Das Assembler-Buch

Grundlagen, Einführung und Hochsprachenoptimierung

Die Reihe Programmer's Choice

Von Profis für Profis
Folgende Titel sind bereits erschienen:

Bjarne Stroustrup
Die C++-Programmiersprache
1072 Seiten, ISBN 3-8273-1660-X

Elmar Warken
Kylix – Delphi für Linux
1018 Seiten, ISBN 3-8273-1686-3

Don Box, Aaron Skonnard, John Lam
Essential XML
320 Seiten, ISBN 3-8273-1769-X

Elmar Warken
Delphi 6
1334 Seiten, ISBN 3-8273-1773-8

Bruno Schienmann
Kontinuierliches Anforderungsmanagement
392 Seiten, ISBN 3-8273-1787-8

Damian Conway
Objektorientiertes Programmieren mit Perl
632 Seiten, ISBN 3-8273-1812-2

Ken Arnold, James Gosling, David Holmes
Die Programmiersprache Java
628 Seiten, ISBN 3-8273-1821-1

Kent Beck, Martin Fowler
Extreme Programming planen
152 Seiten, ISBN 3-8273-1832-7

Jens Hartwig
PostgreSQL – professionell und praxisnah
456 Seiten, ISBN 3-8273-1860-2

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster
480 Seiten, ISBN 3-8273-1862-9

Heinz-Gerd Raymans
MySQL im Einsatz
618 Seiten, ISBN 3-8273-1887-4

Dusan Petkovic, Markus Brüderl
Java in Datenbanksystemen
424 Seiten, ISBN 3-8273-1889-0

Joshua Bloch
Effektiv Java programmieren
250 Seiten, ISBN 3-8273-1933-1

PEARSON Education

ADDISON-WESLEY

Trutz Eyke Podschun

Das Assembler-Buch

Grundlagen, Einführung und Hochsprachenoptimierung

eBook

Die nicht autorisierte Weitergabe dieses eBooks an Dritte ist eine Verletzung des Urheberrechts!

ADDISON-WESLEY

An imprint of Pearson Education Deutschland GmbH

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei der
Deutschen Bibliothek erhältlich.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

5 4 3 2 1

05 04 03 02

ISBN 3-8273-1929-3

© 2002 by Addison-Wesley Verlag, ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Str. 10-12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Christine Rechl, München
Titelbild: Polystichum falcatum, Sichelförmiger Punktfarn. © Karl Blossfeldt Archiv – Ann und Jürgen Wilde, Zülpich / VG Bild-Kunst, Bonn 2002
Lektorat: Christiane Auf, cauf@pearson.de
Korrektorat: Simone Meißner, Fürstenfeldbruck
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: text&form GbR, Fürstenfeldbruck
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer
Printed in Germany

Inhaltsverzeichnis

Vorwort 11

Einleitung 21

Teil 1: Einführung in die Assembler-Programmierung 27

1 Assembler-Befehle – Oder: was macht ein Compiler mit »I := 0«? 29
1.1 CPU-Operationen 30
1.1.1 Arithmetische Operationen 45
1.1.2 Logische Operationen 63
1.1.3 Operationen zum Datenvergleich 69
1.1.4 Bitorientierte Operationen 74
1.1.5 Operationen zum Datenaustausch 86
1.1.6 Operationen zur Datenkonvertierung 99
1.1.7 Verzweigungen im Programmablauf: Sprungbefehle 101
1.1.8 Andere bedingte Operationen 117
1.1.9 Programmunterbrechungen durch Interrupts/Exceptions 120
1.1.10 Instruktionen zur gezielten Veränderung des Flagregisters 125
1.1.11 Operationen mit »Strings« 127
1.1.12 Präfixe 134
1.1.13 Adressierungs-Befehle 141
1.1.14 Spezielle Befehle 143
1.1.15 Verwaltungs-(System-)Befehle 166
1.1.16 Obsolete Befehle 180
1.1.17 Privilegierte Befehle 182
1.1.18 CPU-Exceptions 183
1.2 FPU-Operationen 187
1.2.1 Grundlegende arithmetische Operationen 205

1.2.2	Trigonometrische Operationen	218
1.2.3	Andere transzendente Operationen	224
1.2.4	Operationen zum Datenvergleich und Datenklassifizierung	230
1.2.5	Operationen zum Datenaustausch	238
1.2.6	Operationen zur Datenkonversion	250
1.2.7	Verwaltungsbefehle	253
1.2.8	Obsolete Operationen	267
1.2.9	FPU-Exceptions	268
1.2.10	FPU-Emulation	271
1.3	SIMD-Operationen	272
1.3.1	SIMD, die Erste: MMX	274
1.3.2	MMX-Exceptions	306
1.3.3	MMX-Emulation	306
1.3.4	SIMD, die Zweite: SSE	307
1.3.5	SIMD, die Dritte: SSE2	343
1.3.6	Exceptions unter SSE/SSE2	360
1.3.7	Sind die SIMD verfügbar?	365
1.3.8	3DNow!, die Erste: das AMD-SSE	368
1.3.9	3DNow!, die Zweite: das AMD-SSE2	379
1.3.10	3DNow!, die Dritte: das Intel-SSE	382
1.3.11	Exceptions unter 3DNow!, 3DNow!-X und 3DNow! Professional	382
1.3.12	Ist 3DNow! verfügbar?	382
2	**Hintergründe und Zusammenhänge**	385
2.1	Stack	385
2.1.1	Der Stack – ein Stapel Daten	386
2.1.2	Stack frames – Verwaltung eines Stapels	389
2.1.3	Stack Switching	393
2.2	Speicherverwaltung	394
2.2.1	Speicherorganisation	394
2.2.2	Segmente	395
2.2.3	Die Betriebsmodi des Prozessors	399
2.2.4	Segmenttypen, Gates und ihre Deskriptoren	407
2.2.5	Deskriptorentabellen	427
2.2.6	Selektoren	429
2.2.7	Hardwareunterstützung für Deskriptoren und Deskriptortabellen	431

