controller weiß nun, daß der Prozessor auf keinen Fall unterbrochen werden will, und sammelt daher evtl. einkommende Wünsche.

Nach diesem Präludium nun tut der Prozessor genau das, was der Alarmplan für solche Fälle vorsieht. Dieser Alarmplan ist das Betriebssystem DOS und das BIOS, also der Teil der Software, der festverdrahtet im ROM des Rechners liegt und die Zusammenarbeit zwischen dem Prozessor und dem Peripherie-Gerät regelt, das die Unterbrechung verursacht hat. Wurde dieser Plan abgearbeitet, setzt der Prozessor wieder das Interrupt-Enable-Flag, holt sich den Flagzustand wieder und macht mit der unterbrochenen Arbeit weiter bis zur nächsten Unterbrechung, die evtl. schon wartet.

Soweit der bildliche Ablauf dessen, was passiert, wenn Peripheriebausteine Aufmerksamkeit verlangen. Wie Sie gesehen haben, exisitert eine Sammlung von ganz speziellen Routinen, die ganz konkret auf die Eigenheiten der anderen Chips eingehen können. Die Adressen dieser Routinen stehen in einer Liste. Das bedeutet, daß unabhängig von Programmen Systemroutinen im Speicher liegen, die im Falle von Unterbrechungen aufgerufen werden können.

Neben den eben besprochenen *Hardware-Interrupts*, also Unterbrechungen, die durch die Hardware ausgelöst werden, gibt es auch noch *Software-Interrupts*. Diese werden, wie der Name schon sagt, von der Software, also von Programmen, ausgelöst. Auch diese Routinen bedienen sich ganz speziellen Codes, der auf bestimmte Peripheriebausteine abgestimmt ist. Sie sind im Prinzip ebenfalls Verhaltensmaßregeln für den Prozessor für den Alarmfall. Doch dieses Mal ist dies nicht eine Unterbrechung »von außen«, sondern eine durch den Prozessor selbst.

Das heißt nichts anderes, als daß bestimmte Systemroutinen, die für bestimmte Zwecke zur Verfügung gestellt werden, auch in Ihren eigenen Programmen als Unterprogramme aufgerufen werden können – hier z.B. die Textausgabe des Strings auf dem Bildschirm, die uns das Betriebssystem DOS abnimmt. Da diese Routinen losgelöst vom Programm existieren, können sie auch nicht einfach mit dem CALL-Befehl aufgerufen werden.

Denn dazu müßte ja bekannt sein, an welcher Stelle im Speicher sie stehen. Und das wissen *Sie* nicht – aber der Alarmplan! Rufen Sie also nur Alarmfall (per INT-Befehl!) $21 mit dem Wunsch $2C auf – den Rest erledigen der Prozessor und das Betriebssystem.

Das Betriebssystem und das BIOS stellen eine Menge Funktionen zur Verfügung! Falls Sie an diesem Thema weiteres Interesse und noch wenig Vorkenntnisse haben, sei auf Sekundärliteratur verwiesen. Für dieses Werk genügen die »Alarmereignisse« $2C (»Textausgabe«) und $4C (»Beenden des Programmes«) des Alarmplaneintrags $21 (=»Betriebssystem«).

ACHTUNG! Beide Formen von Unterbrechungen lassen den Prozessor absolut gleich reagieren! In beiden Fällen werden der Flagzustand und die Rücksprungadresse auf dem Stack abgelegt und das Interrupt-Enable-Flag wird gesetzt! Beide Arten von Routinen müssen daher auch gleich beendet werden: durch den IRET-Befehl, der nun die Flags zurücklädt und an die ursprüngliche Adresse zurückspringt.

C tpASM

C.1 Der tpASM-Editor

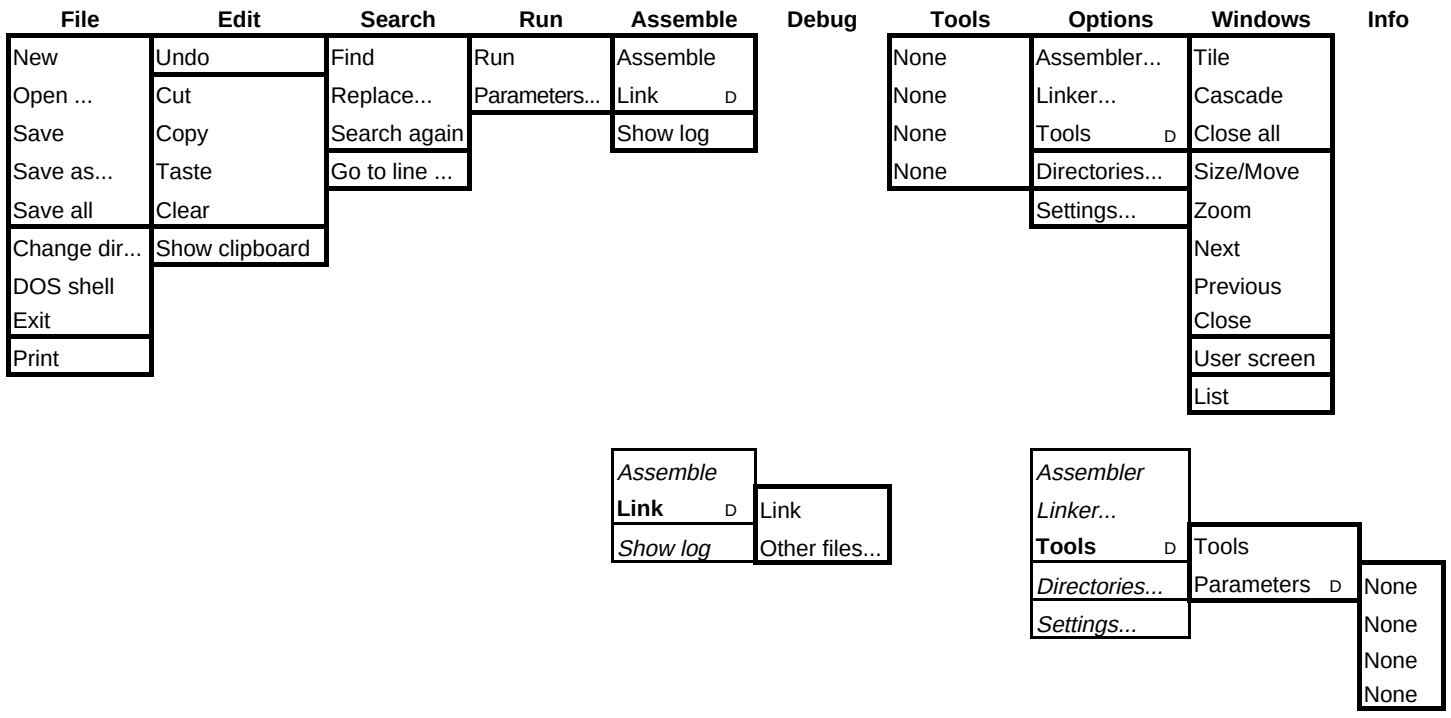
Im folgenden sind die Tasten und Tastenkombinationen aufgeführt, die innerhalb der Editorfenster von tpASM definiert sind. In einigen Fällen bestehen alternative Tastenkombinationen, die gleichberechtigt verwendet werden können. Ferner lassen sich bestimmte Aktivitäten (*Undo*, *Serach* etc.) auch über Funktionstasten und/oder das Menü aktivieren.

Befehl	**Tastenkombination**	**alternativ**
Cursorbewegung:		
Zeichen nach links	←	Ctrl + S
Zeichen nach rechts	→	Ctrl + D
Wort nach links	Ctrl + ←	Ctrl + A
Wort nach rechts	Ctrl + →	Ctrl + F
Zeile nach oben	↑	Ctrl + E
Zeile nach unten	↓	Ctrl + X
Seite nach oben	Page ↑	Ctrl + R
Seite nach unten	Page ↓	Ctrl + C
Zeilenanfang	Home	Ctrl + Q S
Zeilenende	End	Ctrl + Q D
Textanfang	Ctrl + Page ↑	Ctrl + Q R
Textende	Ctrl + Page ↓	Ctrl + Q C
Einfügen und Löschen		
Zeichen löschen	Del	Ctrl + G
Zeichen links löschen	Backspace	Ctrl + H
Zeile löschen	Ctrl + Y	
Zeile einfügen	Ctrl + M	

bis Zeilenende löschen	Ctrl + Q Y	
bis Zeilenanfang löschen	Ctrl + Q H	
Wort löschen	Ctrl + T	
Einfügemodus an/aus	Ins	Ctrl + V
Blockbefehle		
Blockanfang markieren	Ctrl + K B	
Blockende markieren + kopieren	Ctrl + K K	
in Zwischenablage kopieren	Ctrl + Ins	
in Zwischenablage ausschneiden	Shift + Del	Ctrl + K Y
aus Zwischenablage einfügen	Shift + Ins	Ctrl + C
Blockende markieren + löschen	Ctrl + Del	
Blockende markieren + drucken	Ctrl + K P	
Blockerweiterungsbefehle		
Zeichen nach links	Shift + ←	
Zeichen nach rechts	Shift + →	
Wort nach links	Shift + Ctrl + ←	
Wort nach rechts	Shift + Ctrl + →	
Zeilenanfang	Shift + Home	
Zeilenende	Shift + End	
gleiche Spalte, vorangehende Zeile	Shift + ↑	
gleiche Spalte, nächste Zeile	Shift + ↓	
Seite nach oben	Shift + Page ↑	
Seite nach unten	Shift + Page ↓	
Textanfang	Shift + Ctrl + Home	
Textende	Shift + Ctrl + End	
sonstige Befehle		
automatischer Einzug ein/aus	Ctrl + O	
Suchen	Ctrl + Q F	
Suchen und Ersetzen	Ctrl + Q A	
Suche wiederholen	Ctrl + L	
Rückgängig	Ctrl + U	

C.2 Das Menü von tpASM

Auf der folgenden Seite ist das Menü von tpASM dargestellt, wie es aussehen würde, wenn man alle Menüpunkte nebeneinander aufklappen könnte. Beachten Sie bitte in der unteren Hälfte der Darstellung die Untermenüs, die sich zeigen, wenn die Menüpunkte *Assemble/Link*, *Assemble/Link/Link* sowie *Option/Tools*, *Option/Tools/Tools* und *Option/Tools/Tools/Parameters* angewählt werden.


File
New
Open ...
Save
Save as...
Save all
Change dir...
DOS shell
Exit
Print
Edit
Undo
Cut
Copy
Taste
Clear
Show clipboard
Search
Find
Replace...
Search again
Go to line ...
Run
Run
Parameters...
Assemble
Assemble
Link D
Show log
Debug
Tools
None
None
None
None
Options
Assembler...
Linker...
Tools D
Directories...
Settings...
Windows
Tile
Cascade
Close all
Size/Move
Zoom
Next
Previous
Close
User screen
List
Info
Assemble
Link D
Show log
Link
Other files...
Assembler
Linker...
Tools D
Directories...
Settings...
Tools
Parameters D
None
None
None
None

D Glossar

Adreßbus Als Adreßbus bezeichnet man die Summe aller Leitungen, mit denen ein Prozessor den verfügbaren Speicherbereich adressieren kann. Prozessoren der Typen 8086/88 und 80186/88 sowie deren Clones besitzen 20 Adreßleitungen, die den Adreßbus bilden. Die Prozessortypen 80286 und deren Nachbauten besitzen 24 Adreßleitungen, ab dem 80386 kommen bis zum derzeitig aktuellen Prozessor Pentium 32 Leitungen zum Einsatz.

Adreßraum Der Adreßraum ist der Speicher, der mit den verfügbaren Adreßleitungen des Adreßbusses angesprochen werden kann. Beim Adreßbus der 8086- und 80186-Prozessoren können 2^{20}, beim 80286 2^{24} und ab dem 80386 2^{32} Byte Speicher angesprochen werden, weshalb der Adreßraum dieser Prozessoren 1 MByte, 16 MByte bzw. 4 GByte beträgt.

Affines Modell Beim affinen Modell der Zahlentheorie liegen alle Zahlen auf einem Zahlenstrahl, der von $-\infty$ bis $+\infty$ reicht. Diese beiden Extremwerte werden somit als unterschiedlich behandelt. Dies ist die Standardbehandlung aller Coprozessoren ab dem 80387. Sie wird auf jeden Fall verwendet, selbst wenn über das Bit *Infinity Control* des Kontrollworts das projektive Modell angewählt wird. Zwischen beiden Modellen wählen kann man nur beim 8087 und 80287.

Alignment Das Ausrichten (*alignment*) eines Segments ist dessen Ansiedlung im Speicher auf eine Weise, daß die Segmentgrenze bei bestimmten Potenzen von 2 zu liegen kommt. Wenn z.B. byteweise ausgerichtet wird, heißt das, daß die Segmentgrenzen an Vielfachen von $2^0 = 1$ Byte beginnen können, also de facto an jeder beliebigen Adresse. Ein *Word-Alignment* richtet die Segmentgrenzen an Worten, also $2^1 = 2$ Bytes aus. Dies bedeutet, daß die Segmentgrenzen ohne Restbildung durch 2 teilbar sein müssen. Weitere häufig benutzte Ausrichtungen sind DWort-Ausrichtungen ($2^2 = 4$ Bytes), Paragraphen-Ausrichtungen ($2^4 = 16$ Bytes) und Seitenausrichtungen ($2^8 = 256$ Bytes). Sinn und Zweck des Ausrichtens ist die Beschleunigung des Zugriffs auf Daten durch den Prozessor oder andere Komponenten, falls die Segmente in entsprechender Weise ausgerichtet werden: wortweise beim Zugriff eines Prozessors mit 16-Bit-Datenbus, dwortweise beim 32-Bit-Datenbus, seitenweise beim Zugriff eines *Cache-Controllers*.

Assemblat Als Assemblat bezeichnet man das Resultat der erfolgreichen Aktivität des Assemblers. Das Assemblat besteht aus einer Datei, in der neben den Opcodes der Prozessorbefehle auch diverse andere Informationen wie Datenbytes, Adreßlisten und Emulationsangaben stehen. Das Assemblat ist üblicherweise in OBJ-Dateien verzeichnet, die dann entweder durch Linker oder durch manche Hochsprachencompiler weiterverwendet werden können, um letztendlich lauffähige

Programme zu erzeugen. Assemblate sind nur sehr schwer les- und interpretierbar, da man – im allgemeinen in Unkenntnis der Struktur einer OBJ-Datei und aufgrund der Bytecodierung der Opcodes – nur umständlich die Prozessorbefehle eruieren kann. Daher benutzt man z.B. mit Debuggern Programme, die die Bytefolgen der Opcodes wieder in *Mnemonics* disassemblieren können.

Ausrichtung siehe *alignment*.

Bedingte Assemblierung Als bedingte Assemblierung bezeichnet man analog zur bedingten Kompilierung die Tatsache, daß die Assemblierung des gesamten Quelltextes oder auch nur von Teilen davon von Bedingungen abhängig gemacht werden kann. *Achtung:* Ein häufig gemachter Fehler besteht darin, anzunehmen, daß diese Bedingungen auch noch zur Laufzeit eines Programms bestehen und somit das Programm gemäß dieser Bedingungen unterschiedlich abläuft. *Dies ist nicht der Fall!* So kann z.B. *nicht* mittels der Anweisung *IF @Cpu AND 4* Code erzeugt werden, der sich im Falle des Ausführens des assemblierten (und natürlich gelinkten) Programms auf 80386-Prozessoren anders verhält als auf 8086-Prozessoren! Zur Laufzeit des Programms ist der Code nämlich schon gemäß der Bedingungen assembliert und unveränderbar. Vielmehr erkennt der Assembler im genannten Fall *während der Assemblierung*, ob ein 80386-Prozessor vorliegt, und erstellt in diesem Fall ein anderes Assemblat, das dann auf 8086ern nicht mehr lauffähig ist (besser: sein kann!).

Clones Dieser Begriff bezeichnet den Nachbau eines elektronischen Bauteils durch andere Firmen. Notwendige Voraussetzung für *Clones* ist, daß sie sich makroskopisch so wie die Originale verhalten, wenn sie auch aus urheberrechtlichen Gründen mikroskopisch dieses Verhalten anders herstellen müssen. Bekannte Clones der Intel-Prozessoren 8086 und 8088 sind die NEC-Prozessoren V20 und V30.

Codesegment Ein Codesegment ist ein Teilbereich des verwendbaren Speichers, der zur Speicherung von Code, also durch den Prozessor ausführbare Befehle, dient. Aufgrund der Verhältnisse beim 8086 kann ein Codesegment maximal 65.536 Bytes groß sein. Die aktuelle Größe des Codesegments kann jedoch auch wesentlich kleiner sein und richtet sich nach der Anzahl und Art der in ihm codierten Prozessorbefehle.

Compilat Das Compilat ist anaolg zum Assemblat das Ergebnis der erfolgreichen Aktivität eines Compilers. Im Prinzip handelt es sich hierbei um etwas absolut Identisches, nämlich eine Datei, die neben ausführbarem Code und Daten auch diverse Listen und weitere Informationen beinhaltet, die der Linker zur Erstellung eines lauffähigen Programms benötigt. Die verbale Unterscheidung zwischen Assemblat und Compilat spiegelt also lediglich die unterschiedliche Quelle des Resultats wider. Dennoch sei angemerkt, daß einige Compiler, wie z.B. Turbo Pascal, neben der eigentlichen Aufgabe des Compilierens auch die Aktivität des Linkers wahrnehmen und Compilate nur intermediär erzeugen! Bei diesen »Compilern« ist das Ergebnis das direkt ausführbare Programm.

Datenbus Analog zur Summe aller Adreßleitungen eines Prozessors, die man als Adreßbus bezeichnet, stellt der Datenbus die Summe aller Datenleitungen dar. Über den Datenbus kann dann aus einer Speicherstelle, die mittels des Adreßbusses angewählt wurde, ein Datum ausgelesen oder in sie geschrieben werden. Gemäß der Breite grundlegender Daten gibt es 8-Bit-Datenbusse, mit denen parallel und somit gleichzeitig 8 Bits (= 1 Byte) ausgelesen oder geschrieben werden können (bei den 8088- und 80188-Prozessoren!), 16-Bit-Datenbusse (16 Bits = 2 Bytes = 1 Wort parallel; 8086, 80186 und 80286) sowie 32-Bit-Datenbusse (32 Bits = 4 Bytes = 1 DWort; 80386, 80486 und Pentium). Eine sinnvolle Ausrichtung (*Alignment*) des Datensegments kann hierbei von großem Nutzen sein: byteweise bei 8-Bit-Datenbussen, wortweise bei 16-Bit- und dwortweise bei 32-Bit-Datenbussen.

Datenport Der Datenport bildet zusammen mit dem Steuerport eine Möglichkeit, gezielt andere (»Peripherie-«) Bausteine des Computers ansprechen zu können, wie z.B. die Chips zur Bildschirmansteuerung oder die seriellen und parallelen Schnittstellen. Hierzu muß dem Prozessor die Fähigkeit gegeben werden, Steuerbefehle und Daten an diese Bausteine abzugeben oder von ihnen zu erhalten. Da dies keine Frage des RAMs oder ROMs ist, spielen hierbei Daten- und Adreßbus *keine* Rolle, da über diese Wege lediglich mit dem Speicher kommuniziert werden kann. Mit den Daten- und Steuerports ist aber eine vergleichbare, wenn auch anders ablaufende Kommunikation möglich: Über die Datenports kann der Prozessor, nachdem er mittels des Steuerports den gewünschten Chip entsprechend programmiert hat, Daten an ihn senden oder von ihm empfangen.

Datensegment Als Datensegment bezeichnet man analog zum Codesegment einen Speicherbereich, in dem der Prozessor Daten ablegen und verwalten kann, die bei der Abarbeitung eines Programms erforderlich sind. Vgl. Codesegment.

Deskriptoren Deskriptoren sind im Protected-Mode des Prozessors Strukturen, in denen bestimmte Daten abgelegt werden, die der Prozessor bei der Abarbeitung von Programmen benötigt. In solchen Deskriptoren befinden sich beispielsweise Angaben zu den verschiedenen Segmenten eines Programms und über die Zugriffsberechtigung. Die Adressen solcher Deskriptoren werden in den Segmentregistern des Prozessors verwaltet.

Disassemblat Als Disassemblat bezeichnet man die »Rückassemblierung« eines assemblierten Quelltextes. Da nämlich das Assemblat aufgrund des komplexen Aufbaus aus verschiedensten Informationen und wegen seines grundsätzlich byteorientierten Charakters schlecht lesbar ist, wird in Programmen, mit denen man assemblierte Dateien zu lesen versucht (den Debuggern), ein Disassemblat hergestellt, das diese Informationen wieder in für Menschen besser lesbare *Mnemonics* zurückverwandelt. ACHTUNG: Da Disassemblate lediglich die »Rückübersetzung« assemblierten Quellcodes sind, können lediglich die Informationen verwendet werden, die auch in assemblierter Form vorliegen. So gehen z.B. alle Informationen während des Assemblierens verloren, mit denen der Assembler gesteuert wird

(Assembleranweisungen!). Daher können in Disassemblaten z.B. keine Zusammenhänge zu Symbolen oder Strukturen mehr hergestellt werden, weshalb sie dort auch nicht mehr auftreten. Daß einige moderne Debugger dies dennoch können, liegt lediglich in der Tatsache begründet, daß die korrespondierenden Assembler der assemblierten Datei weitergehende Informationen mitgeben, die der Linker oder das Hochsprachenmodul *nicht* benötigen und nur für Debugger gedacht sind! Zusammen mit dem eventuell noch vorhandenen Quelltext läßt sich auf diese Weise ein recht komfortabel zu lesendes Disassemblat herstellen, wenn Assembler und Debugger die zusätzlichen Informationen gleich interpretieren.

Extrasegment Als Extrasegment bezeichnet man ein Segment, in dem analog zum Code- und Datensegment Informationen stehen, die der Prozessor bei der Abarbeitung eines Programms benötigt. Die Adresse dieses Segments steht im Segmentregister ES des Prozessors. In der Regel handelt es sich um ein weiteres Datensegment. Vgl. Datensegment.

Flag Ein FLag ist ein Schalter: Es kann gesetzt (»an«) sein oder gelöscht (»aus«). Mit diesen beiden Zuständen, die in einem Bit gespeichert werden können, kann der Prozessor bestimmte Situationen kenntlich machen oder der Programmierer den Prozessor zu bestimmtem Verhalten bewegen. Mit Hilfe der Flags, die der Prozessor setzt und/oder auswertet, läßt sich ein Programm in Abhängigkeit von der herrschenden Situation steuern.

Flat [memory] model Das flat [memory] model ist ein Begriff, der mit dem 80386 eng verknüpft ist. Voraussetzung für das »flache Modell« ist, daß die Adressen, die in diesem Modell verwendet werden, linear sind. Dies heißt, daß sie nicht durch mathematische Manipulation zweier Adreßanteile (Segment und Offset) zusammengesetzt werden müssen, wie das bei Speichersegmentierung der Fall ist. Im flat [memory] model sind daher die Adressen ein-eindeutig, während bei Speichersegmentierung mehrere Adressen auf ein und dieselbe physikalische Adresse zeigen können. Diese Linearität der Adressen ist erst ab den 80386-Prozessoren möglich, da diese mit ihren 32-Bit-Registern jede der über die 32 Adreßleitungen ansprechbare physikalische Adresse in ihren Registern linear abbilden können. Analog zu den Verhältnissen innerhalb eines Segments, in dem alle Adressen logischerweise *near* sind, sind auch im flat [memory] model alle Adressen des physikalisch verfügbaren Adreßraums *near*.

Hardware-Interrupts Hardware-Interrupts sind Ereignisse, mit denen periphere Bausteine dem Prozessor kundtun, daß eine Situation herrscht, in der die spezifische Aufmerksamkeit des Prozessors erforderlich wird. Dies kann z.B. das Drücken einer Taste, das Bewegen der Maus oder das Ankommen eines Datums an der seriellen Schnittstelle sein, wenn ein Modem Daten erhält. Oder es ist ein Zeitsignal des Timerbausteins, der dadurch den Prozessor zu regelmäßigen Aktivitäten veranlaßt. In allen Fällen signalisiert der entsprechende Baustein dem Interrupt-Controller, daß er den Prozessor bei seiner aktuellen Tätigkeit unterbrechen soll. Dieser sammelt alle

anliegenden Interrupt-Anforderungen, ordnet sie nach Priorität und aktiviert den Prozessor, der seinerseits anhand einer zusätzlich übergebenen Adresse die Programme abarbeitet, die dem Ereignis entsprechend zur Klärung der Situation notwendig sind. Anschließend fährt der Prozessor mit der unterbrochenen Tätigkeit fort.

High Byte Als H*igh Byte* wird das »obere« Byte eines Wortes bezeichnet, also das Byte, das die Bits 15 bis 8 des Wortes widerspiegeln.

IDE IDE ist die Abkürzung für *Integrated Development Environment*, also für die integrierte Entwicklungsumgebung.

Integrierte Entwicklungsumgebung Die integrierte Entwicklungsumgebung ist ein Programm, aus dem heraus alle Aktivitäten möglich sind, die bei der Entwicklung eines Programms notwendig werden. Dies ist neben der Erstellung des Quelltexts mittels sogenannter Editoren auch die Übersetzung dieses Quelltexts in Maschinensprache, also die eigentliche Assemblierung oder Kompilierung. Ferner wird häufig auch die Fehlersuche durch Integration eines Debuggers erleichtert. Manche IDEs erlauben darüber hinaus auch das Einbinden anderer Programme, die bei der Programmentwicklung hilfreich sein können, wie z.B. Profiler. Typischerweise lassen sich in IDEs hochkomplexe Vorgänge wie Kompilieren, Linken und Ausführen eines Programms und dessen Debuggen durch simplen »Knopfdruck« realisieren.

Integrierter Assembler Als integierte Assembler (*Inline Assembler*) werden die Assembler bezeichnet, die fest in eine Hochsprache eingebaut sind und es erlauben, im Hochsprachentext Assemblerbefehle zu benutzen. Integrierte Assembler werden von den jeweiligen Hochsprachencompilern bei Bedarf automatisch aktiviert.

Kommutativ Als kommutativ (vertauschbar) bezeichnet man Operationen, bei denen das gleiche Ergebnis entsteht, wenn man die Reihenfolge der Operanden vertauscht. Additionen sind z.B. kommutative Operationen, da es einerlei ist, ob 3 zu 2 addiert wird oder 2 zu 3; das Ergebnis ist in beiden Fällen das gleiche. Subtraktionen dagegen sind nicht kommutativ, da das Abziehen von 2 von 3 offensichtlich ein anderes Ergebnis hat als das Abziehen von 3 von 2.

Least Significant Bit Das niedrigstsignifikante Bit einer Zahl ist immer das mit der niedrigsten Bitposition, also Bit 0.

Linker Der Linker ist ein eigenständiges Programm, das jedoch manchmal (z.B. bei Turbo Pascal) auch Teil eines »Compilers« sein kann (der Name Compiler ist dann strenggenommen falsch!). Er benutzt die Informationen, die der Compiler/Assembler in den sog. OBJ-Dateien erstellt hat, um aus ihnen ablauffähige Programme zu erstellen. Hierzu gehört, daß er Bezüge einzelner Module zueinander herstellt, also sog. Referenzen auflöst. Dies heißt, daß er an die Stellen eines Moduls, an denen Aufrufe »modulfremder« Programmteile erfolgen, die entsprechenden

korrekten Adressen einträgt. Er verbindet (*linkt*) damit verschiedene Module zu einem gemeinsamen Programm. Eine weitere Aufgabe des Linkers ist, dem Programm eine Struktur voranzustellen, in der das jeweilige Betriebssystem Informationen vorfindet, die es benötigt, um das Programm zu starten, wie z.B. die Startadresse.

Low Byte Das *Low Byte* repräsentiert die Bits 7 bis 0 eines Wortes, also dessen »unteres« Byte.

Microcode Als *Microcode* bezeichnet man chipinterne Befehle, die den Chip in die Lage versetzen, einen Prozessorbefehl auszuführen. So wie nämlich Programme aus Prozessorbefehlen aufgebaut sind, sind diese Befehle aus noch elementareren Strukturen, eben dem *Microcode* zusammengesetzt. Zugriff auf den *Microcode* hat man nicht, da dieser die fest verdrahteten Verbindungen unterschiedlicher Transistoren, Widerstände und Kondensatoren des Chips repräsentiert.

Mnemonics sind die verbalen Äquivalente der Opcodes. Ein Prozessorbefehl setzt sich aus einem, häufig jedoch auch mehr als einem Byte zusammen. Diese Bytefolge, die den Prozessor eine ganz bestimmte, genau definierte Aktion ausführen läßt, beispielsweise die Addition der Inhalte zweier Rechenregister, nennt man Opcode (Operation Code). Da es nichts anderes als Bytes, also Zahlenwerte sind, sind Opcodes schwer zu handhaben. Erschwerend kommt hinzu, daß teilweise unterschiedliche Opcodes gleiche Aktivitäten zur Folge haben, lediglich mit anderen Daten. Auch die Angabe von Adressen von Operanden in Zahlenform trägt nicht zur besseren Verständlichkeit bei. Daher ordnete man den Opcodes sog. *Mnemonics* zu. Diese *Mnemonics* sind verbale Abkürzungen, die üblicherweise einprägsam die Art der Operation wiederspiegeln: ADD. *Mnemonics* sind also nichts anderes als die (lesbaren) Namen der Opcodes, in die der Assembler sie übersetzt.

Most Significant Bit Das höchstsignifikante Bit einer Zahl ist immer das mit der höchsten Bitposition. In Bytes ist es Bit 7, in Worten Bit 15, in Doppelworten Bit 31. Bei Realzahlen unterscheidet man die MSBs (***M**ost **S**ignificant **B**its*) der Exponenten und der Mantisse.

NaN Eine ***N**ot **a** **N**umber* ist eine »Realzahl«, deren Exponent signalisiert, daß die zugrundeliegende Bytefolge keine Zahl repräsentiert. NaNs sind somit Codes, die irgendeine Besonderheit widerspiegeln sollen. Sie entstehen z.B., wenn man Berechnungen mit leeren Registern durchführen will, durch 0 dividiert oder Unendlichkeiten erzeugt. Man unterscheidet mehrere Arten von NaNs. Allen gemeinsam ist ein Exponent mit Betrag 0, aber negativem Vorzeichen (dann sind alle Bits des Exponenten gesetzt!). Bei den sog. *quiet NaNs* oder *qNaNs* ist neben dem *MSB* der Mantisse, das die Vorkommastelle »1.« in TEMPREALs repräsentiert, immer auch das unmittelbar folgende Bit gesetzt. Die Bitfolge der Mantisse lautet in diesem Fall 1.11…1 bis 1.10…0. Ist dieses Bit dagegen gelöscht, wie im Bereich 1.01…1 bis 1.00…1, so spricht man von *Signaling-NaNs* oder *sNaNs*. Die verbliebene letzte Bitkombi-

nation mit negativem Null-Exponenten, 1.00…0, ist die spezielle NaN *Infinity*, also Unendlichkeit. Da jede Mantisse zusätzlich noch ein Vorzeichenbit besitzt, können alle eben angesprochenen NaNs sowohl in positiver als auch negativer Form vorliegen. Bei den TEMPREALs, bei denen das MSB der Mantisse ja explizit angegeben wird, sei noch ergänzt, daß dieses auch 0 werden kann. Die daraus resultierenden NaNs werden als *Pseudo-NaNs* bezeichnet. Sie lassen sich ganz analog zu den *echten* NaNs in *qPNaNs* (0.11…1 bis 0.10…0), *sPNaNs* (0.01…1 bis 0.00…1) und *Pseudo-Infinities* (0.00…0) einteilen. Diese Pseudo-NaNs werden nicht von allen Prozessoren unterstützt! Bei den anderen beiden Realzahltypen (SINGLEREAL und DOUBLEREAL) sind nur *echte* NaNs definiert, da hier das MSB der Mantisse nicht explizit angegeben und somit nicht verändert werden kann.

NMI NMIs sind *Non Mascable Interrupts*, also Interrupts, die man nicht maskieren kann. Hierunter versteht man, mittels CLI auf Prozessorseite oder durch Programmierung des Interrupt-Controllers eine Unterbrechung der augenblicklichen Aktivitäten des Prozessors zu unterbinden. NMIs führen somit auf jeden Fall zur Unterbrechung und signalisieren somit eine schwerwiegende Ausnahmesituation, die die unmittelbare Aufmerksamkeit des Prozessors erfordert.

Nibble ist eine häufig benutzte Benennung eines halben Bytes, also von vier konsekutiven Bits. Man unterscheidet analog zu *Low* und *High Byte* ein unteres (Bits 3 bis 0) und oberes (Bits 7 bis 4) Nibble (*Low* und *High Nibble*) des Bytes.

Offset Als Versatz oder Offset bezeichnet man einen Anteil an einer Adresse, der zu einem bestimmten Grundwert addiert werden muß, um die korrekte Adresse zu erhalten. Bei der Speichersegmentierung ist der Offset ein 16-Bit-Wert, der zu der Segmentgrenze addiert werden muß, um die gewünschte Adresse innerhalb des Segments zu erhalten. In Strukturen wie z.B. Tabellen ist der Offset der Wert, der zur Adresse des ersten Tabellenelements addiert werden muß, um die Adresse des gewünschten Tabelleneintrags zu erhalten. Solche Tabellenoffsets sind in der Regel leicht aus dem Tabellenindex errechenbar: Es braucht lediglich der Index mit der Größe der Tabelleneinträge multipliziert zu werden, um den Offset zu erhalten. Wenn z.B. eine Tabelle aus Bytes besteht, so ist der Index mit dem Offset identisch. Bei DWord-Tabellen muß der Index mit vier (DWord = 4 Bytes) multipliziert werden, um den Offset zu erhalten.

Opcode Der Opcode oder *Operation Code* ist die Byte-Sequenz, die einem genau definierten Prozessorbefehl entspricht. Er kann aus einem oder mehreren Bytes bestehen. Teil eines Opcodes ist immer ein Code für den Befehl, der ggf. um Adressen von Operanden ergänzt sein kann, die der Befehl benutzt. Bestimmte Präfixe können ebenfalls Teil des Opcodes sein, so z.B. das Präfix *AdrSiz:*, das im Flat-Model die Benutzung von 32-Bit-Adressen erzwingt, oder Segmentpräfixe wie *CS:* oder *ES:*, die für die explizite Verwendung anderer als der Standardsegmente notwendig sind.

Pipelinig Das *Pipelinig* ist ein Begriff, der mit dem Pentium und seiner neuen Floating-Point-Unit, also dem »Coprozessor« verknüpft ist. Hier, wie auch allgemein, versteht man darunter, daß etwas »über eine *Pipeline*« abgearbeitet wird. Hintergedanke hierbei ist, daß in dieser »Röhre« an verschiedenen Punkten Handlungen vorgenommen werden, die in ihrer Summe und gemäß ihrer Reihenfolge das Gewünschte erzeugen. Anschaulich läßt sich das Pipelining mit einem Fließband erklären, an dem alle Teile z.B. eines Autos in spezifischer Art und Reihenfolge über mehrere Zwischenschritte zusammengefügt werden, bis »hinten« das fertige Produkt herauskommt. Der Vorteil beim Pipelinig ist, daß die einzelnen »Zwischenstufen« unabhängig voneinander arbeiten können, solange sie »im Takt« bleiben. Dies bedeutet z.B. für die Floating-Point-Unit, daß ihre Komponenten sich unmittelbar nach der Bearbeitung des einen Datums mit der Bearbeitung des nächsten beschäftigen können. Möglich wird dies, da die (jeweils halbfertigen) Zwischenprodukte die Pipeline nicht verlassen können, also seriell strömen. Resultat des Pipelinings ist, daß die Ressourcen weitaus effektiver genutzt werden können: Die gesamte *Unit* kann mehrere Daten taktversetzt gleichzeitig bearbeiten, so wie sich auf dem Fließband immer mehrere Autos in unterschiedlichem Montagezustand befinden. Dies führt zu höherer Performance.

Port ist der allgemeine Begriff für einen Daten- und/oder Steuerport.

Projektives Modell Beim projektiven Modell der Zahlentheorie liegen alle Zahlen auf einem Zahlenkreis, den man sich als geschlossenen Zahlenstrahl des affinen Modells vorstellen kann. Hier wird der Kreis an den Extremwerten $-\infty$ und $+\infty$ geschlossen. Sie werden somit als identisch behandelt.

Protected-Mode Der Protected-Mode ist einer von mehreren (beim 80286: zwei, beim 80386 und 80486: drei und beim Pentium: vier) Betriebszuständen des Prozessors. Im Unterschied zum Real-Mode läßt der Protected-Mode eine neue, unbegrenzte Adressierung des Speicherbereichs zu. Ferner bietet sich die Möglichkeit, bestimmte Schutzfunktionen zu aktivieren, wie z.B. den Schreibschutz auf einzelne Segmente (daher auch der Name). Programme für den Protected-Mode laufen nicht im Real-Mode!

RAM Der *Random Access Memory* ist der beschreibbare Speicher, in dem sich Daten und Programme befinden, wenn sie abgearbeitet werden.

Real-Mode Der Real-Mode ist der allen Prozessoren der Intel-Baureihe und deren Clones innewohnende »Grundbetriebszustand« des Prozessors. Seine Besonderheiten sind die magische Grenze des Speichers von 1 MByte sowie das Fehlen jeglichen Schutzkonzeptes. Im Real-Mode können daher nicht mehrere Programme gleichzeitig nebeneinander ablaufen.

Register Als Register bezeichnet man einen Bereich auf dem Prozessorchip, den dieser für das temporäre Speichern von Daten nutzt. In den Registern laufen die Berechnungen des Prozessors ab, dort hält er wichtige Adressen und Informationen.

Segmentgrenze Als Segmentgrenze bezeichnet man eine Speicherstelle, deren physikalische Adresse durch 16 ohne Restbildung teilbar ist. An diesen Stellen können Segmente beginnen oder aufhören.

Segmentregister Ein Segmentregister ist ein Register, in dem der Prozessor die Adresse eines Segments, also dessen Segmentgrenze, verzeichnet. Solche Segmentregister werden bei der Berechnung von physikalischen Adressen verwendet, wo immer diese benötigt werden. Der Prozessor kennt als wichtigste Segmentregister das Code-, Stack- und Datensegmentregister. Schließlich gibt es noch ein Extra-Segmentregister sowie bei machen Prozessoren noch weitere Segmentregister.

Stack Der Stapel oder Stack ist eine bestimmte Struktur im Speicher, in der der Prozessor wichtige Daten wie Rücksprungadressen und Flagzustände temporär speichern kann. Der Stack dient ferner zur Datenübergabe an Routinen sowie zur temporären Nutzung für lokale Variablen. Nicht zu verwechseln ist der Stack mit dem Coprozessorstack! Bei diesem handelt es sich um einen Satz (Stapel) von Registern, in denen die Berechnungen des Coprozessors ablaufen.

Stacksegment Segmentgrenze, an der der Stack beginnt. Vgl. auch Datensegment und Codesegment.

Stapel siehe Stack.

Steuerport Der Steuerport bildet zusammen mit dem Datenport eine Möglichkeit, gezielt andere (»Peripherie-«) Bausteine des Computers ansprechen zu können, wie z.B. die Chips zur Bildschirmansteuereung oder die seriellen und parallelen Schnittstellen. Hierzu muß dem Prozessor die Fähigkeit gegeben werden, Steuerbefehle und Daten an diese Bausteine abzugeben oder von ihnen zu erhalten. Da dies keine Frage des RAMs oder ROMs ist, spielen hierbei Daten- und Adreßbus *keine* Rolle, da über diese Wege lediglich mit dem Speicher kommuniziert werden kann. Mit den Daten- und Steuerports ist aber eine vergleichbare, wenn auch anders ablaufende Kommunikation möglich: Über die Datenports kann der Prozessor, nachdem er mittels des Steuerports den gewünschten Chip entsprechend programmiert hat, Daten an ihn senden oder von ihm empfangen.

Takt Der Takt ist der Rhythmus, mit dem Prozessor und Coprozessor bestimmte Befehle abarbeiten. Da die meisten dieser Befehle sehr komplex sind (für den Prozessor!), müssen sie in Portionen hintereinander ausgeführt werden. Damit es hierbei nicht zu Problemen kommt, erfolgt die Abarbeitung dieser »Pakete« in einem Rhythmus, dem Takt, den ein Quartz selbst generiert. Da genau bekannt ist, aus wie vielen Paketen ein Befehl besteht, läßt sich der Takt somit dazu benutzen, die Ausführungsdauer eines Befehls anzugeben. Man verwendet diese Art der Angabe der Dauer eines Befehls, da sie unabhängig von der Prozessorgeschwindigkeit ist: Ein bestimmter Befehl hat immer die gleiche Anzahl von Takten, solange der Prozessortyp der gleiche ist. Welche Zeitspanne die Bearbeitung eines Befehls dann tat-

sächlich benötigt, läßt sich durch Multiplikation der Taktzahl des Befehls mit dem Kehrwert des Prozessortaktes, der Taktdauer, ermitteln.

Task Als *Task* wird in Multitasking-Betriebssystemen ein Programm bezeichnet, das parallel zu anderen quasi gleichzeitig ausgeführt wird.

Timerbaustein Der Timerbaustein ist ein Chip, der die Aktivität eines Quartzes dazu nutzt, bestimmte Taktraten zu generieren. Diese werden von vielen Chips zur Synchronisation nicht nur ihrer Aktivitäten, sondern auch zur Synchronisation mit anderen Bausteinen benutzt. Eine wesentliche Aufgabe des Timerbausteins ist die Herstellung eines Systemtaktes (für DOS) sowie die Auffrischung des Speichers (*refresh*). Ferner dient der Timerbaustein auch heute noch zur Tonerzeugung.

TOS Der *Top-of-Stack* ist das jeweils aktuelle Register im Coprozessorstack. In diesem Register laufen alle Berechnungen des Coprozessors ab und über dieses Register kommuniziert der Coprozessor mit dem Speicher. TOS kann je nach vorangegangenen Befehlen jedes der acht physikalischen Register des Stacks sein! Welches zu einem bestimmten Zeitpunkt der TOS ist, wird im Statuswort des Coprozessors verzeichnet.

Type-Casting Typenumwandlung. Als Type-Casting bezeichnet man das Überführen eines Datums eines bestimmten Typs (z.B. Integer) in einen anderen Typ (z.B. Byte).

Virtual-8086-Mode Der *Virtual-8086-Mode* ist eine Errungenschaft des 80386 und ermöglicht diesem, den adressierbaren Speicherbereich in mehrere Blöcke zu jeweils 1 MByte zu unterteilen, die dem Adreßraum eines 8086 entsprechen. In diesem Modus kann der 80386 also mehrere 8086 emulieren, sie also virtuell darstellen.

E Die Routinen der Routinensammlung Mathe

Die Routinensammlung Mathe ist eine hauptsächlich in Assembler geschriebene Bibliothek mit mächtigen mathematischen Routinen. Auf eine detaillierte Beschreibung aller Routinen dieser Sammlung kann verzichtet werden, da besonders wichtige, trickreiche oder aussagekräfigte Passagen und Module an anderer Stelle dieses Buches ausführlich besprochen wurden. Alle weiteren Teile dieser Sammlung sind aufgrund der kurzen, aber ausreichenden Kommentierung selbsterklärend.

Sie können die Bibliothek Mathe bedenkenlos in Ihre Hochsprachenmodule einbauen, ohne tiefgreifende Kenntnisse in Assembler zu haben. Sie müssen lediglich darauf achten, daß Sie in Ihrem Hauptprogramm die Coprozessoremulation einbinden (das ist meistens über Compilerschalter einfach zu bewerkstelligen). Denn die Bibliothek selbst nutzt die Möglichkeit, einen Coprozessor zu emulieren. So sind in jedem Fall – ob ein Coprozessor vorhanden ist, oder nicht – alle Routinen dieser Sammlung jederzeit verwendbar! An dieser Stelle folgt eine kurze Beschreibung der Sammlung und ihrer Funktion.

Anzumerken ist noch, daß die Sammlung in *drei* Versionen auf der CD-ROM beiliegt: als Turbo Pascal-Unit MATHE.TPU, als Turbo C++-Bibliothek MATHE_P.LIB und als Visual C++-Bibliothek MATHE_C.LIB mit dazugehöriger *Header*-Datei MATHE_P.H bzw. MATHE_C.H. Natürlich liegen ebenfalls alle Quelltexte vor!

ACHTUNG! MATHE_C.LIB und MATHE_P.LIB verwenden das Speichermodell *small*. Sie müssen daher ggf. die Assembler- und C-Routinen mit dem gewünschten Speichermodell reassemblieren und recompilieren. Beiden C-Dialekten liegt ein *Tool* bei, mit dem Sie die Bibliothek selbst erstellen können (LIB.EXE bzw. TLIB.EXE). Achten Sie dabei darauf, daß

- Sie die richtigen Assembler- und C-Dateien verwenden (die Pascal-konformen MATHE?_P.ASM-Dateien für Turbo C++ und die C-konformen MATHE?_C.ASM für Visual C++!).
- diese Dateien dann auch im richtigen Speichermodell assembliert/compiliert werden. Eventuell müssen Sie mittels Assembler-/Compilerschalter ein Modell einstellen oder den Quelltext um die Anweisung *.MODEL xxx* erweitern, wobei xxx für TINY, SMALL, MEDIUM, COMPACT, LARGE oder HUGE steht.
- bei der Assemblierung ggf. auf *Case-Sensitivity* zu achten ist (bei den Pascalkonformen Versionen muß mittels Assembler-Schalter die *Großschreibung* der öffentlichen Labels eingeschaltet werden, im Falle der C-konformen Versionen auf *Unterscheidung Groß-/Kleinschreibung*!).

Anzumerken ist auch noch:

- Bei der Assemblierung einiger Dateien werden Sie Warnungen des Assemblers erhalten! Dies hat didaktischen Wert, da Sie hierdurch sehen können, daß nicht jede (»illegitime«) Verwendung (eine sog. *Redefinition*) von Schlüsselworten zu Fehlern führt. Die Dateien werden korrekt (!) assembliert.
- Auch bei der Kompilierung von MATHE7_?.C werden Warnungen ausgegeben, dieses Mal durch den Compiler. Auch diese Warnungen können Sie getrost vergessen, da sie lediglich darauf hinweisen, daß *Sin, Cos* etc. keinen Funktionswert zurückgeben, obwohl sie als Funktionen deklariert werden. Dennoch werden auch diese beiden Dateien korrekt compiliert. Warum die genannten Funktionen keinen Funktionswert zurückgeben, haben wir auf Seite 473 angesprochen.

E.1 Neue Typen der Bibliothek Mathe

Mathe definiert folgende Typen:

AngleMode = (Rad, Deg, Grad)

Mit *AngleMode* kann die Bibliothek auf einen bestimmten Winkelmodus eingestellt werden. Alle Winkelfunktionen greifen auf diese Definition zurück. Zur Verfügung stehen die Modi

- *Rad* (*radiant*). In diesem Modus werden Winkelangaben im Bogenmaß erwartet, d.h. ein Kreis besitzt den Umfang 2π. Soll also z.B. der Sinus eines rechten Winkels berechnet werden, so hat dies mit *Sin(π/2)* zu erfolgen, da 90° = π/2.
- *Deg* (*degrees*). Dieser Modus entspricht dem allgemein üblichen Winkelmodus, in dem ein Kreis einen Winkel von 360° (Altgrad!) besitzt. So wird der Cosinus eines rechten Winkels über *Cos(90)* ermittelt.
- *Grad* (*grad*). In diesem Modus besitzt ein Kreis 400° (Neugrad!), so daß der Secans eines rechten Winkels mit *Sec(100)* ermittelt wird.

Die Voreinstellung ist *Rad*.

DispMode = (Fix, Float, Sci, Eng)

Mit *DispMode* kann festgelegt werden, wie die Prozedur *Extract* die übergebene Realzahl zerlegen soll. Es bestehen folgende Möglichkeiten:

- *Fix* (*Fixed Decimal Notation* – Festkommadarstellung). In diesem Modus wird die übergebene Zahl als Festkommazahl mit einer definierbaren Anzahl von Nachkommastellen interpretiert. Hierbei ist folgendes zu beachten:
  - Liegt der Exponent der Zahl unter einer definierbaren Schranke, so erfolgt *keine* Trennung von Mantisse und Exponenten. Die Zahl $1.23 \cdot 10^{11}$ wird dann (bei standardmäßig eingestellter Schranke für den Exponenten von 12) in der Mantisse übergeben (und könnte als 123000000000 dargestellt werden). Der Exponentanteil ist hier 0.
  - Ab dieser Schranke wird in das *Sci*-Format umgeschaltet, so daß $1.23 \cdot 10^{12}$ in die Mantisse 1.23 und den Exponenten +12 zerlegt wird.

- *Float* (*Floating Point Notation* – Fließkommadarstellung). Dies ist praktisch ein *Bypass* der Routine! In diesem Modus wird die übergebene Zahl unverändert zurückgegeben. Er dient lediglich dazu, Fallunterscheidungen der Art

```
If NumberIsToBeExtracted then Extract(X,M,E)
else M := X
```

  im Programm zu vermeiden! Auf diese Weise kann grundsätzlich *Extract(X,M,E)* verwendet werden. Wurde im Modus *float* gearbeitet, enthält *M* den Wert von *X*, andernfalls den überarbeiteten Wert.
- *Sci* (*Scientific Notation* – wissenschaftliche Darstellung). Dieser Modus zerlegt die übergebene Zahl in eine Mantisse und einen Exponenten in wissenschaftlicher Notation. Beispiel: die Zahl 0.0123 wird in die Mantisse 1.23 und den Exponenten -2 zerlegt.
- *Eng* (*Engineer's Notation* – ingenieurwissenschaftliche Darstellung). Auch in diesem Modus wird die Zahl in eine Mantisse und einen Exponenten zerlegt, jedoch ist der Exponent immer ein ganzzahliges Vielfaches von 3. Dieser Modus eignet sich daher besonders gut zur Berechnung von Werten, wenn Einheiten berücksichtigt werden sollen. So wird die Zahl 0.0123 in die Mantisse 12.3 und den Exponenten -3 zerlegt.

RoundMode = (Round, Down, Up, Trunc, Next)

Auch dieser Modus wird von *Extract* verwendet. Mit ihm läßt sich steuern, auf welche Weise gerundet werden soll, falls in einem der Modi *fix, sci* oder *eng* gearbeitet wird. Zur Verfügung stehen

- *Round* (Runden). In diesem Fall wird die Zahl auf die eingestellte Anzahl Nachkommastellen »richtig« gerundet. »Richtig« bedeutet in diesem Zusammenhang, daß der Nachkommateil ab 0.5 (*einschließlich!*) zur nächsten höheren Zahl auf- und bis 0.5 abgerundet wird. Analog werden Nachkommateile ab -0.5 (*einschließlich!*) zur nächsten Zahl ab- und bis -0.5 aufgerundet. Beispiele:

1,499	→	1,000
1,500	→	2,000
-1,499	→	-1,000
-1,500	→	-2,000

- *Next* (Runden zur nächsten Zahl). Next arbeitet wie *Round*, rundet aber »falsch«. »Falsch« bedeutet in diesem Zusammenhang, daß der Nachkommateil ab 0.5 (*ausschließlich!*) zur nächsten höheren Zahl auf- und bis 0.5 abgerundet wird. Analog werden Nachkommateile ab -0.5 (*ausschließlich!*) zur nächsten Zahl ab- und bis -0.5 aufgerundet. Beispiele:

1,501	→	2,000
1,500	→	1,000
-1,501	→	-2,000
-1,500	→	-1,000

- *Down* (Abrunden). Down rundet grundsätzlich ab, d.h. es wird in Richtung -∞ gerundet. Beispiele:

  1,499 → 1,000
  1,501 → 1,000
  -1,499 → -2,000
  -1,501 → -2,000

- *Up* (Aufrunden). Up rundet grundsätzlich auf, d.h. es wird in Richtung +∞ gerundet. Beispiele:

  1,499 → 2,000
  1,501 → 2,000
  -1,499 → -1,000
  -1,501 → -1,000

- *Trunc* (*truncate* – Abschneiden). Hier wird die Zahl mit der spezifizierten Anzahl von Nachkommastellen beschnitten! Beispiele:

  1,499 → 1,000
  1,501 → 1,000
  -1,499 → -1,000
  -1,501 → -1,000

E.2 Die Routinen der Bibliothek Mathe

E.2.1 Turbo Pascal

Die Unit Mathe besitzt vier Routinen, mit denen logarithmische Werte erzeugt werden können. Es handelt sich um den Logarithmus naturalis (Ln), den Logarithmus dualis (Ld), den Logarithmus decalis (Lg) sowie den allgemeinen Logarithmus zu einer beliebigen Basis (Log):

- function Ln(X:extended):extended
- function Ld(X:extended):extended
- function Lg(X:extended):extended
- function Log(X, Base:extended):extended

Auch die »Unkehrfunktionen« sind implementiert:

- function Power_e(X:extended):extended
- function Power_2(X:extended):extended
- function Power_10(X:extended):extended
- function Power(X,Y:extended):extended

Es folgen nun die periodischen Routinen und ihre »Verwaltungsroutinen«, mit denen der aktuelle Winkelmodus eingestellt und abgefragt werden kann:

- procedure SetAngleMode(Mode:AngleMode)
- function GetAngleMode:AngleMode
- function Sin(X:extended):extended

- function Cos(X:extended):extended
- function Tan(X:extended):extended
- function Cot(X:extended):extended
- function Sec(X:extended):extended
- function Csc(X:extended):extended
- function ASin(X:extended):extended
- function ACos(X:extended):extended
- function ATan(X:extended):extended
- function ACot(X:extended):extended
- function ASec(X:extended):extended
- function ACsc(X:extended):extended

Mit *xDiv* und *xMod* lassen sich analog zu *Div* und *Mod* auch Realzahlen bearbeiten. Das bedeutet, daß *xDiv* analog zur Integervariante *Div* das (ganzzahlige) Divisionsergebnis der Division X / Y berechnet, während *xMod* den dazugehörigen (gebrochenen!) Divisionsrest bildet. *xInt* und *xFrac* sind Nachbildungen der Funktionen *Int* und *Frac*, die jedoch auf Realzahlen mit hoher Genauigkeit (*Extended*) ausgerichtet sind. So liefert *xInt* im Gegensatz zum Original den Vorkommateil (absichtlich!) als Realzahl.

- funcktion xInt(X:extended):extended
- function xFrac(X:extended):extended
- function xDiv(X,Y:extended):extended
- function xMod(X,Y:extended):extended

Beispiel: xDiv(9.87654321, 2.345) ergibt 4.00000000; xMod(9.87654321, 2.345) führt zu 0.49654321. Das Zusammensetzen dieser Werte nach 4.00000000·2.345+0.49654321 ergibt selbstverständlich wieder 9.87654321!

Die folgende Routinen ermöglichen die Aufbereitung einer Realzahl:

- procedure Extract(X:extended;var Mant:extended; var Exp:integer)
- function xRound(X:extended):extended
- procedure SetDisplayMode(Mode:DispMode; Digits:Byte)
- procedure GetDisplayMode(var Mode:DispMode; var Digits:Byte)
- procedure SetRoundMode(Mode:RoundMode)
- function GetRoundMode:RoundMode
- procedure SetMaxFixDisplay(Exp:Integer);
- function GetMaxFixDisplay:Integer;

SetDisplayMode setzt hierbei den Modus, wie die Realzahl zerlegt werden soll und definiert auch die Anzahl von Nachkommastellen, die ggf. zu berücksichtigen sind. *SetRoundMode* steuert die Art der Rundung. Der Prozedur *Extract* wird die zu zerlegende Realzahl in X übergeben. Sie liefert die aufbereitete Mantisse in *Mant* und den dazugehörigen Exponenten in *Exp* zurück. Einfacher, aber ebenso hilfreich, arbeitet *xRound*: Diese Funktion rundet einfach die ihr übergebene Zahl anhand der mit *SetDisplayMode* und *SetRoundMode* gesetzten Parameter. Sie ersetzt somit die Standardfunktionen *Round* und *Trunc*, die lediglich auf ganze Zahlen runden bzw. ab-

schneiden. Die letzten beiden Routinen dienen dazu, den größten Exponenten festzulegen und festzustellen, den *Extract* im Modus *Fix* benutzt. Es sind positive und negative Exponenten erlaubt!

E.2.2 C

Die gleichen Funktionen und Prozeduren existieren für die beiden C-Dialekte. Dennoch gibt es hier Unterschiede! Da Visual C++ etwas eigene Wege bei der Übergabe von Parametern geht, wenn Fließkommazahlen im Spiel sind, existieren mit MATHE?_C.ASM Assemblerdateien und mit MATHE_C.LIB eine Bibliothek, die die C-Namens- und Übergabekonvention benutzen. Hierzu gehört dann die *Header*-Datei MATHE_C.H. Natürlich sind diese Dateien auch unter Turbo C++ verwendbar – dennoch liegt mit MATHE_P.H und MATHE_P.LIB auch eine speziell für Turbo C++ entwickelte Header-Datei samt Bibliothek vor, die die Pascal-Konventionen benutzt und somit die Pascal-Assemblerdateien MATHE?_P.ASM einbindet.

Anzufügen ist auch noch, daß die Programmteile, die neben der Deklaration der Prototypen und der Definition von Typen in typischer Turbo Pascal-Manier in der Unit existieren, unter C in eigene Dateien ausgelagert wurden. Sie heißen MATHE7_C.C und MATHE7_P.C.

Die Funktionen, die in beiden Bibliotheken verzeichnet sind, gleichen den aus MATHE.TPU für Turbo Pascal aufs Haar. Daher sollen sie nicht mehr besprochen, sondern an dieser Stelle nur noch die Prototypen, die C verwendet, vorgestellt werden:

Turbo C++:

```
:#ifndef __BORLANDC__
  #error Kein TURBO C / C++
#endif

enum AngleMode {Rad,Deg,Grad};
enum DispMode  {Fix,Float,Sci,Eng};
enum RoundMode {Round,Down,Up,Trunc,Next};

extern long double far pascal Ln(long double X);
extern long double far pascal Ld(long double X);
extern long double far pascal Lg(long double X);
extern long double far pascal Log(long double X,
                                  long double Base);

extern long double far pascal Power_e(long double X);
extern long double far pascal Power_2(long double X);
extern long double far pascal Power_10(long double X);
extern long double far pascal Power(long double X,
                                    long double Y);
```

```
extern void SetAngleMode(enum AngleMode Mode);
extern enum AngleMode GetAngleMode(void);

extern long double far Sin(long double X);
extern long double far Cos(long double X);
extern long double far Tan(long double X);
extern long double far Cot(long double X);
extern long double far Sec(long double X);
extern long double far Csc(long double X);

extern long double far ASin(long double X);
extern long double far ACos(long double X);
extern long double far ATan(long double X);
extern long double far ACot(long double X);
extern long double far ASec(long double X);
extern long double far ACsc(long double X);

extern long double far pascal xDiv(long double X,
                                   long double Y);
extern long double far pascal xMod(long double X,
                                   long double Y);
extern long double far pascal xInt(long double X);
extern long double far pascal xFrac(long double X);

extern void far pascal Extract(long double X,
                               long double far *Mant
                               signed int far *Exp);
extern long double far pascal xRound(long double X);
extern void SetDisplayMode(enum DispMode Mode,
                           unsigned char Digits);
extern void GetDisplayMode(enum DispMode far *Mode,
                           unsigned char far *Digits);
extern void SetRoundMode(enum RoundMode Mode);
extern enum RoundMode GetRoundMode(void);
extern void SetMaxFixDisplay(signed int Exp);
extern signed int GetMaxFixDisplay(void);

extern signed int MathError(void);
```

Beachten Sie bei der Betrachtung dieser *Header*-Datei bitte folgendes:

- Wegen der unterschiedlichen Behandlung von Funktionsergebnissen des Typs *long double* in Visual C++ und Turbo C++ erfolgt der Abbruch der Kompilierung, falls versucht wird, die Header-Datei in Nicht-Turbo C-Programme einzubauen.

- Mit dem Schlüsselwort *pascal* werden nur Routinen versehen, die auch tatsächlich mit den Pascal-Namens- und Übergabekonventionen aufgerufen werden müssen. Das sind alle Routinen, die in den Assemblermodulen stehen, nicht aber die, die in MATHE7_P.C in C realisiert sind! Da *Sin, Cos* etc. nur »Laderoutinen« für die Assemblerroutine *Trans* sind, wurden sie, analog zur Unit Mathe für Turbo Pascal, als Hochsprachenroutinen definiert. Diese Routinen folgen selbstverständlich den C-Konventionen!
- Alle in Assemblermodulen realisierten Routinen werden als *far* deklariert. Dies dient der Übereinstimmung mit den *far*-Deklarationen in den jeweiligen Assembler-Dateien.

Da sie im Zusammenhang mit der Einbindung von Assemblerroutinen wichtig sind, sollen an dieser Stelle auch kurz wichtige Details aus der Datei MATHE7_P.C angesprochen werden:

```
extern void far pascal Trans(unsigned short A,
                             unsigned short B,
                             unsigned short C,
                             unsigned short D);
extern void far pascal ATrans(unsigned short A);

const unsigned int     pascal K_200    = 200;
const unsigned int     pascal K_180    = 180;
const unsigned int     pascal K_4      =   4;
const float            pascal K_05     =   0.5;
const unsigned int     pascal K_10     =  10;
      enum AngleMode   pascal M_Angle_ = Rad;
      enum DispMode    pascal M_Disp_  = Float;
      float            pascal M_Digits_= 100.0;
      enum RoundMode   pascal M_Round_ = Round;
const float            pascal Epsilon  = 2.0E-17;
      float            pascal M_Fix_   = 1.0E12;
      signed int       pascal M_Error_ =   0;
```

- *Trans* und *ATrans* sind zwar global deklariert, also öffentlich, sollen jedoch nicht von C-Programmen aus aufgerufen werden. Daher stehen sie *nicht* in der Header-Datei und müssen somit hier ihren Prototyp erhalten, damit die in diesem Modul deklarierten Routinen *Sin, Cos* etc. auf sie zurückgreifen können.
- Auf alle hier deklarierten Daten greifen die Assemblerroutinen zurück. Dies erfolgt mittels Pascal-Namenskonvention. Daher müssen auch in dieser Datei die Pascalschen Konventionen bei der Deklaration der Daten berücksichtigt werden, weshalb sie mit dem Schlüsselwort *pascal* versehen sind.

Alles andere in dieser Datei dürfte einem C-Programmierer keine ernsthaften Schwierigkeiten bereiten. Sie finden hier auch noch zwei Beispiele für die Nutzung des integrierten Assemblers.

Visual C++:

Die Header-Datei sieht ganz analog zu der aus Turbo C++ aus, nur daß, da die C-Versionen der ASM-Dateien verwendet werden, keinerlei Deklaration mit *pascal* erfolgt:

```
enum AngleMode {Rad,Deg,Grad};
enum DispMode  {Fix,Float,Sci,Eng};
enum RoundMode {Round,Down,Up,Trunc,Next};

extern long double far Ln(long double X);
extern long double far Ld(long double X);
extern long double far Lg(long double X);
extern long double far Log(long double X, long double Base);

extern long double far Power_e(long double X);
extern long double far Power_2(long double X);
extern long double far Power_10(long double X);
extern long double far Power(long double X, long double Y);

extern void SetAngleMode(enum AngleMode Mode);
extern enum AngleMode GetAngleMode(void);

extern long double far Sin(long double X);
extern long double far Cos(long double X);
extern long double far Tan(long double X);
extern long double far Cot(long double X);
extern long double far Sec(long double X);
extern long double far Csc(long double X);

extern long double far ASin(long double X);
extern long double far ACos(long double X);
extern long double far ATan(long double X);
extern long double far ACot(long double X);
extern long double far ASec(long double X);
extern long double far ACsc(long double X);

extern long double far xDiv(long double X, long double Y);
extern long double far xMod(long double X, long double Y);
extern long double far xInt(long double X);
extern long double far xFrac(long double X);

extern void far Extract(long double X,
                        long double far *Mant,
                        signed int far *Exp);
extern long double far xRound(long double X);
```

```
extern void SetDisplayMode(enum DispMode Mode,
                           unsigned char Digits);
extern void GetDisplayMode(enum DispMode far *Mode,
                           unsigned char far *Digits);
extern void SetRoundMode(enum RoundMode Mode);
extern enum RoundMode GetRoundMode(void);
extern void SetMaxFixDisplay(signed int Exp);
extern signed int GetMaxFixDisplay(void);

extern signed int MathError(void);
```

Dementsprechend fehlt auch die Pascal-Definition der Daten im MATHE7_C.C:

```
extern void far Trans(unsigned short A, unsigned short B,
                      unsigned short C, unsigned short D);
extern void far ATrans(unsigned short A);

const unsigned int     K_200    = 200;
const unsigned int     K_180    = 180;
const unsigned int     K_4      =   4;
const float            K_05     =   0.5;
const unsigned int     K_10     =  10;
      enum AngleMode   M_Angle_ = Rad;
      enum DispMode    M_Disp_  = Float;
      float            M_Digits_= 100.0;
      enum RoundMode   M_Round_ = Round;
const float            Epsilon  = 2.0E-17;
      float            M_Fix_   = 1.0E12;
      signed int       M_Error_ =   0;
```

Noch ein Hinweis: Damit diese Datei auch mit Turbo C++ kompiliert werden kann, wurde auf die bedingte Kompilierung zurückgegriffen. Denn wie in MATHE7_P.C auch, wurden zwei Routinen mit dem integrierten Assembler realisiert. Und wie Sie ja wissen, haben die beiden *Inline Assembler* von Borland und Microsoft kleine, aber feine Eigenheiten …

E.3 Die Fehlerbearbeitung von Mathe

Mit *MathError* läßt sich feststellen, ob und, wenn ja, was für ein Fehler bei der zuletzt ausgeführten Routine der Bibliothek Mathe aufgetreten ist. Diese Funktion existiert als einzige Routine der Sammlung in zwei Formen.

Turbo Pascal: Die Unit Mathe definiert eine Pascal-Funktion im Implementierungsteil gemäß

```
function  MathError:integer
```

Visual C++, Turbo C++: Bei beiden C-Versionen wurde ein C-Modul namens MATHE7_?.C in die Bibliothek MATHE_?.LIB aufgenommen, in der die Funktion wie folgt implementiert ist:

```
int MathError(void)
```

In allen Fällen liefert diese Funktion einen Codewert zurück, der folgende Fehlermeldungen codiert:

- 1 unvollständige Modulo-Bildung; kein Fehler, Warnung
- 0 kein Fehler
- -1 Division durch 0
- -2 Überlauf
- -3 Unterlauf
- -4 Ln, Log oder Ld einer negativen Zahl
- -5 unerlaubte(s) Argument(e)
- -6 Winkel für Berechnungen zu groß
- -7 Tan / Cot nicht definiert!

Beachten Sie bitte, daß *MathError* nach dem Aufruf den Fehlercode löscht! Ferner ist zu berücksichtigen, daß jede anschließend aufgerufene Routine aus Mathe ebenfalls den Fehlercode überschreibt! Falls also ein eventueller Fehlercode ausgewertet werden soll, so hat dies unmittelbar nach der zu überprüfenden Routine zu erfolgen.

HINWEIS: Vergleichen Sie einmal die verschiedenen, analogen ASM-Dateien. So können Sie z.B. sehen, welche Erleichterungen die modernen Assembler heute bieten (*.MODEL, .CODE* und *.DATA* in MATHE?_?.ASM gegenüber MATHE?.ASM für Turbo Pascal, wo mit den SEGMENT-Anweisungen gearbeitet wird). Auch die (geringen) Unterschiede, die Sie berücksichtigen müssen, wenn Sie von Pascal- auf C-Konvention umschalten, lassen sich durch Vergleich der MATHE?_P.ASM-Dateien mit den MATHE?_C.ASM-Dateien demonstrieren.

F Tabelle der Assemblerbefehle

Auf den folgenden Seiten sind die Opcodes aller Befehle tabellarisch aufgeführt. Die Tabelle F.1 umfaßt hierbei bis auf folgende Ausnahmen die Ein-Byte-Befehle des Prozessors, das heißt, daß der Befehl durch genau ein Byte exakt definiert ist. Unberücksichtigt bleibt hierbei selbstverständlich, daß die Mehrzahl dieser Ein-Byte-Befehle weitere Bytes in Form von Argumenten nach sich ziehen!

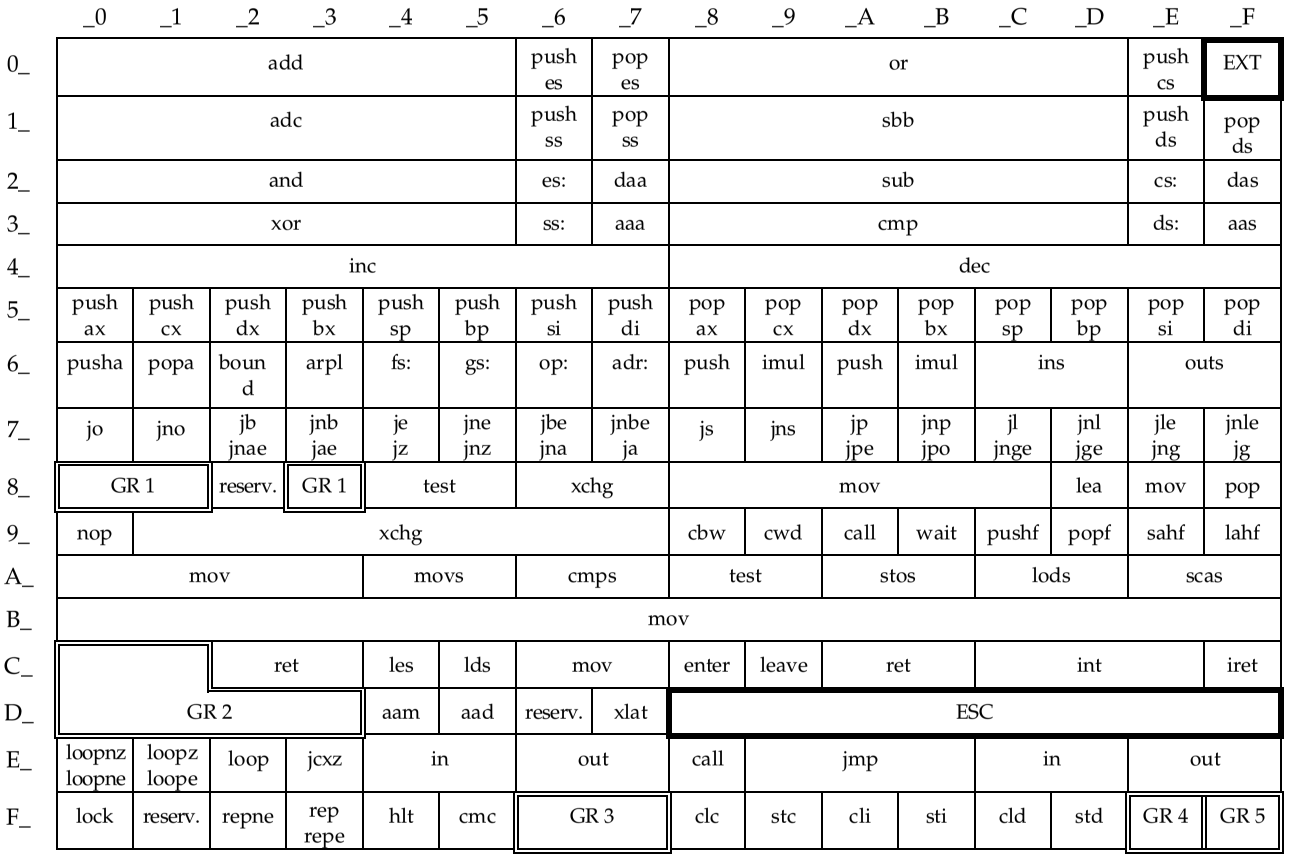
Die Ausnahmen sind der Opcode 0Fh, der die Tabelle der Zwei-Byte-Befehle einleitet, die ESC-Sequenzen, die die Coprozessorbefehle einleiten, sowie die mit GR1 bis GR5 bezeichneten Gruppen. Bei den ESC-Sequenzen sind die oberen Bits das Zeichen für den Prozessor, daß ein Coprozessorbefehl codiert ist. Die unteren Bits codieren dann zusammen mit dem folgenden, zweiten Byte des Opcodes den eigentlichen Coprozessorbefehl. Bei den Gruppenbefehlen wird im folgenden Adreßbyte (*mod-reg-r/m* – Byte 2) das *reg*-Feld zur Unterscheidung der Befehle herangezogen.

Tabelle F.2 enthält die jeweils zweiten Bytes des Opcodes für die Befehle, deren erstes Byte 0Fh ist. Auch hier sind Befehlsgruppen besonders gekennzeichnet. Deren Codierung ist identisch mit der der Gruppen 1 bis 5: Das folgende *mod-reg-r/m*-Byte (also insgesamt Byte 3!) spezifiziert mit seinem *reg*-Feld den Befehl.

Tabelle F.3 schlüsselt diese Gruppen auf. Dargestellt sind jeweils die Bitwerte für das *reg*-Feld für die acht Gruppen. Vorsicht! Die Decodierung des *mod-reg-r/m*-Bytes ist nicht trivial! Sie können in der Regel diesem Byte nicht auf Anhieb ansehen, welchen Befehl es codiert, da *mod* (Bits 7 und 6) sowie *r/m* (Bits 2 – 0) Angaben über die verwendeten Parameter enthalten und damit situationsabhängig unterschiedlich sind!

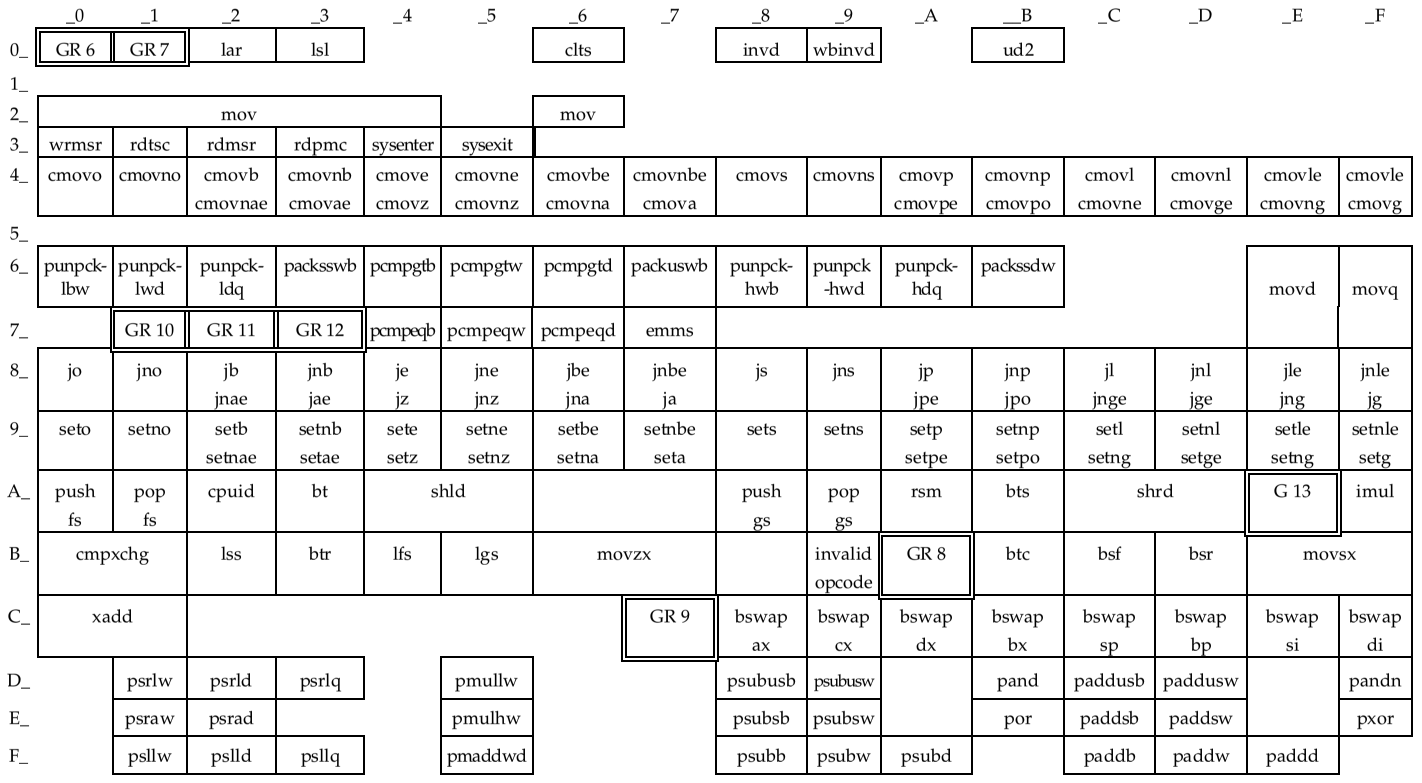
Tabelle F.4 schließlich listet die Coprozessorbefehle auf, wobei zu berücksichtigen ist, daß die Bits der vorangehenden ESC-Sequenz ebenfalls zur Codierung benutzt werden. Dementsprechend beginnen die Coprozessorbefehle mit den Bytes D8 bis DF, die in der ersten Spalte der Tabelle dargestellt sind. Die weiteren Spalten geben die Bitstellung des *spec*-Feldes im zweiten (*mod-spec-r/m*-) Byte des Opcodes an.

Da einige Befehle keine Operanden benötigen, kann bei diesen das *r/m*-Feld ebenfalls zur Befehlscodierung herangezogen werden. Bei diesen Befehlen wird das *mod*-Feld auf 11 gesetzt. Die sich daraus ergebenden Ausnahmen werden in den Tabellen F.5 bis F.9 dargestellt.



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	add						push es	pop es	or						push cs	EXT
1_	adc						push ss	pop ss	sbb						push ds	pop ds
2_	and						es:	daa	sub						cs:	das
3_	xor						ss:	aaa	cmp						ds:	aas
4_	inc								dec							
5_	push ax	push cx	push dx	push bx	push sp	push bp	push si	push di	pop ax	pop cx	pop dx	pop bx	pop sp	pop bp	pop si	pop di
6_	pusha	popa	boun d	arpl	fs:	gs:	op:	adr:	push	imul	push	imul	ins		outs	
7_	jo	jno	jb jnae	jnb jae	je jz	jne jnz	jbe jna	jnbe ja	js	jns	jp jpe	jnp jpo	jl jnge	jnl jge	jle jng	jnle jg
8_	GR 1		reserv.	GR 1	test		xchg		mov					lea	mov	pop
9_	nop	xchg							cbw	cwd	call	wait	pushf	popf	sahf	lahf
A_	mov				movs		cmps		test		stos		lods		scas	
B_	mov															
C_			ret		les	lds	mov		enter	leave	ret		int			iret
D_	GR 2				aam	aad	reserv.	xlat	ESC							
E_	loopnz loopne	loopz loope	loop	jcxz	in		out		call	jmp			in		out	
F_	lock	reserv.	repne	rep repe	hlt	cmc	GR 3		clc	stc	cli	sti	cld	std	GR 4	GR 5

Tabelle F.1: Ein-Byte-Opcodes. Die doppelt umrandeten Felder verweisen auf Gruppen von Befehlen, die anhand des folgenden *mod-reg-r/m*-Bytes unterschieden werden können (siehe Tabelle F.3). Die fett eingerahmten Befehle spezifizieren die Zwei-Byte-Opcodes mit führendem Byte 0Fh (siehe Tabelle F.2) sowie die Coprozessorbefehle (siehe Tabelle F.4).



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	__B	_C	_D	_E	_F
0_	GR 6	GR 7	lar	lsl			clts		invd	wbinvd		ud2				
1_																
2_	mov						mov									
3_	wrmsr	rdtsc	rdmsr	rdpmc	sysenter	sysexit										
4_	cmovo	cmovno	cmovb cmovnae	cmovnb cmovae	cmove cmovz	cmovne cmovnz	cmovbe cmovna	cmovnbe cmova	cmovs	cmovns	cmovp cmovpe	cmovnp cmovpo	cmovl cmovne	cmovnl cmovge	cmovle cmovng	cmovle cmovg
5_																
6_	punpck-lbw	punpck-lwd	punpck-ldq	packsswb	pcmpgtb	pcmpgtw	pcmpgtd	packuswb	punpck-hwb	punpck-hwd	punpck-hdq	packssdw			movd	movq
7_		GR 10	GR 11	GR 12	pcmpeqb	pcmpeqw	pcmpeqd	emms								
8_	jo	jno	jb jnae	jnb jae	je jz	jne jnz	jbe jna	jnbe ja	js	jns	jp jpe	jnp jpo	jl jnge	jnl jge	jle jng	jnle jg
9_	seto	setno	setb setnae	setnb setae	sete setz	setne setnz	setbe setna	setnbe seta	sets	setns	setp setpe	setnp setpo	setl setng	setnl setge	setle setng	setnle setg
A_	push fs	pop fs	cpuid	bt	shld				push gs	pop gs	rsm	bts	shrd		G 13	imul
B_	cmpxchg		lss	btr	lfs	lgs	movzx			invalid opcode	GR 8	btc	bsf	bsr	movsx	
C_	xadd							GR 9	bswap ax	bswap cx	bswap dx	bswap bx	bswap sp	bswap bp	bswap si	bswap di
D_		psrlw	psrld	psrlq		pmullw			psubusb	psubusw		pand	paddusb	paddusw		pandn
E_		psraw	psrad			pmulhw			psubsb	psubsw		por	paddsb	paddsw		pxor
F_		psllw	pslld	psllq		pmaddwd			psubb	psubw	psubd		paddb	paddw	paddd	

Tabelle F.2: Zwei-Byte-Opcodes. Das erste, führende Byte ist hierbei 0Fh (siehe Tabelle F.1). Die doppelt umrandeten Felder verweisen auf Gruppen von Befehlen, die anhand des folgenden *mod-reg-r/m*-Bytes unterschieden werden können (siehe Tabelle F.3).

	000	001	010	011	100	101	110	111
GR 1	add	or	adc	sbb	and	sub	xor	cmp
GR 2	rol	ror	rcl	rcr	shl	shr		sar
GR 3	test		not	neg	mul	imul	div	idiv
GR 4	inc	dec						
GR 5			call		jmp		push	
GR 6	sldt	str	lldt	ltr	verr	verw		
GR 7	sgdt	sidt	lgdt	lidt	smsw		lmsw	invdpg
GR 8					bt	bts	btr	btc
GR 9		cmpxchg8b						
GR 10			psrlw		psarw		psllw	
GR 11			psrld		psrad		pslld	
GR 12			psrlq				psllq	
GR 13	fxsave	fxrstor						

Tabelle F.3: Die Gruppen 1 bis 8. Die Interpretation des führenden Opcodes ist abhängig vom *reg*-Feld des folgenden *mod-reg-r/m*-Bytes. Die Bitstellungen dieses Feldes stehen im Tabellenkopf und spezifizieren den Befehl.

	000	001	010	011	100	101	110	111
D8	fadd	fmul	fcom	fcomp	fsub	fsubr	fdiv	fdivr
D9	fld		fst	fstp	fldenv	fldcw	fstenv	fstcw
DA	fiadd	fimul	ficom	ficomp	fisub	fisubr	fidiv	fidivr
DB	fild		fist	fistp		fld		fstp
DC	fadd	fmul	fcom	fcomp	fsub	fsubr	fdiv	fdivr
DD	fld		fst	fstp	frstor		fsave	fstsw
DE	fiadd	fimul	ficom	ficomp	fisub	fisubr	fidiv	fidivr
DF	fild		fist	fistp		fld		fstp

Tabelle F.4: Die Coprozessorbefehle. In der ersten Spalte wird jeweils das erste Byte des Zwei-Byte-Opcodes genannt. Hierbei entspricht z.B. das Byte D8 dem Befehl ESC – 0 (vgl. Tabelle F.1). Das zweite Byte wird über die Felder *mod* – *spec* – *r/m* codiert. Diese Tabelle zeigt Befehle, bei denen das *mod*-Feld *nicht* 11b ist. Die Spaltenüberschriften bezeichnen die Bitstellungen des *spec*-Feldes, das nicht dargestellte *r/m*-Feld codiert zusammen mit dem ebenfalls nicht dargestellten *mod*-Feld die Operanden. Die Befehlscodes in den jeweiligen Spezialfällen *mod* = 11b werden auf den nächsten Seiten dargestellt.

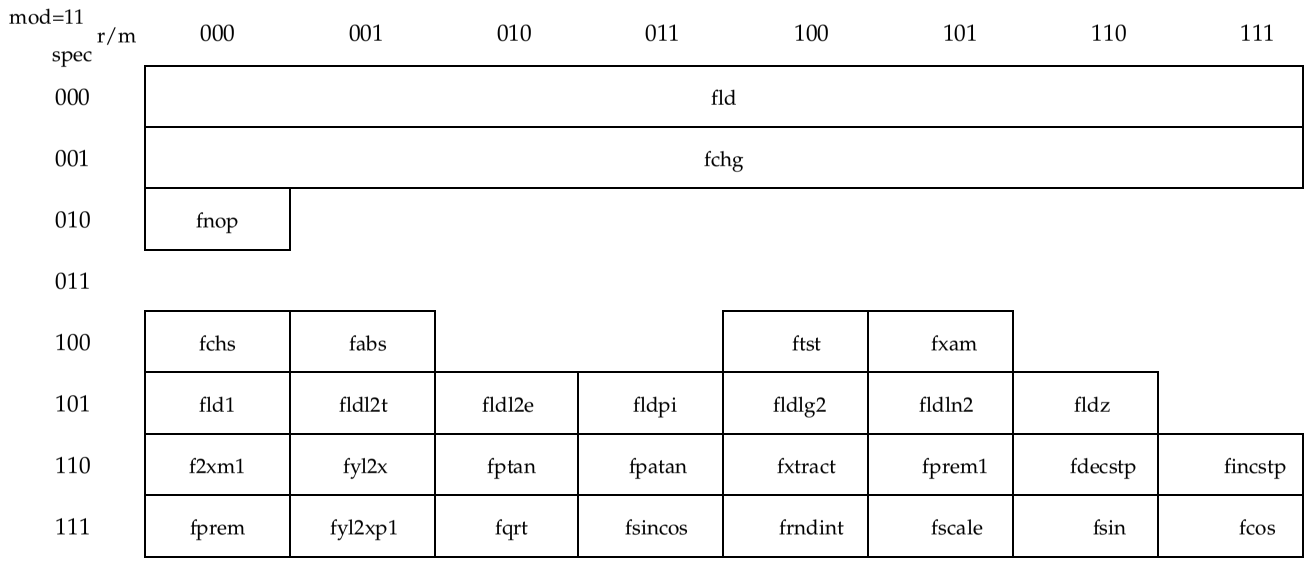


Tabelle F.5: ESC-1-(=D9)-Befehle, bei denen das *mod*-Feld des zweiten Bytes des Opcodes 11b ist. In der Tabelle sind das *spec*-Feld als Spalte und das *r/m*-Feld als Zeile dargestellt. Der Opcode zu den einzelnen Befehlen kann leicht aus den Daten der Tabelle berechnet werden. Beispiel: FLDLG2 steht unter *spec* = 101b und *r/m* = 100b. Somit ist das *mod-reg-r/m*-Feld des Befehls 11 – 101 – 100 oder, umgruppiert, 1110 1100b = ECh. Der gesamte Opcode für FLDLG2 ist dann (ESC-1 = D9!): D9 EC. Befehle, die die ganze Zeile belegen (z.B. FLD), codieren in ihrem *r/m*-Feld die Nummer des betroffenen Registers der FPU (mit drei Bits, da es nur acht Register gibt). Beispiel: bei FCHS ST(3) hätte das *r/m*-Feld den Wert 011b. Das gesamte *mod-spec-r/m*-Feld besitzt somit den Inhalt 11 – 001 – 011 oder CBh. Der Opcode lautet dann D9 CB.

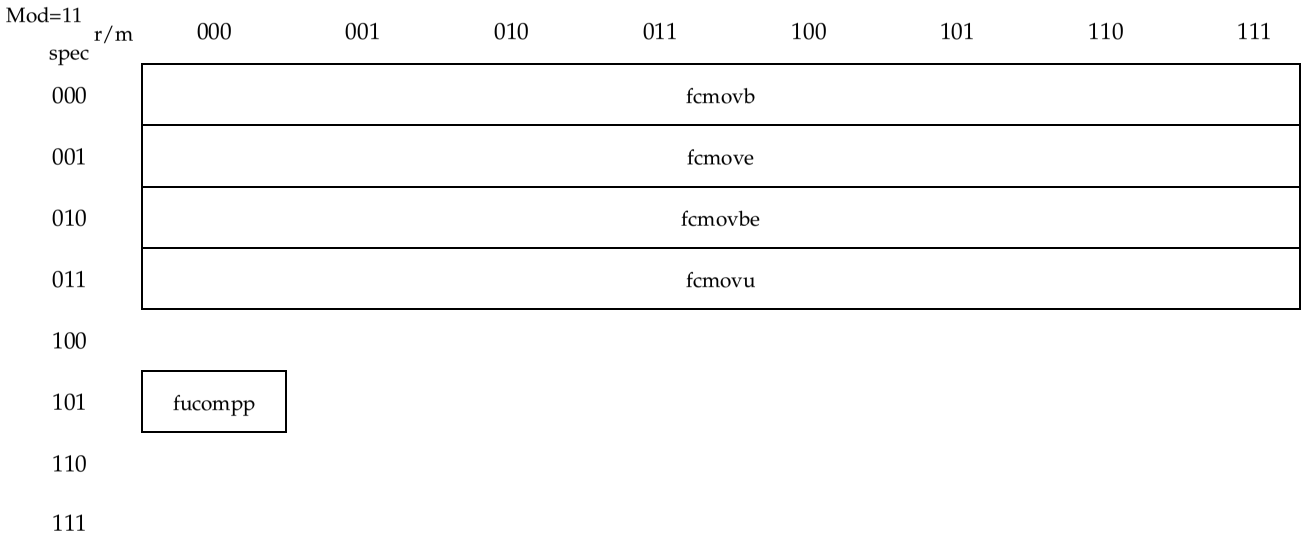


Tabelle F.6: ESC-2-(=DA)-Befehle, bei denen das *mod*-Feld des zweiten Bytes des Opcodes 11b ist. In der Tabelle ist das *spec*-Feld als Spalte und das *r/m*-Feld als Zeile dargestellt. Zu Details der Opcode-Berechnung siehe die Legende von Tabelle F.5.

Mod=11 r/m / spec	000	001	010	011	100	101	110	111
000	fcmovnb							
001	fcmovne							
010	fcmovnbe							
011	fcmovnu							
100	feni	fdisi	fclex	finit	fsetpm			
101	fucomi							
110	fcomi							
111								

Tabelle F.7: ESC-3-(=DB)-Befehle, bei denen das *mod*-Feld des zweiten Bytes des Opcodes 11b ist. In der Tabelle sind das *spec*-Feld als Spalten und das *r/m*-Feld als Zeile dargestellt. Zu Details der Opcode-Berechnung siehe die Legende von Tabelle F.5.

Mod=11 r/m / spec	000	001	010	011	100	101	110	111
000								
001								
010								
011	fcompp							
100								
101								
110								
111								

Tabelle F.8: ESC-6-(=DE)-Befehle, bei denen das *mod*-Feld des zweiten Bytes des Opcodes 11b ist. In der Tabelle sind das *spec*-Feld als Spalten und das *r/m*-Feld als Zeile dargestellt. Zu Details der Opcode-Berechnung siehe die Legende von Tabelle F.5.

mod=11 spec / r/m	000	001	010	011	100	101	110	111
000								
001								
010								
011								
100	fstsw ax							
101	fucomip							
110	fcomip							
111								

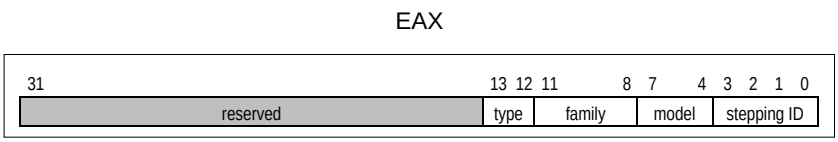
Tabelle F.9: ESC-7-(=DF)-Befehle, bei denen das *mod*-Feld des zweiten Bytes des Opcodes 11b ist. In der Tabelle sind das *spec*-Feld als Spalten und das *r/m*-Feld als Zeile dargestellt. Zu Details der Opcode-Berechnung siehe die Legende von Tabelle F.5.

G Die Register und Flags der Prozessoren

An dieser Stelle möchte ich noch einmal die wichtigsten Register moderner Intel-Prozessoren aufführen. Dargestellt werden die Register der P6-Familie, da sie als bislang neueste Prozessorgeneration zum einen alle Veränderungen seit dem 8086 berücksichtigen, zum anderen eben im Rahmen der »Abwärtskompatibilität« nichts unterschlagen, was etwa eine Vorgängergeneration an Funktionalität besessen hat. Die Auflistung stellt somit das Nonplusultra dar, das im Einzelfall so nicht realisiert sein kann. Für jedes Flag bzw. jedes Register ist verzeichnet, ab welcher Prozessorgeneration es implementiert wurde. Zu Einzelheiten vgl. die einzelnen Kapitel in Teil 1 des Buches.

G.1 CPUID

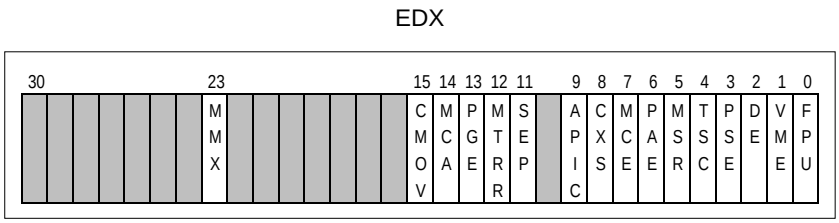
Obwohl kein Register, sei an dieser Stelle der CPUID-Befehl erwähnt, da er eine Reihe von Informationen über Features des verwendeten Prozessors liefert. Die meisten seiner Informationen bezieht er aus verschiedenen Registern, die im Anschluß dargestellt werden. Doch es gibt auch Features, die nur über den CPUID-Befehl festgestellt werden können (da Intel die korrespondierenden Flags in den Registern nicht dokumentiert hat): Zu Einzelheiten über die Anwendung des CPUID-Befehls siehe die Referenz in Teil 3. An dieser Stelle werden nur die Informationen dargestellt, die der CPUID-Befehl in den Registern EAX und EDX zurückgibt, wenn er mit dem Wert 1 in EAX aufgerufen wird. Die Inhalte der Register EBX und ECX sind in diesem Fall reserviert.



Angaben zur Prozessorarchitektur werden in EAX zurückgegeben. Im Feld *family* (Bits 11 bis 8) wird mit vier Bits die Prozessorfamilie angegeben. Für neuere Versionen des 80486, die den CPUID-Befehl unterstützen, wird hier der Wert »4« (= 0100) zurückgegeben, für die Pentium-Familie »5« (= 0101). Die Prozessoren der Familie P6 (Pentium Pro, Pentium II) tragen hier den Code »6« (= 0110) ein. Man braucht kein Prophet zu sein, um zu wissen, welcher Code von einer der nächsten Prozessorgenerationen an dieser Stelle verwendet wird. (Man kommt ins Grübeln: Hört Intel bei P15 auf? Oder wird dann CPUID aufgebohrt? Oder haben wir danach den Weltuntergang?)

Innerhalb der Familie gibt es noch Modelle. Das dazugehörige Model-Field gibt folgerichtig an, welches Modell der Familie vorliegt, also z.B. Pentium Pro oder Pentium II als Mitglieder der P6-Familie. »Updates« innerhalb eines Modells werden durch die Stepping-ID codiert.

In EDX werden die Features zurückgegeben, die der Prozessor unterstützt:



FPU Floating-Point-Unit on Chip; der »Coprozessor« ist auf dem Chip des Prozessors realisiert.

VME Virtual-8086-Mode Enhancements; der Prozessor unterstützt folgende Erweiterungen:

- VME das Flag VME in CR4 schaltet die Erweiterungen des Virtual-8086-Mode ein.
- PVI Protected-Mode-Virtual-Interrupts werden erlaubt.
- VIF das Virtual-Interrupt-Flag VIF in EFlags ist nutzbar.
- VIP das Virtual-Interrupt-Pending-Flag in EFlags ist nutzbar.

DE Debugging-Extensions; der Prozessor unterstützt I/O-Breakpoints und das DE-Flag in CR4.

PSE Page-Size-Extensions; der Prozessor unterstützt die Nutzung von 4-MByte-Pages. Das umfaßt die Unterstützung folgender Flags:

- PSE das Flag PSE in CR4 erlaubt die Nutzung von 4-MByte-Pages.
- PS das Page-Size-Flag in Page-Directory-Entries (PDEs) und Page-Table-Entries(PTEs) gibt die Seitengröße an (PS=0: 4-kByte-Pages, PS=1: 4-MByte-Pages).

TSC Time-Stamp-Counter; der Prozessor unterstützt Time-Stamps. Dies wird bewerkstelligt durch:

- RDTSC den Befehl RDTSC, mit dem das Time-Stamp-Counter-Register (ein MSR) ausgelesen werden kann.
- TSD Time-Stamp-Disable in CR4, das den Zugriff mittels RDTSC unterbindet.

MSR model-specific-registers; der Prozessor unterstützt die Befehle RDMSR und WRMSR zum Auslesen und Beschreiben von modellspezifischen Registern.

PAE Physical-Address-Extension; der Prozessor unterstützt das PAE-Flag in CR4 und somit Adressen mit mehr als 32 Bits und somit einen größeren Adreßraum als 4 GByte. Die Anzahl der Bits für die Adressen ist implementationsabhängig (beim Pentium Pro sind es 36 Bits). Gesteuert werden größere Adressen durch das:

MCE *Machine-Check*-Exception; der Prozessor unterstützt das MCE-Flag in CR4 und damit die Möglichkeit, *Machine-Check*-Exceptions auszulösen.

CX8 der Prozessor unterstützt den Befehl CMPXCHG8B.

APIC Advanced-Interrupt-Controller-on-Chip; der Prozessor besitzt auf seinem Chip einen Advanced-Interrupt-Controller, der auch aktiviert und einsatzbereit ist.

SEP Sysenter-Present; der Prozessor unterstützt den Fast-System-Call und damit die Befehle SYSENTER und SYSEXIT.

MTRR Memory-Type-Range-Registers; der Prozessor unterstützt bestimmte Model-Specific-Registers, die MTRRs.

PGE der Prozessor unterstützt das PGE-Flag in CR4 und somit die Möglichkeit, das Global-Bit in Page-Directory-Entries (PDEs) und Page-Table-Entries(PETs) zu verändern.

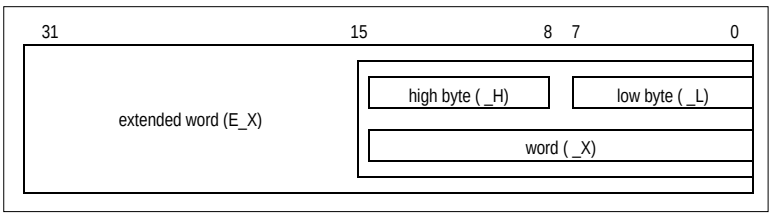
MCA Machine-Check-Architecture; der Prozessor unterstützt ein bestimmtes Model-Specific-Register (MSR), das sog. MCG_CAP.

CMOV der Prozessor unterstützt den CMOVcc-Befehl.

MMX der Prozessor unterstützt die MMX-Befehle.

G.2 Die Allzweckregister des Prozessors

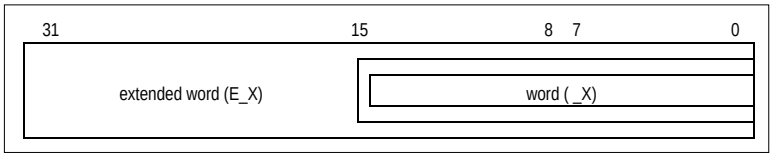
Es gibt vier Allzweckregister: (E)AX, (E)BX, (E)CX und (E)DX.



Einzelheiten zu diesen Registern finden Sie in Teil 1 des Buches.

G.3 Die Allzweckregister des Prozessors mit besonderen Funktionen

Es gibt vier Allzweckregister mit besonderen Funktionen: (E)BP, (E)SP, (E)SI und (E)DI



Einzelheiten zu diesen Registern finden Sie in Teil 1 des Buches.

G.4 Das Flagregister (E)Flags des Prozessors

Das Flagregister hält verschiedene StatusFlags, die Auskunft über die Ergebnisse von arithmetischen Instruktionen geben. Darüber hinaus finden sich im EFlags-Register auch verschiedene SystemFlags. SystemFlags sollten bis auf DF nicht von Anwendungen verändert werden, da sie vom Betriebssystem für dessen Funktion verwendet werden.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

StatusFlags:

CF *Carry*-Flag; das Flag signalisiert einen Überlauf bei Integer-Berechnungen mit vorzeichenlosen Integers.

PF Parity-Flag; das niederwertige Byte enthält bei gesetzten PF eine gerade Anzahl von gesetzten Bits.

AF Adjust-Flag; eine Integer-Berechnung liefert einen Überlauf an Bit 3; genutzt für BCD-Arithmetik.

ZF Zero-Flag; das Ergebnis einer Integer-Berechnung ist 0.

SF Sign-Flag; das Most-Significant-Bit (MSB; bit 7 bei Bytes, bit 15 bei Words, bit 31 bei Doublewords) wurde bei einer Integer-Berechnung gesetzt.

OF Overflow-Flag; signalisiert wie CF einen Überlauf, wobei jedoch das MSB nicht berücksichtigt wird; verwendet bei vorzeichenbehafteten Integers.

SystemFlags:

TF Trap-Flag; erlaubt, wenn gesetzt, die Ausführung von Programmen im Single-Step-Mode.

IF Interrupt-Flag; erlaubt, wenn gesetzt, die Antwort auf sog. maskierbare Interrupts.

DF Direction-Flag; steuert die Schreib-/Leserichtung bei Stringbefehlen (MOVS, STOS, LODS, SCAS, INS, OUTS). Bei gesetztem Bit wird die Adresse automatisch dekrementiert (Richtung: von hohen zu niedrigen Adressen), bei gelöschtem Bit inkrementiert (Richtung: von niedrigen zu hohen Adressen).

IOPL I/O-Privileg-Level; gibt die Privilegstufe für Input/output-Aktionen des aktuellen Tasks. Um auf den I/O-Adreßraum zurückgreifen zu können, muß das IOPL kleiner oder gleich dem CPL sein.

NT Nested-Task-Flag; gibt im gesetzten Zustand an, daß der aktuelle Task mit dem vorangehenden verbunden ist; dessen Adresse steht im Nested-Task-Link-Field des TSS des aktuellen Tasks. NT wird im Kontext mit Interrupts verwendet.

RF Resume-Flag; steuert die Reaktion des Prozessors auf Debug-Exceptions.

VM Virtual-8086-Mode; schaltet, wenn gesetzt, in den Virtual-8086-Mode um. Wird das Flag gelöscht, wird in den Protected-Mode zurückgeschaltet.

AC Alignment-Check; erlaubt, wenn gesetzt, in Verbindung mit dem AM Flag in CR0 bei CPL = 3 die Prüfung auf korrekt ausgerichtete Daten.

VIF Virtual-Interrupt-Flag; dies ist das virtuelle Abbild des IF-Flags. Wird in Verbindung mit dem VIP-Flag bei der Interruptbehandlung im Virtual-8086-Mode benutzt, wenn die Virtual-8086-Mode-Extensions durch Setzen des VME-Flags in CR4 verfügbar gemacht wurden.

VIP Virtual-Interrupt-Pending; wird von der Software gesetzt, wenn virtuelle Interrupts anhängig sind. Der Prozessor liest dieses Flag nur, verändert es aber selbst nicht. VIP wird in Verbindung mit VIF verwendet.

ID Identification-Flag; signalisiert, daß der Befehl CPUID unterstützt wird, wenn es gesetzt und gelöscht werden kann.

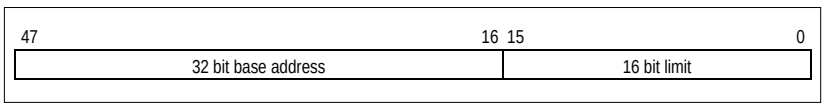
G.5 Die Segmentregister des Prozessors

Es gibt acht Segmentregister: CS, DS, ES, FS, GS, SS, TR und LDTR

Auf diese Register kann über die Befehle LDS, LES, LFS, LGS, LSS, LTR/STR und LLDT/SLDT nur auf die Bits 64 bis 79 zugegriffen werden, um die Selektoren einzutragen/auszulesen. Die als nicht sichtbar bezeichneten Felder stellen einen Cache dar, der vom Prozessor nach dem Laden des Selektors aus dem korrespondierenden Deskriptor gefüllt wird.

G.6 Die Adreßregister des Prozessors

Es gibt zwei Adreßregister: GDTR und IDTR.



Diese beiden Register können (und müssen) vollständig mittels der Befehle LGDT/SGDT und LIDT/SIDT geladen und ausgelesen werden. In ihnen sind die virtuelle Adresse und die Größe der korrespondierenden Deskriptortabelle (GDT bzw. IDT) verzeichnet.

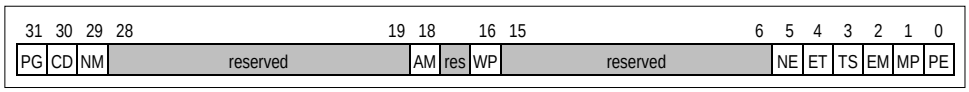
G.7 Das Machine-Status-Word (MSW)

Das *Machine-Status-Word* ist mit den Bits 15 bis 0 von Kontrollregister CR0 identisch. Es wurde mit dem 80286 als eigenständiges Register eingeführt und ist ab dem 80386 in den neu eingeführten Kontrollregistern aufgegangen. Die Befehle, mit denen auf das *Machine-Status-Word* zugegriffen werden kann, LMSW und SMSW, sind nur noch aus Gründen der Abwärtskompatibilität implementiert und durch die Erweiterung des MOV-Befehls obsolet worden.

G.8 Die Kontrollregister des Prozessors

Es gibt fünf Kontrollregister: CR0 bis CR4.

G.8.1 CR0



CR0 ist eines von bislang zwei Registern moderner Prozessoren, in denen System-Flags beheimatet sind, die die Funktion des Prozessors in bestimmten Situationen steuern. Eingeführt mit dem 80386 ist das CR0, das »älteste« Kontrollregister für SystemFlags. Es beinhaltet daher alle Flags, die von allen Prozessoren ab dem 80386 unterstützt werden:

PE Protection-Enable (80386); wenn das Flag gesetzt ist, wird der Protected-Mode eingestellt. In diesem Fall greifen zwar die Schutzkonzepte auf Segmentebene, nicht aber notwendigerweise auf Page-Ebene. Dies muß durch das Setzen des PG-Flags explizit eingeschaltet werden. Wird das PE-Flag gelöscht, wird in den Real-Mode umgeschaltet.

MP Monitor-Coprocessor (80386); siehe TS-Flag.

EM Emulation (80386); im gesetzten Zustand signalisiert dieses Flag, daß weder eine interne (80486 DX, Pentium, P6) noch externe (80386, 80486 SX) FPU zur Verfügung steht und FPU-Befehle emuliert werden müssen. Im gelöschten Zustand ist eine FPU vorhanden, und die FPU-Befehle können dann ausgeführt werden. In diesem Fall kann durch Setzen des EM-Flags erreicht werden, daß trotz Vorhandenseins einer FPU die FPU-Emulation erfolgt. Ist EM gesetzt, so wird bei jedem FPU-Befehl eine #NM-Exception ausgelöst. Es liegt dann in der Verantwortung des Handlers dieser Exception, FPU-Befehle zu emulieren.

Achtung: EM hat auch auf die MMX-Befehle Einfluß. Ist EM gesetzt, so führt jede Ausführung eines MMX-Befehls zu einer #DU-Exception.

TS Task-Switched (80386); der Prozessor hat einen Task-Switch durchgeführt. Das Flag dient dazu, die FPU-Umgebung nach einem Task-Switch erst dann zu sichern, wenn ein FPU-Befehl im neuen Task ausgeführt werden soll. Dies dient dazu, unnötige, zeitaufwendige Sicherungen der FPU-Umgebung zu verhindern, falls im neuen Task nicht auf die FPU zurückgegriffen wird. TS wird vom Prozessor bei jedem Task-Switch gesetzt und vor jeder FPU-Instruktion geprüft. Im Falle eines gesetzten TS wird vor jedem FPU-Befehl eine #NM-Exception ausgelöst, falls zusätzlich MP gesetzt ist, auch im Falle von WAITs/FWAITs. Der Exception-Handler sollte dann mit dem Befehl CLTS (Clear-Task-Switch) dieses Flag wieder zurücksetzen, da es von niemand anderem verändert wird.

ET Extension-Type (Pentium); zeigt im gesetzten Zustand an, daß die 80387-FPU-Instruktionen auf 80386- und 80486-Systemen verwendet werden können. Bei Pentium- und P6-Prozessoren ist dieses Bit reserviert und auf »1« gesetzt (»FPU-Instruktionen werden unterstützt«).

NE Numeric-Error (80486); steuert die Art des Reports von Exceptions der FPU.

WP Write-Protect (80486); ist dieses Flag gesetzt, so verhindert es einen schreibenden Zugriff von Prozeduren im Supervisory-Level auf Read-only-Pages mit User-Privilegien. Dadurch werden bestimmte Funktionen einiger Betriebssysteme unterstützt (Forking bei Unix).

AM Alignment-Mask (80486); schaltet im gesetzten Zustand und bei gesetztem AC-Flag in EFlags die Ausrichtungsprüfung ein, die bei einem CPL = 3 Daten auf ihre korrekte Ausrichtung im Speicher hin überprüft.

NM Not-Write-Through (80486); schaltet im gelöschten Zustand den Write-Back- (Pentium und P6) bzw. den Write-Through-Mechanismus (80486) ein, mit dem bei Schreibzugriffen auf den Cache dieser geleert und invalidiert wird. NW arbeitet mit CD zusammen.

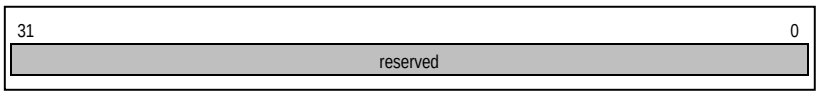
CD Cache Disable (80486); schaltet den Caching-Mechanismus teilweise ab. Im gelöschten Zustand kann der gesamte physikalische Speicher in den prozessorinternen und -externen Cache-Speichern gepuffert werden. CD arbeitet mit NW zusammen.

PG *Paging* (80386); schaltet den Paging-Mechanismus frei, mit dem eine dynamische Speicherverwaltung realisiert werden kann. Dieses Flag hat nur eine Bedeutung und kann (ohne Auslösen einer #GP-Exception) nur verwendet werden, wenn das PE-Flag in CR0 gesetzt ist.

Die folgende Tabelle gibt an, welche Aktion der Prozessor in Abhängigkeit vom Status der Flags EM, MP und TS ausführt:

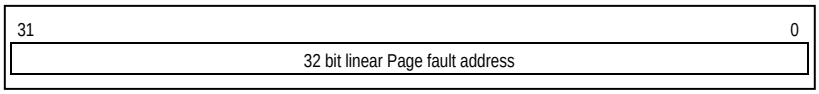
Flag			Instruktion		
EM	MP	TS	WAIT/FWAIT	FPU-Befehl	MMX-Befehl
0	0	0	WAIT/FWAIT	FPU-Befehl	MMX-Befehl
0	0	1	WAIT/FWAIT	#NM-Exception	#NM-Exception
0	1	0	WAIT/FWAIT	FPU-Befehl	MMX-Befehl
0	1	1	#NM-Exception	#NM-Exception	#NM-Exception
1	0	0	WAIT/FWAIT	#NM-Exception	#UD-Exception
1	0	1	WAIT/FWAIT	#NM-Exception	#UD-Exception
1	1	0	WAIT/FWAIT	#NM-Exception	#UD-Exception
1	1	1	#NM-Exception	#NM-Exception	#UD-Exception

G.8.2 CR1



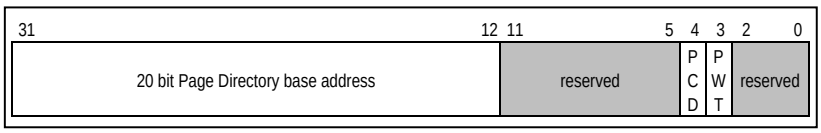
No comment!

G.8.3 CR2



Das Kontrollregister CR2 nimmt im Falle einer #PF-Exception die lineare 32-Bit-Adresse der Page auf, auf die zugegriffen werden sollte, die jedoch im Page-Directory-Entry oder Page-Table-Entry als not present markiert ist. Der Exception-Handler kann auf diese Weise dafür sorgen, daß die entsprechende Page nachgeladen wird. (Die übergebene Adresse ist ja eine Virtuelle Adresse, die in den Bits 31 bis 22 den Selektor für das Page-Table-Directory und in den Bits 21 bis 12 den Selektor für die Page-Table beinhaltet. Die benötigte Page ist damit feststellbar.)

G.8.4 Das Page-Directory-Base-Register (PDBR; CR3)

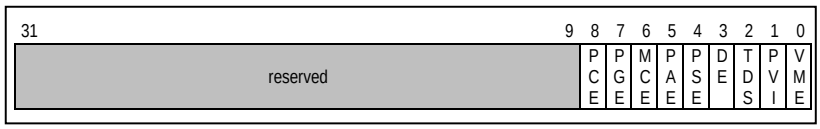


Das Register hält die Adresse des Page-Table-Directory. Aus den Bits 12 bis 31 dieses Registers wird die virtuelle Adresse dieser Schlüsseltabelle für den Paging-Mechanismus durch Skalieren mit 2^{12} gebildet. Ferner beinhaltet dieses Register noch einige VerwaltungsFlags, die beim Umgang mit der durch die Adresse referenzierten Tabelle eine Rolle spielen:

PWT Page-Level-Writes-Transparent (80486); legt die Strategie für das Cachen des aktuellen Page-Directory fest: bei gesetztem Flag erfolgt Write-through-Caching, beim gelöschtem Flag Write-back-Caching. Dies betrifft nur die internen Caches (sowohl L1 wie auch L2, wenn verfügbar), externe Caches sind nicht betroffen. Falls Paging abgeschaltet wurde (PG in CR0 ist gelöscht) oder Caching inaktiviert wurde (CD in CR0 ist gesetzt), hat das Flag keine Bedeutung.

PCD Page-Level-Cache-Disable (80486); ist dieses Flag gesetzt, wird das Cachen des aktuellen Page-Directory unterbunden. Andernfalls erfolgt ein Cachen, so das Paging überhaupt erfolgt (PG in CR0 ist gesetzt) und das Cachen grundsätzlich erlaubt wurde (CD in CR0 ist gelöscht). Welche Strategie für das Cachen benutzt wird, entscheidet das PCD-Flag.

G.8.5 CR4



Das Kontrollregister CR4 ist das zweite und neueste Register für Systemflags. Es beinhaltet die Flags, die spezifische Funktionen der Prozessoren ab dem Pentium steuern:

VME Virtual-8086-Mode-Extension (Pentium); ein gesetztes VME-Flag schaltet die Exception und Interrupt-Extensions im Virtual-8086-Mode ein.

PVI Protected-Mode-Virtual-Interrupts (Pentium); ein gesetztes PVI-Flag schaltet die Hardwareunterstützung für virtuelle Interrupts im Protected-Mode ein: Wenn das PVI-Flag gesetzt ist, kann das VIF-Flag in EFlags geändert werden. Ist das PVI-Flag dagegen gelöscht, so wird auch das VIF-Flag gelöscht.

TDS Time-Stamp-Disable (Pentium); wenn dieses Flag gesetzt wird, kann auf den Time-Stamp-Counter über den Befehl RDTSC nur noch bei CPL = 0 zugegriffen werden. Bei gelöschten Flag ist dies in allen Privilegstufen möglich.

DE Debugging-Extension (Pentium); schaltet im gesetzten Zustand die Debug-Extensions frei und somit die Debugregister DR4 und DR5. Ist es gelöscht, referenzieren die Debugregister DR4 und DR5 aus Kompatibilitätsgründen mit anderen Prozessorfamilien die Debugregister DR6 und DR7.

PSE Page-Size-Extension (Pentium); erlaubt die Nutzung von 4-MByte-Pages, falls es gesetzt ist. Im gelöschten Zustand haben Pages eine Größe von 4 kByte.

PAE Physical-Address-Extension (Pentium); im gesetzten Zustand erlaubt dieses Flag die Nutzung von mehr als 32 Bit zur Adressierung. Voraussetzung ist allerdings, daß die Hardware dies durch Bereitstellung von mehr als 32 Adreßleitungen auch unterstützt. Wie viele Adreßleitungen verfügbar sind, ist von der Prozessorfamilie abhängig.

MCE Machine-Check-Enable (Pentium); dieses Flag ermöglicht im gesetzten Zustand eine Prüfung der Hardware, die prozessorspezifisch realisiert und damit nicht auf- und abwärtskompatibel ist. Wird ein Fehler in der Hardware (Prozessor!) vorgefunden, so wird die #MC-Exception ausgelöst.

PGE Page-Global-Enable (Pentium Pro); erlaubt die Nutzung des Global-Flags G in Page-Directory-Entries (PDEs) und Page-Table-Entries (PTEs). Wird nach Setzen des PGE-Flags in einem PDE oder PTE das G-Flag gesetzt, wird die

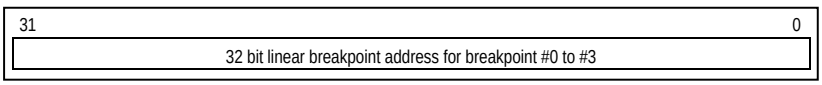
Page als global verfügbar betrachtet und daher nicht in den Paging-Mechanismus einbezogen: sie bleiben bei Task-Switches im Translation-Lookaside-Buffer (TLB) erhalten. Sinn macht dies bei häufig benutzten Pages oder Pages, die von vielen Tasks geteilt werden und somit potentiell immer wieder nachgeladen werden müßten.

PCE Performance-Monitoring-Counter-Enable (Pentium); wenn dieses Flag gesetzt wird, ist unter allen Privilegstufen ein Zugriff auf den Performance-Monitoring-Counter mittels des Befehls RDPMC möglich. Andernfalls ist dies nur bei CPL = 0 möglich.

G.9 Die Debugregister des Prozessors

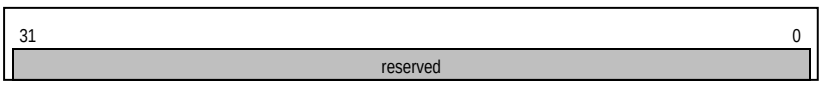
Es gibt acht Debugregister, die eine hardwareseitige Unterstützung für das Debuggen ermöglichen: DR0 bis DR7. Die ersten vier davon, DR0 bis DR3, sowie die Register DR6 und DR7 wurden mit dem 80386 eingeführt.

G.9.1 Breakpoint-Address-Register DR0 bis DR3



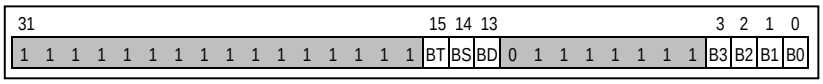
Die Debugregister DR0 bis DR3 nehmen die virtuellen Adressen von bis zu vier Breakpoints auf. Die Breakpoint-Adressen werden vor einer Umrechnung auf physikalische Adressen mit dem Program-Counter verglichen. Neben der Angabe seiner Adresse in einem der Debugregister wird ein Breakpoint auch durch Angaben im *Debug-Control-Register* DR7 definiert.

G.9.2 DR4 und DR5



DR4 und DR5 haben nur dann eine Funktion, wenn durch das Setzen des DE-Flags in Kontrollregister CR4 die Debugging-Extensions eingeschaltet wurden. Ein Zugriff auf diese reservierten Register führt dann zu einer #UD-Exception. Ist das DE-Flag dagegen gelöscht, verweisen die Registerbezeichnungen DR4 und DR5 auf die Debugregister DR6 bzw. DR7. Ein Zugriff auf DR4 bzw. DR5 hat somit einen Zugriff auf DR6 bzw. DR7 zur Folge, ohne eine Exception auszulösen.

G.9.3 Debug-Statusregister DR6



Das *Debug-Statusregister* enthält die Informationen über Debug-Conditions seit dem letzten Aufruf des Debug-Exception-Handlers. Das Register wird nur dann upgedated, wenn eine neue Exception ausgelöst wird. Solange bleibt der bisherige Zustand erhalten. Die Bits in DR6 zeigen den Grund für die Auslösung der Exception an:

B0 – B3 Breakpoint-Condition-Detected; für jeden der vier möglichen Breakpoints existiert ein Flag, das im gesetzten Zustand anzeigt, daß die Breakpoint-Condition, die im *Debug-Kontrollregister* DR7 in den Feldern LENx und R/Wx für jeden Breakpoint definiert wird, eingetreten ist, als die #BP-Exception ausgelöst wurde. Der Exception-Handler ist auf diese Weise in der Lage festzustellen, welcher Breakpoint zu der Exception geführt hat. Achtung: Die Flags B0 bis B3 werden auch gesetzt, wenn die Breakpoints über die Flags Ln und Gn inaktiviert wurden! Ein Handler muß daher unbedingt auch den Zustand dieser Flags berücksichtigen.

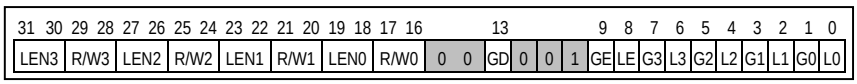
BD Debug-Register-Access-Detected; dieses Flag zeigt an, daß die nächste auszuführende Instruktion ein Befehl ist, der auf ein Debugregister zugreift. Dieses Flag wird jedoch nur gesetzt, wenn das Flag GD in DR7 gesetzt ist.

BS Single-Step; das gesetzte BS-Flag zeigt an, daß die Debug-Exception im Single-Step-Mode erfolgte (das Flag TF in EFlags ist dafür gesetzt!). Single-Step-Exceptions sind Exceptions der höchsten Priorität – daher können auch andere Debugbedingungen zutreffen (beim Trappen eines Breakpoints im Single-Step-Mode wird die Exception als Single-Step-Exception gewertet, auch wenn Breakpoint-Conditions zutreffen und daher entsprechende Flags gesetzt sind).

BT Task-Switch; ein gesetztes BT-Flag zeigt an, daß die Exception durch einen Task-Switch erfolgte. Dieser Zustand kann nur eintreten, wenn das T-Flag im TSS des neuen Tasks gesetzt ist – es gibt kein korrespondierendes Flag in einem der Debugregister.

Hinweis: Die Flags in DR6 sind sticky. Das bedeutet, daß sie nicht automatisch zurückgesetzt werden. Es liegt also in der Verantwortung des Exception-Handlers für Debug-Exceptions, die Condition-Flags zurückzusetzen, wenn der Grund für die Exception festgestellt worden ist. Andernfalls kann es zu falschen Reaktionen kommen.

G.9.4 Debug-Kontrollregister DR7



Über das *Debug-Kontrollregister* können die Bedingungen festgelegt werden, unter denen ein Breakpoint, dessen Adresse in eines der Breakpoint-Address-Registers DR0 bis DR3 eingetragen wurde, eine Exception auslöst. Diese Bedingungen können sein:

L0 – 3 Local-Breakpoint-Enable; schaltet die Breakpoint-Condition für den korrespondierenden Breakpoint im aktuellen Task frei. Wird eine Breakpoint-Condition festgestellt und ist das korrespondierende L-Flag gesetzt, so wird eine #BP-Exception ausgelöst. Dieses Flag regelt die lokale Freischaltung von Breakpoint-Conditions: bei jedem Task-Switch werden diese Flags vom Prozessor gelöscht, um nicht zu unerwünschten Breakpoint-Exceptions im neuen Task zu führen.

G0 – 3 Global-Breakpoint-Enable; schaltet die Breakpoint-Conditions für den korrespondierenden Breakpoint für alle Tasks frei. Das bedeutet, daß im Unterschied zu Lx bei Gx unabhängig vom Task immer eine #BP-Exception ausgelöst wird, wenn die Bedingungen erfüllt sind. Der Prozessor löscht diese Flags bei einem Task-Switch nicht.

LE Local-Exact-Breakpoint-Enable und

GE Global-Exact-Breakpoint-Enable stellen die exakte Instruktion fest, die zu der Exception geführt hat. Da nicht alle Prozessoren diese Flags gleichartig unterstützen (Prozessoren der P6-Familie unterstützen diese Flags beispielsweise gar nicht!), sollten diese Flags grundsätzlich auf »1« gesetzt sein.

GD Global-Detect-Enable; mit diesem Flag ist es möglich, die Debugregister zu schützen: ist es gesetzt, so wird *vor* jeder Ausführung eines MOV-Befehls mit einem Debugregister als Operanden das BD Flag in DR6 gesetzt und eine #BP-Exception ausgelöst. Der Prozessor löscht dann vor dem Eintritt in den Exception-Handler das Flag GD, um dem Handler die Möglichkeit zu geben, auf die Debugregister zuzugreifen.

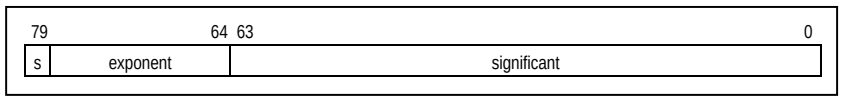
R/W0–3 Read/write-Fields; diese Felder spezifizieren die Breakpoint-Condition, die zur Auslösung einer Debug-Exception #BP führen. Das DE-Flag in DR6 regelt die Bedingungen näher (ein gelöschtes DE-Flag stellt die auf 80386- oder 80486-Prozessoren möglichen Bedingungen ein):

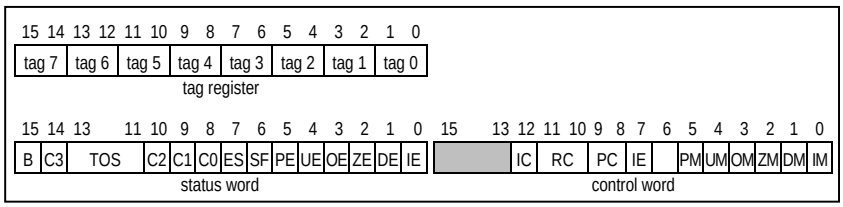
	Read/write Field			
DE	00	01	10	11
0	on instruction only	on data writes only	undefined	on data reads/writes
1	on instruction only	on data writes only	on I/O reads/writes	on data reads/writes

LEN0–3 gibt die Länge der Speicherstelle an, unter der die Adresse aufgefunden werden kann, die im korrespondierenden Breakpoint-Address-Register verzeichnet ist. Es gibt vier mögliche Werte: 00 = 1-Byte-Daten, 01 = 2-Byte-Daten, 10 = undefiniert und 11 = 4-Byte-Daten. Die Angabe der Länge des Datums macht nur bei Breakpoints Sinn, die nicht Instruktionen und deren Ausführung betreffen (R/Wx = 00). Hier sollte LENx = 00 sein (andernfalls ist das Verhalten der Exception-Auslösung nicht vorhersehbar, weil undefiniert!). LENx dient dazu, bei Read/write-Exceptions mit Daten die Größe der Daten zu definieren. Achtung: Wenn LENx nicht 00 ist, so müssen die Daten entsprechend ihrer Größe im Speicher ausgerichtet sein: bei LENx = 01 an Wortgrenzen, bei LENx = 11 an Doppelwortgrenzen.

G.10 Die FPU-Register des Prozessors

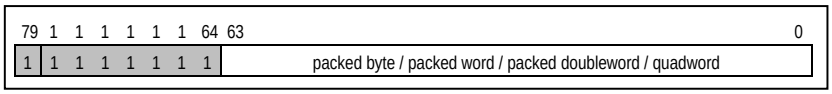
Es gibt acht Rechenregister: R0 bis R7. Sie sind in Form eines FPU-Stacks angeordnet. Der Zugriff auf die FPU-Register erfolgt nicht durch Angabe der Nummer des physikalischen Registers (R0 bis R7), sondern des Stapelregisters (ST(0) bis ST(7)). Welches der physikalischen Register die Spitze des Stapels darstellt (TOS), ist im Feld TOS des Statusworts angegeben. Steht an dieser Stelle der Wert 010 (= »2«), so ist R2 = TOS = ST(0). In diesem Fall ist R3 = ST(1), R4 = ST(2), R5 = ST(3), R6 = ST(4), R7 = ST(5), R0 = ST(6) und R1 = ST(7).





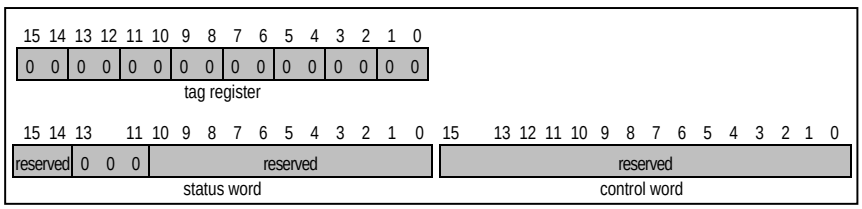
G.11 Die MMX-Register des Prozessors

Es gibt acht MMX-Register, die physikalisch mit den 8 FPU-Registern identisch sind: MM0 bis MM7. Der Zugriff auf ein MMX-Register erfolgt im Unterschied zum Zugriff auf FPU-Register immer durch Angabe der physikalischen Registernummer (R0 bis R7; die MMX-Register definieren keinen Stack!). So ist *immer* MM0 = R0, MM1 = R1 etc. Das TOS-Feld im Statuswort hat damit bei MMX-Registern keine Bedeutung und enthält daher immer den Wert 000.



Ferner werden von den FPU-Registern lediglich die Bits 0 bis 63 als MMX-Register verwendet. Die Bits 79 bis 64 haben keine Bedeutung und sind somit alle auf »1« gesetzt. Unter MMX sind alle Register mit gültigen Einträgen versehen, weshalb das Tag-Register vollständig den Wert »0« hat (außer nach Ausführung des Befehls EMMS: dann sind alle Register als empty markiert und das Tag-Register hat den Wert $FFFF).

Da die FPU-Register unter MMX nicht zur Bearbeitung von Fließkommazahlen mit den FPU-Befehlen dienen, machen unter MMX auch das *Kontroll-* und das Statuswort keinen Sinn. Deren Inhalte haben somit als reserviert zu gelten, ein Zugriff auf sie, der nur mit FPU-Befehlen möglich ist, führt zur Auslösung einer #UD-Exception.



H ASCII- und ANSI-Tabelle

Die Zeichen 0 bis 31 des ASCII- und ANSI-Zeichensatzes sind identisch und als Steuerzeichen zu werten. Neben ihrem dezimalen (dez) und hexadezimalen (hex) Wert sind auch ihre Darstellung (💻), Bezeichnung (Code) und Funktion aufgeführt.

dez	hex	💻	Code	Funktion
0	00		NUL	*null character*
1	01	☺	SOH	*start of header*; Kopfbeginn
2	02	☻	STX	*start of text*; Textbeginn
3	03	♥	ETX	*end of text*; Textende
4	04	♦	EOT	*end of transmission*; Übertragungsende
5	05	♣	ENQ	*enquiry*; Anfrage
6	06	♠	ACK	*acknowledge*; Bestätigung
7	07	•	BEL	*bell*; Tonsignal
8	08	◘	BS	*backspace*; Rückschritt
9	09	○	HT	*horizontal tabulation*; Horizontaltabulator
10	0A	◙	LF	*line feed*; Zeilenvorschub
11	0B	♂	VT	*vertical tabulation*; Vertikaltabulator
12	0C	♀	FF	*form feed*; Seitenvorschub
13	0D	♪	CR	*carriage return*; Wagenrücklauf
14	0E	♫	SO	*shift out*; Breitschrift
15	0F	☼	SI	*shift in*; Engschrift
16	10	►	DLE	*data link escape*; Steuerung der Verbindung
17	11	◄	DC1	*device control #1*; Einheitensteuerung 1
18	12	↕	DC2	*device control #2*; Einheitensteuerung 2
19	13	‼	DC3	*device control #3*; Einheitensteuerung 3
20	14	¶	DC4	*device control #4*; Einheitensteuerung 4
21	15	§	NAK	*negative acknowledgement*; negative Bestätigung
22	16	▬	SYN	*synchronous idle*; synchronisierter Leerlauf
23	17	↨	ETB	*end of transmission block*; Ende des Übertragungsblockes
24	18	↑	CAN	*cancel*; Abbruch
25	19	↓	EM	*end of medium*; Ende des Mediums
26	1A	→	SUB	*substitute*; Ersetzen
27	1B	←	ESC	*escape*; Steuerung
28	1C	∟	FS	*file separator*; Datei-Trenner
29	1D	↔	GS	*group separator*; Gruppen-Trenner
30	1E	▲	RS	*record separator*; Datensatz-Trenner
31	1F	▼	US	*unit separator*; Einheiten-Trenner

In der folgenden Tabelle sind die Zeichen mit den Nummern 32 bis 127 verzeichnet, die im ASCII- und ANSI-Zeichensatz identisch sind.

dez	hex	💻	dez	hex	💻	dez	hex	💻
32	20	[Leertaste]	64	40	@	96	60	´
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	

I Neue MASM- und TASM-Versionen

Gleich zu Beginn: Umwerfendes hat sich in der Version 4.0 des Turbo Assemblers nicht geändert! Auch in dieser Version muß der Anwender seinen eigenen Editor mitbringen. Weiterhin fehlt eine integrierte Entwicklungsumgebung wie bei den anderen Borland-Produkten oder MASM 6.0, so daß tpASM auch nach dem Erscheinen der neuen Version des Borland-Assemblers seine Existenzberechtigung behält. Und um auch dies gleich abzuhandeln: die Optionen, mit denen der Turbo Assembler aus der Kommandozeile gestartet wird, sind gleich geblieben. Lediglich einige ergänzende Angaben bei einigen Schaltern wurden hinzugefügt, so z.B. die »neuen« Warnklassen INT und MCP beim Schalter */w*. Allerdings werden Sie, vor allem als Nicht-*Power-User*, diese Unterschiede kaum nutzen. Dies bedeutet, daß Sie weiterhin TASM320.EXE als tpCONFIG.EXE verwenden können.

Worin also liegen die Unterschiede zwischen beiden Versionen? Hauptsächlich im »Feintuning«! TASM konnte auch schon in der Version 3.2 32-Bit-Code generieren, wie ihn z.B. WINDOWS NT oder OS/2 benutzen. Hierzu gab und gibt es zwei Versionen: TASM, das »nur« den konventionellen Speicher von 640 kByte benutzt, und TASMX, das DPMI-Dienste nutzt und somit größere Programme erstellen kann und konnte. Auch heute stellen, wie bisher, diese beiden Versionen »nur« 16-Bit-Debug-Informationen zur Verfügung. Sollen aber 32-Bit-Debug-Informationen genutzt werden, muß die nun neue Version TASM32 eingesetzt werden. TASM32 nutzt ebenfalls DPMI-Dienste, ist also die »32-Bit-Version« von TASMX.

Was heißt das im Klartext? Es kommt ganz darauf an, welche Version des Debuggers Sie nutzen wollen, um Ihre Assemblerteile zu untersuchen. Wollen (oder müssen) Sie TD32, den 32-Bit-Debugger benutzen, der nur in 32-Bit-Umgebungen wie WINDOWS NT oder mit WIN32s abläuft, so müssen die Assemblerquelltexte mit TASM32 assembliert worden sein. Haben Sie jedoch TASM oder TASMX für die Assemblierung benutzt, so müssen Sie TD zum Debuggen verwenden. Ganz allgemein: 32-Bit-Code und -programmierung haben nun sehr viel mehr Einzug in den Assembler von Borland gehalten. So wurden die verschiedenen Hilfsprogramme aufeinander ab- und eingestimmt, so daß zu erwarten ist, daß einige Probleme der Vergangenheit angehören. Aber Achtung: wer 32-Bit sagt, muß auch 32-Bit verwenden: TASM32, TD32 und – bloß nicht vergessen! – TLINK32 und TLIB32. Und denken Sie an TSTRIP32!

Hinzugekommen sind auch einige neue Assembleranweisungen, wie z.B. PUSHSTATE und POPSTATE. Diese Direktiven arbeiten so ähnlich wie PUSHCONTEXT und POPCONTEXT bei MASM 5.0 und werden, da sie den Rahmen dieses Buches sprengen, nicht weiter erläutert! Wichtige Erweiterungen sind die Anweisungen .586 und .587, mit denen, analog zu den Direktiven .8086 bis .487, die Besonderheiten der Pentiums berücksichtigt werden können (zu den genannten MASM-kompatiblen Angaben gibt es, natürlich, auch die IDEAL-Modus-kompatiblen P586 und P587). Sie können also ab sofort Programme schreiben, die nur noch mit Pentium-Prozessoren ausführbar sind!

Borland hat reagiert! Der Turbo Debugger TD, Version 4.0, stellt nun die Opcode-Sequenz für den Befehl CMPXCHG korrekt dar. Alle, die somit die neue Version 4.0 des Turbo Assemblers besitzen, können meine Anmerkungen in dem Kapitel über Tips und Tricks sowie bei der Beschreibung des Befehls im Referenzteil überlesen! Aber – warum wurden bei der Bereinigungscampagne CPUID und CMPXCHG8B vergessen? Zwar versteht TASM 4.0 die Direktive .586, die zur korrekten Assemblierung des Befehls notwendig ist, sowie das Mnemonic CPUID sehr wohl (TASM 3.2 meckert hier noch über unbekannte Befehle). Aber: zumindest in meiner Version des TD 4.0 wird die Sequenz 0F – A2 nicht erkannt! Analoges gilt für die anderen Pentium-Befehle. Die Bemerkung im Handbuch zu TASM 4.0, daß nun auch Pentiums unterstützt werden, ist somit ein wenig schöngefärbt!

Einige Angenehmlichkeiten wurden ebenfalls implementiert, wenn ihr Sinn auch manchmal etwas fragwürdig ist. So sind z.B. unter TASM 4.0 auch beim 8086 und seinen Rotationsbefehlen direkte Angaben der Anzahl zu rotierender Bits erlaubt. Da jedoch der 8086 dies nicht im Microcode ermöglicht, »übersetzt« TASM 4.0 z.B. den Befehl *ROR CX,3* in eine dreimalige Wiederholung des Befehls *ROR CX,1*. Doch: programmiert heute – vor allem, nachdem mit den neueren Windows- (for Workgroups, NT, Cairo) Versionen endgültig der Bruch mit den Prozessoren vor 80386 erfolgte – tatsächlich noch jemand für 8086er?

Dies sind wohl die wesentlichsten Neuerungen, die Sie interessieren dürften. Es gibt noch sehr viel Detailarbeit, die in der neuen Version steckt, wie z.B. DLLs, mit denen Turbo Debugger in Windows-Fenstern ablaufen kann, Windows NT- und OS/2- Unterstützung, erweitertes und verbessertes *Remote-Debugging* u.v.a. Darauf möchte ich jedoch im Rahmen dieses Buches nicht eingehen.

Wollen Sie Delphi-Programme debuggen, so werden Sie mit TDW 4.0 keinen Erfolg haben. Es sind auch heute noch Turbo-Assembler-Versionen auf dem Markt, denen lediglich TDW 4.0, nicht aber die Version TDW 4.6, beigelegt ist. Zur Bearbeitung von Delphi- oder Delphi-2-Programmen ist aber letztere Version absolut notwendig. Wenden Sie sich daher bitte im Zweifel an Borland. Hinweis: TDW 4.6 ist wirklich nur dann für Sie interessant, wenn Sie Delphi-Programme verwenden. Mit Delphi wurde eine neue Version des Linkers ausgeliefert, die TDW 4.0 noch nicht kennt. Arbeiten Sie dagegen »nur« mit BPW, so reicht Ihnen TDW 4.0 allemal – ja im Gegenteil: Nun meckert TDW 4.6 über einen falschen Linker, falls Sie BPW-Programme debuggen wollen!

TASM 5.0 ist der nun aktuelle Assembler von Borland. Er unterscheidet sich nur wenig von TASM 4.x. Die neuen Fähigkeiten beschränken sich auf eine verbesserte Kompatibilität zum Makro-Assembler von Microsoft und darauf, daß die 32-Bit-Version von TASM, TASM32, in der Lage ist, das sog. *flat thunking* unter Windows 95 zu unterstützen. Hierbei handelt es sich um eine Möglichkeit, Programmaufrufe zwischen 16- und 32-Bit-Modulen durchzuführen, und zwar in beiden Richtungen! Möglich ist dies nur unter Windows 95. Die Nützlichkeit dieses *features* sei dahingestellt ...

TDW 5.0 bietet nun Debugging-Fähigkeiten mit zwei Bildschirmen unter Windows 95, unterstützt die neuen C++-Konstrukte und bietet 16-Bit-Debugging unter Windows NT.

Die aktuelle MASM-Version von Microsoft ist 6.11. Allein schon an der Versionsnummer läßt sich erkennen, daß sich im Vergleich zu MASM 6.0 nichts Wesentliches verändert hat.

J Neue Stringtypen

Die Entwicklung vom alten Pascal-typischen String über pChar in *Borland Pascal 7.0* zu AnsiString zeigt beispielhaft, wie sich in den letzten Jahren die Softwarewelt insgesamt verändert hat. Neben dem eigentlichen Inhalt ist es entscheidend, die Länge des Strings zu kennen. Der hierfür notwendige Längenwert kann nun entweder direkt angegeben oder aber durch Abzählen der den String bildenden Zeichen erhalten werden.

Gibt man ihn direkt an, so ist es eine sehr gute Idee, ihn dem eigentlichen String voranzustellen, da Strings ja in ihrer Länge flexibel sein sollen, das Ende des Strings also variabel ist[17]. Nicht so sein Anfang. Diesen Weg gingen die Entwickler von Pascal. Strings sind nach dieser Philosophie Zeichenketten, die »nach außen« der Deklaration String = Array[1..x] of Char folgen, wobei »intern« ein Feldelement 0 existiert, das die jeweils gültige Länge des Strings codiert, aber »nach außen« nicht sichtbar ist. Anfangsadresse eines Strings ist immer das Element 0, also die Längenangabe, da diese zum String gehört und die einzige Information über dessen Länge und somit extrem wichtig ist. Das aber bedeutet, daß Code, der Strings benutzt, dies »wissen« und berücksichtigen muß.

Folgt man der anderen Philosophie, die beispielsweise in C realisiert ist, so beginnt jeder String tatsächlich mit dem ersten Zeichen, was man auch intuitiv erwarten würde. Das bedeutet jedoch, daß die Längenangabe anders zu erfolgen hat: Der sogenannte »null-terminierte« String war geboren. Darin ist das letzte Zeichen eines jeden Strings das Zeichen NULL (ASCII 0, nicht etwa Ziffer 0!). So endet per definitionem jeder String an der Stelle, an der das erste NULL-Zeichen der Zeichenreihe vorgefunden wird. Das aber bedeutet, daß die Länge des Strings nur festgestellt werden kann, indem man, an seinem Anfang beginnend, die Zeichen bis zum ersten NULL-Zeichen zählt.

Nun ist es reine Geschmackssache, welche Philosophie einem mehr liegt. Beide Versionen haben Vor- und Nachteile. Vorteil der Pascal-Strings: durch Auslesen des Elementes 0 ist sofort die aktuelle Stringlänge erkennbar, eine Längenveränderung kann durch Überschreiben dieses Elementes 0 mit der neuen Stringlänge erfolgen (was zwar nicht im Sinne der Pascal-Macher ist, die dazu Routinen geschaffen ha-

---

17 Bitte verwechseln Sie nicht die Variabilität der Strings mit dem Platzbedarf, den sie haben. Eine Variable vom Typ String benötigt 256 Bytes Platz, auch wenn der Inhalt nur fünf Zeichen beträgt, wie in „Hallo“ oder 22 wie in „Hallo, Ihr da draußen!“. Variablen vom Typ String[39] belegen immer statische 40 Bytes (39+Längenbyte!), egal welchen Inhalt sie haben – selbst wenn sie dynamisch reserviert wurden.

ben, dafür aber sehr einfach und schnell geht – eben »*quick and dirty*«). Nachteil: Strings können auf diese Weise nur maximal 255 Bytes lang sein – außer man definiert neue, die nicht ein Längenbyte, sondern ein Längenwort haben (was dann aber etwas komplizierter, wenn auch nicht unmöglich ist, wie wir sehen werden!). Vorteil der C-Strings: Sie können theoretisch beliebig lang sein. Ihr Nachteil: Stellen Sie einmal durch Abzählen die Größe eines 64-kByte-Strings fest! Das folgende Codefragment ist die Funktion StrLen, die das tut. Sie sehen, wie kompliziert das wird, vor allem, wenn man nun an mögliche 2 GByte denkt ...

```
CS : 07AF•55            PUSH   BP          ; Stackrahmen
     07B0 89E5          MOV    BP,SP       ;  einrichten
     07B2 FC            CLD                ; vorwärts suchen
     07B3 C47E06        LES    DI,[BP+06]  ; source laden
     07B6 B9FFFF        MOV    CX,FFFF     ; max. $FFFF chars
     07B9 30C0          XOR    AL,AL       ; NULL-Byte in AL
     07BB F2AE          REPNZ SCASB        ; suche Zeichen AL
     07BD B8FEFF        MOV    AX,FFFE     ; rechne CX in An-
     07C0 29C8          SUB    AX,CX       ;  zahl Zeichen um
     07C2 C9            LEAVE              ; entferne Stack
     07C3 CA0400        RETF   0004        ; zurück
```

Weiterer Nachteil: Das NULL-Zeichen kann nicht für andere Informationen verwendet werden (in Pascal-Strings sind Strings aus Nullzeichen möglich, wovon z.B. in Tabellen häufig Gebrauch gemacht wird!). Hinsichtlich des Platzbedarfs besteht kein Unterschied (die häufig zu findende Kritik, Strings in Pascal seien immer statisch im Datensegment, stimmt nicht ganz! Wer hindert mich daran, eine Variable S : ^String zu definieren und den Speicherplatz dynamisch anzufordern?).

Windows wurde zum großen Teil in C entwickelt, und aus diesem Grunde basiert Windows auf den C-Strings. Dies führte dazu, daß man Pascal und damit dann natürlich auch Delphi um einen Typ erweitern mußte, der zu C-Strings kompatibel ist. Eigentlich war das nicht besonders schwer, da Pascal ja Arrays kennt und somit ein Array of Char die Bedeutung hat, die null-terminierte Strings in C haben, wenn man dafür sorgt, daß das letzte Zeichen immer das NULL-Zeichen ist. Dennoch erwies es sich als sinnvoll, einen neuen Begriff zu prägen: pChar. Nun ist pChar nicht etwa gleichbedeutend mit einem Array of Char, sondern ein Zeiger darauf, also ein Zeiger, der auf einen null-terminierten String vom C-Typ zeigt. Um nun dem Anwender die Illusion zu lassen, pChars seien selbst null-terminierte Strings, lockerte man gleichzeitig die strikten Regeln von Pascal und erlaubte z.B. die direkte Zuweisung von Daten an solche Zeiger:

```
Var S : String;
    P : pChar;
begin
  S := 'Dies ist ein String ...';
  P := '... und dies eigentlich verboten';
end;
```

Der Compiler macht's möglich. Somit konnte man nun auch mit einfachen Mitteln Windows-Funktionalität in Pascal nutzen oder – besser – Pascal zur einfachen Windows-Programmierung verwenden.

Doch gehen wir noch ein wenig ins Detail. Schaut man sich an, was Delphi mit der Zuweisung des Strings an S macht, so findet man Gewohntes:

```
CS : 0179 BF5D01         MOV     DI,015D
     017C 0E             PUSH    CS
     017D 57             PUSH    DI
     017E BF460A         MOV     DI,0A46
     0181 1E             PUSH    DS
     0182 57             PUSH    DI
     0183 68FF00         PUSH    00FF
     0186 9AF22B1F4B     CALL    StrCopy
```

Der an CS:015D stehende String (eine vorgegebene Konstante) wird über eine Systemroutine physikalisch an eine Stelle (DS:0A45) im Datensegment kopiert, die S repräsentiert. Dazu werden beide Adressen sowie die Stringgröße (255) auf den Stack gelegt, und die Kopierroutine wird aufgerufen. Das erste Zeichen an beiden Adressen ist das Längenbyte des Strings.

Ich möchte an dieser Stelle nochmals deutlich darauf hinweisen, daß pChars Zeiger sind, auch wenn direkte Zuordnungen wie oben möglich sind und evtl. keine Allozierung von Speicher offensichtlich ist! Dies ist insbesondere dann wichtig, wenn pChars an in Assembler geschriebene Routinen übergeben werden sollen. Die notwendigen Verwaltungsaufgaben erledigt der Compiler im Hintergrund, um den Eindruck zu vermitteln, daß pChars eine Art String, eben null-terminierte Strings sind. In Wirklichkeit sind Variablen vom Typ pChar aber Zeiger auf solche Strings, wie die Zuweisung der Zeichenkette an die Variable P oben zeigt.

```
CS : 0167 B84000         MOV     AX,0040
     016A 8CDA           MOV     DX,DS
     016C A3420A         MOV     [0A42],AX
     016F 8916440A       MOV     [0A44],DX
```

Die Adresse des Strings (DS:0040) wird in die Variable P eingetragen, die groß genug (4 Bytes) ist, um solche Adressen aufzunehmen. Bei diesem String handelt es sich um eine statisch angelegte Konstante vom Typ Array of Char. Weil der Compiler dieses Array mit pChars in Verbindung zu bringen hat, wurde die String-Konstante vom Compiler null-terminiert. pChar zeigt auf das erste Zeichen des Strings. Sie werden feststellen, daß keinerlei irgendwie geartete Stringmanipulation wie z.B. ein Kopieren des Strings stattgefunden hat.

HINWEIS Spätestens dann, wenn Sie einmal einem pChar nicht eine Konstante wie oben zuweisen, sondern die Information aus anderen Quellen kopieren (wie z.B. aus Eingabefeldern in Dialogboxen) und keinen Speicherplatz bereitstellen, werden Sie schmerzlich erfahren, daß der Compiler nicht alles macht!

Mit den 32-Bit-Betriebssystemen ist eine neue Epoche angebrochen. Die ach so flexiblen null-terminierten Strings stoßen nun auch an ihre Grenzen, wenn theoretisch 2 GByte Speicher zur Verfügung stehen! Nachdem nun Strings theoretisch dazu herangezogen werden können, ganze Dokumente an Text aufzunehmen und somit praktisch einige MByte Umfang haben können, ist die Verwaltung mittels NULL-Zeichen und Abzählen der Länge nicht mehr machbar. Ausweg aus diesem Dilemma: ein Hybrid aus Pascal- und C-String.

Diese Chimäre ist als null-terminierter String beliebiger Länge definiert, wobei jedoch dem String bestimmte Informationen vorangestellt werden. Sie werden es erraten: Bei diesem Stringtyp handelt es sich um den in Delphi 2 neu definierten Typ AnsiString. Hier bleibt deutlich festzuhalten, daß auch diese Strings keine echten Strings sind, sondern Zeiger auf solche! Sie ähneln daher nicht nur in dieser Hinsicht den pChars, und die von Borland verbreitete Erklärung, AnsiStrings seien neue »Pascal-«Strings stimmt nur dann, wenn man den Begriff »Pascal-String« umdefiniert!

Trotz der zusätzlichen Information am Anfang bleibt es ein null-terminierter String und seine Adresse (also das, worauf die Variable vom Typ AnsiString zeigt) per definitionem der Ort des ersten Zeichens, nicht etwa der Stringlänge. Dies aber bedeutet, daß die weiteren Informationen an kleineren Adressen liegen (bezogen auf die Stringadresse!). Dort findet man folgende Informationen:

- Offset – 4: das Längen-Cardinal. Das bedeutet, daß nun 31 Bit Informationen zur Verfügung stehen, um die Länge des Strings zu definieren: die berühmten 2 GByte.
- Offset – 8: 4 Bytes (= Cardinal) Informationen über die Anzahl der sich auf diesen String beziehenden Referenzen (*Reference-Counter*).

Die wesentliche Information ist somit das Längen-Cardinal (analog zum Längenbyte in den »alten« Pascal-Strings – klingt irgendwie schön, oder?). Hier ist verzeichnet, wie lang der aktuelle String tatsächlich ist. Was aber hat es mit dem *Reference-Counter* auf sich? Delphi 2 ist auch in diesem Detail sehr sparsam. Während in Pascal – und auch in Delphi – jede Zuweisung eines Strings an eine Variable mit dem Kopieren dieses Strings einherging, ist dies nun vorbei. Bisher konnten 100 Exemplare des Strings »Das ist mir aber zu viel« im Speicher lagern, wenn der Originalstring ebensooft kopiert und an Variablen zugewiesen wurde. Ab Delphi 2 gibt es nur noch einen einzigen Originalstring. Kopien werden erzeugt, indem den entsprechenden Variablen die Adresse des Originalstrings übergeben und der *Reference-Counter* um 1 erhöht wird. Jedes Löschen einer Variablen, die auf diesen String zeigt, erniedrigt den *Counter* wiederum um 1. Nur dann, wenn der *Counter* bei 0 angelangt ist und damit anzeigt, daß keine Variable mehr auf den String zeigt, wird er endgültig eliminiert. Genial! Übrigens: String-Konstanten haben hier den Eintrag -1. Und noch etwas: was passiert, wenn zwei Variablen auf den gleichen String zeigen, der String aber von einer Variablen verändert werden soll? Dann und nur dann wird er tatsächlich kopiert, die beiden *Reference-Counter* werden korrigiert, und der zu modifizierende String wird modifiziert.

Jetzt fragt sich nur noch, warum AnsiString nun noch null-terminiert ist. Antwort: aus Kompatibilität zu pChar. So läßt sich aus einem AnsiString sehr einfach ein pChar machen:

```
Var S : AnsiString;
    P : pChar;

begin
  S := 'Dies ist ein null-terminierter String';
  P := pChar(S);
end.
```

Da ja alle Informationen, die pChar braucht, in AnsiString auch vorhanden sind, fallen keine großen Umwandlungsroutinen an: Es wird lediglich eine kleine Prüfung durchgeführt, was die Routine LStrToPChar übernimmt:

```
:0041F322 B868164200     MOV    EAX,00421668   ; Stringadres-
:0041F327 BA54F34100     MOV    EDX,0041F354   ; se in P mit
:0041F32C E8FF3EFEFF     CALL   @LStrAsg       ; LStrAsg
:0041F331 A168164200     MOV    EAX,[00421668] ; Stringadr.
:0041F336 E8DD41FEFF     CALL   @LStrToPChar   ; Check
:0041F33B A364164200     MOV    [00421664],EAX ; Adresse in P

   ;@LStrToPChar
:00403518 85C0           TEST   EAX,EAX        ; P = NIL ?
:0040351A 7402           JE     0040351E       ; (= Len=0)
:0040351C C3             RET                   ; nein -> Ende
:0040351D 00                                   ; NULL-Byte
:0040351E B81D354000     MOV    EAX,0040351D   ; ja -> neue
:00403523 C3             RET                   ; Adresse
```

LStrToPChar prüft lediglich, ob der String leer ist oder nicht. Wenn nicht, gibt es die in EAX übergebene Adresse des ersten Zeichens des null-terminierten Pascal-Strings zurück. Wenn er leer ist, so wird die Adresse des Bytes an Position $0040351D zurückgegeben. Dort steht ein NULL-Byte, was einem leeren pChar entspricht. Benutzt werden kann dieses *Type-Casting* überall dort, wo pChars erwartet werden (von Windows), aber AnsiStrings vorliegen (in Delphi 2). Der umgekehrte Fall funktioniert auch, doch überlasse ich es Ihnen, dies mit dem Debugger zu erforschen. Der Aufwand ist erheblich größer.

Schöne Sache, die neuen Stringtypen! Aber bitte bedenken Sie eines: Jeder dieser Strings hat, verglichen mit den *altmodischen* Pascal- oder C-Strings, einen Wasserkopf von acht Bytes. Bitte verwenden Sie sie nicht für Strings der Art »Groß genug«. In solchen Fällen tun es ShortStrings oder pChars auch.

Index

A

Adreßbus 6, 128, 910
Adresse 19, 140, 177
  Logische Adresse 179
  MOD-Byte 207
  Page 190
  Physikalische Adresse 196
  Präfix und OpCode 202
  SIB-Byte 208
  Virtuelle Adresse 189
Adreßleitungen 7, 115, 123, 128, 140
Adreßraum 6, 80, 120, 124, 140, 910
Affines Modell 85, 134, 910
Akkumulator 4, 22, 24, 56, 65, 70, 144
Alignment 12, 282, 296, 434, 453, 910
Anweisung
  .DATA? **866**
Anweisung
  .186 **858**
  .286 **858**
  .287 **859**
  .386 *295*, **859**
  .*387* 439, **860**
  .486 **860**
  .8086 *298*, **861**
  .8087 **861**
  .*CODE 453*, **864**
  .*CONST 453*, **865**
  .*DATA 453*, **865**
  .*DATA? 453*
  .*FARDATA 453*, **877**
  .*FARDATA? 453*, **878**
  .MODEL *453*, *455*, **889**
  .NO87 **890**
  .*STACK 453*, **896**
  \: **861**
  ; **862**
  = **863**
  ASSUME *256*, **863**
  CODE *256*
  COMMENT **864**
  DATA 252
  DB *253*, **866**
  DD *283*, **867**
  DF **868**
  DP **869**
  DQ **869**
  DT **869**
  DUP *274*
  DW *274*, **870**
  ELSE **871**
    ELSEIF **871**
  EMUL *438*, **872**
  END *265*, **872**
  ENDIF *319*, **873**
  ENDM **873**
  ENDP *292*, **874**
  ENDS 253, **874**
  EQU *325*, **875**
  ERR **876**
  EXTRN **876**
  FAR 285
  FOR **878**
  GLOBAL **879**
  GOTO **879**
  GROUP **880**
  IF **880**
    IFB **881**
    IFDEF **881**
    IFDIF **882**
    IFDIFI **882**
    IFE **882**
    IFIDN **883**
    IFIDNI **883**
    IFNB *319*, **883**
    IFNDEF **884**
    INCLUDE **884**
  IFIDNI *469*
  IPR *469*
  IRP **884**
  JUMPS *462*, **885**
  LABEL **886**
  LOCAL 458, *471*, **886**
  LOCALS *464*, **887**
  MACRO *319*, **888**
  MASM **888**

MASM51 **889**
NEAR *285*
NOEMUL **891**
NOJUMPS *462*, **891**
NOLOCALS **891**
NOMASM51 **892**
OFFSET *263*
OPTION **892**
EMULATOR *438*
LJMP *462*
NOLJMP *462*
NOSCOPED *463*
SCOPED *463*
PNO87 **892**
PROC *285*, **893**
PTR *276*
PUBLIC *271*, **894**
REPT **894**
SEG *262*
SEGMENT *252*, **895**
STACK *282*
STRUC *444*, **896**
UNION *448*, **897**
USES *460*, **897**
VERSION **898**
WHILE **898**
Arithmetische Operationen 50, 89, 105
ASCII-Zeichen 64, 74, 254, 310
ASCII-Ziffern 73
Assembleranweisungen *252*
Assemblerbefehle 252

B

BCDs 26, 28, 55, 71, 83, 108
Bedingte Assemblierung 319, 408, 911
Befehl
AAA *73*, **503**
AAD *76*, 319, **504**
AAM *75*, 311, 320, **505**
AAS *75*, **506**
ADC *51*, **507**
ADD 51, **510**
ADRSIZE **618**
AND *47*, 294, **512**
ARPL *125*, **514**
BOUND *121*, **515**
BSF *134*, **517**
BSR *135*, **518**
BSWAP *144*, **519**
BT *135*, **520**
BTC *135*, **521**
BTR *135*, **523**
BTS *135*, **524**
CALL 281, 307, **525**
CBW *78*, **527**
CDQ *138*, **528**
CLC *40*, **529**
CLD *41*, **529**
CLI *41*, 313, **530**
CLTS *125*, **530**
CMC *40*, **531**
CMOVcc 149
CMP *25*, 275, 294, **534**
CMPS *70*, 418, **536**
CMPSB 420, **536**
CMPSD *132*, **536**
CMPSW **536**
CMPXCHG *144*, 466, **538**
CMPXCHG8B *147*, **541**
CPUID *147*, 149, 472, **542**
CS **618**
CWD *78*, **545**
CWDE *138*, **545**
DAA *77*, **546**
DAS *77*, **547**
DEC *62*, **548**
DIV *59*, **549**
DS **618**
EMMS 169, **786**
ENTER *121*, **551**
ES **618**
F2XM1 *102*, **785**
FABS *108*, **689**
FADD *95*, **690**
FADDP *95*, **691**
FBLD *109*, **692**
FBSTP *109*, **693**
FCCOMI 149
FCHS *108*, **694**
FCLEX *112*, **695**
FNCLEX *112*, **695**
FCMOVcc 149
FCOM *97*, **698**, **699**, **700**, **771**, **772**
FCOMP *97*, **701**
FCOMPP *97*, **703**
FCOMIP 149
FCOS *138*, **704**

FDECSTP *110*, **705**
FDISI *112*, **706**
FNDISI *112*, **706**
FDIV *95*, **707**
FDIVP *95*, **708**
FDIVR *96*, **709**
FDIVRP *96*, **711**
FENI *112*, **712**
FNENI *112*, **712**
FFREE *110*, **713**
FIADD *106*, **713**
FICOM *106*, **715**
FICOMP *106*, **716**
FIDIV *106*, **717**
FIDIVR *106*, **719**
FILD **720**
FIMUL *107*, **721**
FINCSTP *110*, **722**
FINIT *112*, 273, **723**
FNINIT *112*, 273, **723**
FIST *107*, 304, **724**
FISTP *107*, 304, **725**
FISUB *107*, **726**
FISUBR *107*, **728**
FLD *91*, **729**
FLD1 *100*, 302, **739**
FLDCW *113*, **730**
FLDENV *113*, **733**
FLDL2E *100*, **736**
FLDL2T *100*, **737**
FLDLG2 *100*, **734**
FLDLN2 *100*, **735**
FLDPI *100*, 313, **737**
FLDZ *100*, 302, **738**
FMUL *95*, **740**
FMULP *95*, **742**
FNOP *114*, **743**
FNSTENV 274
FNSTSW 275
FPATAN *99*, **743**
FPREM *102*, **744**
FPREM1 *139*, **746**
FPTAN *99*, 431, **747**
FRNDINT *102*, **748**
FRSTOR *114*, **749**
FS **618**
FSAVE *114*, **750**
FNSAVE *114*, **750**
FSCALE *101*, **752**
FSETPM *126*, **753**
FSIN *138*, **754**
FSINCOS *138*, **755**
FSQRT *100*, **756**
FST *93*, **757**
FSTP **761**
FSTCW *113*, **758**
FNSTCW *113*, **758**
FSTENV *113*, 274, **759**
FNSTENV *113*, **759**
FSTSW *114*, *126*, **762**
FNSTSW *114*, *126*, **762**
FSUB *91*, *95*, **763**
FSUBP *95*, **765**
FSUBR *96*, **766**
FSUBRP *96*, **767**
FTST *111*, **768**
FUCOM *140*, **769**
FUCOMP *140*, **773**
FUCOMPP *140*, **774**
FUCOMI 149
FUCOMIP 149
FWAIT *115*, 303, **775**
FXAM *111*, **776**
FXCH *111*, **776**
FXRSTOR 169, **777**
FXSAVE 169, **779**
FXTRACT *100*, **782**
FYL2X *101*, 311, **783**
FYL2XP1 *101*, **784**
GS **618**
HLT *79*, **553**
IDIV *59*, **554**
IMUL *57*, *121*, *140*, **555**
IN *24*, **558**
INC *62*, **559**
INS *122*, **561**
INSB *122*, **561**
INSD *132*, **561**
INSW *122*, **561**
INT *265*, **563**
INTO **563**
INVD *145*, **564**
INVDPG *145*, **565**
IRET **566**
IRETD *132*, **566**
IRETW **566**
Jcc *33*, *132*, **532**, **567**
JA **568**

JAE **568**
JB **568**
JBE **568**
JC **568**
JCXZ **568**
JE **568**
JECXZ **568**
JG **568**
JGE **568**
JL **568**
JLE **568**
JNA **568**
JNAE **568**
JNB **568**
JNBE **569**
JNC **569**
JNE **569**
JNG **569**
JNGE **569**
JNL **569**
JNLE **569**
JNO **569**
JNP **569**
JNS **569**
JNZ 278, **569**
JO **569**
JP **569**
JPE **569**
JPO **569**
JS **569**
JZ **569**
JMP *32*, *132*, **570**
LAHF *40*, **573**
LAR *125*, **574**
LDS *19*, **575**
LEA *19*, 265, **576**
LEAVE *121*, **577**
LES *19*, **578**
LFS *132*, **579**
LGDT *125*, **581**
LGS *132*, **582**
LIDT *125*, **583**
LLDT *125*, **584**
LMSW *125*, **586**
LOCK *79*, **587**
LODS *65*, **588**
LODSB 419, **588**
LODSD *132*, **588**
LODSW 429, **588**
LOOP *37*, **590**
LOOPcc *37*, **590**
LOOPE **590**
LOOPNE **590**
LSL *125*, **591**
LSS *132*, **593**
LTR *126*, **594**
MOV *20*, *133*, **595**, **696**
MOVD 168, **787**
MOVQ 168, **787**
MOVS *68*, **599**
MOVSB **599**
MOVSD *132*, **599**
MOVSW **599**
MOVSX *133*, **601**
MOVZX *133*, **602**
MUL *56*, 304, **603**
NEG *63*, 315, **605**
NOP *78*, **606**, **674**
NOT *47*, **607**
OPSIZE **618**
OR 307
OR *47*, **608**
OUT *24*, **610**
OUTS *122*, **612**
OUTSB *122*, **612**
OUTSD *132*, **612**
OUTSW *122*, **612**
PACKSSDW 162, **789**
PACKSSWB 162, **789**
PACKSUWB 790
PACKUSWB 162
PADDB 158, **791**
PADDD 158, **791**
PADDSB 158, **793**
PADDSW 158, **793**
PADDUSB 158, **794**
PADDUSW 158, **794**
PADDW 158, **791**
PAND 167, **795**
PANDN 167, **796**
PCMPEQB 161, **797**
PCMPEQD 161, **797**
PCMPEQW 161, **797**
PCMPGTB 161, **798**
PCMPGTD 161, **798**
PCMPGTW 161, **798**
PMADDWD 160, **799**
PMULHW 159, **800**

PMULLW 159, **800**
POP *23*, 327, **614**
POPA *122*, **615**
POPAD *133*, **615**
POPAW **615**
POPF *40*, **617**
POPFD *133*, **617**
POPFW **617**
POR 167, **801**
Präfix **618**
prefix **618**
PSLLD 166, **802**
PSLLQ 166, **802**
PSLLW 166, **802**
PSRAD 166, **803**
PSRAW 166, **803**
PSRLD 166, **805**
PSRLQ 166, **805**
PSRLW 166, **805**
PSUBB 158, **806**
PSUBD 158, **806**
PSUBSB 158, **807**
PSUBSW 158, **807**
PSUBUDW 158
PSUBUSB 158, **808**
PSUBUSW 808
PSUBW 158, **806**
PUNPCKHBW 809
PUNPCKHDQ 164, **809**
PUNPCKHWB 164
PUNPCKHWD 164, **809**
PUNPCKLBW 164, **811**
PUNPCKLDQ 811
PUNPCKLDQ 164
PUNPCKLWD 164, **811**
PUSH *23*, *121*, 327, **620**
PUSHA *122*, **621**
PUSHAD *133*, **621**
PUSHAW **621**
PUSHF *40*, **623**
PUSHFD *133*, **623**
PUSHFW **623**
PXOR 167, **812**
RCL *45*, *121*, **624**
RCR *46*, *121*, **626**
RDMSR *148*, **628**
RDPMC 149, **629**
RDTSC *148*, **630**
REP *69*, **631**
REPNZ 419
REPZ 420
REPcc *69*, **631**
REPE **631**
REPNE **631**
RET 290, **633**
ROL *43*, *121*, **635**
ROR *43*, *121*, **637**
RSM *148*, **639**
SAHF *40*, **640**
SAL *45*, *121*, **641**
SAR *45*, *121*, **643**
SBB *51*, *55*, **645**
SCAS *70*, 418, **647**
SCASB 419, **647**
SCASD *132*, **647**
SCASW **647**
SETcc *133*, **649**
SETA **650**
SETAE **650**
SETB **650**
SETBE **650**
SETC **650**
SETE **650**
SETG **650**
SETGE **650**
SETL **650**
SETLE **650**
SETNA **651**
SETNAE **651**
SETNB **651**
SETNBE **651**
SETNC **651**
SETNE **651**
SETNG **651**
SETNGE **651**
SETNL **651**
SETNLE **651**
SETNO **651**
SETNP **651**
SETNS **651**
SETNZ **651**
SETO **651**
SETP **651**
SETPE **651**
SETPO **651**
SETS **651**
SETZ **651**
SGDT *125*, **652**

SHL *44*, *121*, 305, **653**
SHLD *136*, **655**
SHR *44*, *121*, 305, **657**
SHRD *136*, **659**
SIDT *125*, **660**
SLDT *125*, **661**
SMSW *125*, **662**
SS **618**
STC *40*, **663**
STD *41*, 424, **664**
STI *41*, 315, **664**
STOS *65*, **665**
STOSB **665**
STOSD *132*, **665**
STOSW **665**
STR *126*, **667**
SUB *51*, *55*, **668**
SYSENTER 150, **670**
SYSEXIT 150, **671**
TEST *29*, 303, **672**
UD2 149
VERR *126*, **674**
VERW *126*, **674**
WAIT *79*, 273, **676**
WBINVD *145*, **676**
WRMSR *148*, **677**
XADD *144*, **678**
XCHG *21*, **679**
XLAT *22*, **681**
XLATB *22*, **681**
XOR *47*, 298, **682**
Befehlssatz 16, 41, 87, 122, 138, 438
Bitschiebeoperationen 41
Boolesche Variable 47

C

CMOVcc
CMOVAB 533
CMOVAE 533
CMOVBE 533
CMOVC 533
CMOVE 533
CMOVG 533
CMOVGE 533
CMOVL 533
CMOVLE 533
CMOVNA 533
CMOVNAE 533
CMOVNB 533
CMOVNC 533
CMOVNE 533
CMOVNG 533
CMOVNGE 533
CMOVNL 533
CMOVNLE 533
CMOVNO 533
CMOVNP 533
CMOVNS 533
CMOVNZ 533
CMOVO 533
CMOVP 533
CMOVPE 533
CMOVPO 533
CMOVS 533
CMOVZ 533
CMOVcc 532
CMOVA 533
Codesegment 19, 177, 256, 257, 356, 372, 379, 452, 485, 911
Condition code *82*, 92, 97, 104, 111, 140, 276, 303
Control word 108, 112, 289, 480
CPUID 542

D

Datenbus 3, 6, 118, 130, 145, 912
Datenport 314, 912
Datensegment 12, 18, 141, 177, 252, 327, 355, 356, 359, 378, 379, 402, 427, 451, 452, 481, 484, 912
Debugger xxi, 82, 261
Descriptoren 124, 912
Deskriptoren 178
Disassemblat 262, 366, 382, 405, 439, 456, 912
DOS xxii, 7, 80, 123, 253, 278, 297, 309, 417, 427, 430

E

Ein- und Ausgabeoperationen 18
Emulation 437
Environment 113, 304
Exceptions 230, 816 Siehe auch Interrupts
Alignment Check Exception (#AC) 835
BOUND Range Exceeded Exception (#BR) 822

Coprozessor Segment Overrun 825
Denormalized Operand Exception (#D) 841
Device Not Available Exception (#NM) 823
Divide Error Exception (#DE) 818
Division-By-Zero Exception (#Z) 842
Double Fault Exception (#DF) 824
Exception-Klassen 816
Floating-Point Error Exception (#MF) 834
General Protection Exception (#GP) 830
Invalid Arithmetic Operand Exception (#IA) 840
Invalid FPU-Operation Exception (#IS, #IA) 838
Invalid OpCode Exception (#UD) 822
Invalid TSS Exception (#TS) 826
Machine Check Exception (#MC) 836
NMI Interrupt 820
Numeric Overflow Exception (#O) 843
Numeric Underflow Exception (#U) 844
Overflow Exception (#OF) 821
Page-Fault-Exception (#PF) 833
Precision Exception (#P) 846
Segment Not Present (#NP) 827
Stack Fault Exception (#SS) 829
Stack Overflow or Underflow Exception (#IS) 839

F

Flags 16, 33, 40, 60, 131, 275, 913
Aligment check flag 296
Alignment check flag *145*
Auxiliary flag *6*, 26, 55, 62, 64, 75
Busy flag *84*
Carry flag *6*, 26, 30, 33, 40, 44, 45, 49, 51, 55, 63, 64, 84
Denormalized operand flag *83*
Direction flag *6*, 41, 66, 68, 70
Exception summary flag *126*
ID flag *148*
Interrupt enable flag *6*, *41*, 120
Interrupt request flag *83*
Invalid operation flag *83*, 92, 109
IO-privileg level flag *132*
Nested tasks flag *132*
Overflow flag *6*, 26, 30, 33, 49, 55, 63, 64, *83*, 97
Parity flag *6*, 26, 30, 49, 55, 62, 64, 84
Precision flag *83*
Resume flag *132*
Sign flag *6*, 26, 30, 33, 49, 55, 60, 63, 64, 84
Single step flag *6*
Trap flag *6*
Underflow flag *83*
Virtual interrupt flag *148*
Virtual interrupt pending flag *148*
Virtual mode flag *132*
Zero divide flag *83*
Zero flag *6*, 26, 30, 33, 38, 49, 55, 62, 64, 69, 84, 97
Flat model 131, 141, 177, 295, 913
Funktion 299

G

Gates 223
Call-Gates 225
Gate-Deskriptoren 224
Interrupt-Gates 226
Task-Gates 225
Trap-Gates 226

H

Hardware-Interrupts 913
heap 141, 327, 452
High Byte 4, 16, 914

I

IDE 914
Instruction pointer 130, 303
Integrierter Assembler 914
Interrupt 41, 60, 79, 120, 265, 427
Interrupt controller 120, 128
Interrupts 229, 837 Siehe auch Exceptions
Interrupt Descriptor Table 230
Interrupt-Deskriptortabelle Register 232
Interrupt-Gates 226

K

Korrekturoperationen 71

L

Label 17, 20, 31, 255, 263, 301, 304, 321, 325, 374, 452, 462
Linker 914
Logische Operationen 46, 83
Low Byte 4, 16, 915

M

Memory-mapped I/O 15
Micro code 60, 915
MMX 151
  MMX-Befehle 157, 786
  MMX-Datenformate 152, 851
  MMX-Register 153, 955
Mnemonics xxi, 65, 102, 116, 320, 915
Modul xxii, 113, 285, 336, 373
Multitasking 214 Siehe auch Task

N

NaN 82, 92, 109, 140, 279, 302, 915
Nybble 28, 321, 916

O

Offset 11, 19, 22, 23, 65, 132, 258, 287, 309, 330, 421, 448, 477, 916
OOP 451
OpCode xxi, 105, 116, 119, 258, 266, 314, 357, 403, 466, 916
Operanden 17, 19, 23
OS/2 xxii, 123

P

Page 190
Page Table 194
Page Table Directory 196
Page Table Directory Register 196
Paging-Mechanismus 197
Pattern matching 417, 421
Pipelinig 917
Port 312, 917
Portadresse 24, 127
Privilegien 235
Privilegstufen 235
  current privileg level 236
  descriptor privileg level 236
  input/output privileg level 241
  requestor privileg level 238
Projektives Modell 85, 113, 134, 917
Protected Mode 10, 123, 133, 356, 379, 430, 917
Prozedur 281, 299, 342, 364

Q

Quelle 21
Queue 287

R

RAM 3, 17, 917
Real Mode xxii, 3, 10, 430, 917
Record 444
Register *3*, *80*, 87, *129*, 917
  Accumulator 4
  Adreßregister 946
  Allzweckregister 943
  Base pointer register 5, *130*, 329, 335
  Base register 4
  Codesegmentregister *5*
  Control word register 84, 112
  Counter register 4
  Data pointer register 85
  Data register 4
  Datensegmentregister *5*
  Debugregister 951
  Destination index register 5, 65, 68
  Extrasegmentregister *5*
  Flagregister 5, *130*, 944
  FPU-Register 954
  Indexregister *5*, 65, *130*
  Instruction pointer register 5, *85*, 303
  Kontrollregister 947
  Machine Status Word 946
  MMX-Register 955
  Rechenregister 3, *81*, *130*
  Segmentregister *5*, *129*, 946
  Source index register 5, 65, 68, *130*
  Stack pointer register 5
  Stacksegmentregister *5*
  Status word register 82, 113
Repetier-Befehle 69

S

Schleifenbefehle 37
Schutzmechanismen 233
  Die Page 240

Gates 223
Ports 241
Segmentgrößen 233
Segmenttypen 235
Zugriffsprivilegien 235
Segment 5, 7, 18, 65, 124, 252
Segmente 180 Siehe auch Adresse
Deskriptoren 180
Global Descriptor Table Register 184
Globale Deskriptortabelle 184
Local Descriptor Table Register 188
Lokale Deskriptortabellen 188
Segmentregister 186
Selektoren 186
Segmentgrenze 8, 918
Segmentregister 7, 22, 918
Speichersegmentierung 7, 11, 141, 265
Splines 414
Sprungbefehle 32
Stack xxii, 23, 40, 81, 86, 109, 141, 281, 327, 918
Stack pointer 84, 100, 276, 316
Stacksegment 918
Status word 98, 104, 126, 134, 289, 303, 481
Statuswort 273
Steuerport 314, 918
String 18, 64, 65, 122, 254, 321, 339, 419
Stringbefehle 24
Stringoperationen 64
Struct 444
Strukturen 444
System management mode 146
Systemtakt 311

T

Takt 918
Task 215, 919
Task Switches 220
Task-Gates 225
Task-Register 220
Task-State 216
Task-State-Segment 217
Task-State-Segmentdeskriptor 219
Timerbaustein 313
TOS 83, 88, 109, 302, 475, 919
Type casting 78, 276, 316, 321, 408, 449, 919

U

Umgekehrte Polnische Notaion 86
Unit 285, 352
Unterprogramm 280

V

Vergleichsoperationen 25
Virtual 8086 Mode xxii, 133, 919

W

Windows xxii, 123, 297, 356, 379, 430

Z

Zeiger 5, 10
Ziel 21