2.2.8 Zugriffe auf den Speicher: Von Adressen und Adressräumen 434
2.2.9 Beziehungskisten: Von der effektiven zur logischen Adresse 435
2.2.10 Speichersegmentierung: Von der logischen zur virtuellen Adresse 438
2.2.11 Paging: Von der virtuellen zur physikalischen Adresse 441
2.2.12 Auslagerungsdatei 457
2.2.13 Das 32-Bit-Betriebssystem Windows 458
2.3 Multitasking 462
2.4 Schutzmechanismen 467
2.4.1 Schutzmechanismen im Rahmen der Speichersegmentierung 468
2.4.2 Schutzmechanismen bei Zugriff auf die Peripherie 483
2.5 Exceptions und Interrupts 486
2.5.1 Interrupts 486
2.5.2 Exceptions 489
2.5.3 Interrupt-Behandlung 489
2.5.4 Emulation von Exceptions 498
2.5.5 CPU-Exceptions 499
2.5.6 FPU-Exceptions 529
2.5.7 SIMD-Realzahl-Exceptions 542
2.5.8 Interrupts und Exceptions im Real und Virtual 8086 Mode 552

Teil 2: Erzeugung und Verwendung von Assemblermodulen 555

3 Der Stand-Alone-Assembler 557
3.1 Vorbemerkungen 558
3.1.1 Datenbezeichnungen 558
3.1.2 Symbole 558
3.1.3 Expression 559
3.1.4 Qualifizierte Typen 560
3.1.5 Beispiele 560
3.2 Direktiven 561
3.2.1 Direktiven zur Datendeklaration 561
3.2.2 Direktiven zur Typ-Deklaration 570
3.2.3 Direktiven zur Symboldeklaration 598
3.2.4 Direktiven zur Daten- und Codeausrichtung 604

3.2.5	Direktiven zur Deklaration und Nutzung von Prozeduren	610
3.2.6	Direktiven zu Scope und Sichtbarkeit	621
3.2.7	Vollständige Segmentkontrolle	627
3.2.8	Vereinfachte Segmentkontrolle	639
3.2.9	Direktiven zur bedingten Steuerung des Programmablaufs	652
3.2.10	Makros	655
3.2.11	Bedingte Assemblierung	662
3.2.12	Direktiven zur Steuerung von Listings	666
3.2.13	Direktiven zur Anwahl des Befehlssatzes	672
3.2.14	Interaktion mit dem Programmierer	675
3.2.15	Assembler-Einstellungen	679
3.2.16	Verschiedenes	689
3.3	Operatoren	690
3.3.1	Operatoren in Ausdrücken	690
3.3.2	Operatoren für Strings	704
3.3.3	Run-Time-Operatoren	704
3.3.4	Operatoren in Makros	706
3.4	Vordefinierte Symbole	709
3.4.1	Vordefinierte String-Symbole (Textmakros)	709
3.4.2	Vordefinierte Symbole (Numerische Makros)	710
3.4.3	Makros zur Verwaltung von Strings	713
3.4.4	TASM-Symbole für OOP	714
3.5	Assemblermodule in Hochsprachen	714
3.5.1	Erzeugung des Assembler-Quelltextes	715
3.5.2	Assemblierung zum OBJ-File	719
3.5.3	Einbindung in Hochsprachen	720
3.5.4	Aufrufkonventionen	723
3.5.5	Übergabekonventionen	725
3.5.6	FAR und NEAR – eine Frage des Standpunktes	726
3.5.7	Tabus	726
3.6	Assembler und die strukturierte Ausnahmebehandlung (SEH)	729
4	**Der Integrierte Assembler**	745
4.1	Programmierung mit dem Inline-Assembler	745
4.2	Inline-Assembler und die strukturierte Ausnahmebehandlung (SEH)	760

Teil 3: Anhang 761

5 Anhang 763
5.1 Definitionen und Erläuterungen 763
5.1.1 Befehlssemantik 763
5.1.2 Adress- und Operandengrößen 765
5.1.3 Mnemonics, Befehlssequenzen, Opcodes und Microcode 768
5.1.4 Anwendungen, Programme, Module, Tasks, Prozesse und Threads 772
5.1.5 »Unschärfen« und Ungenauigkeiten in diesem Buch 776
5.2 Datenformate 778
5.2.1 »Little-Endian«- und »Big-Endian«-Format 781
5.2.2 Binäre Zahlendarstellung und Hexadezimalsystem 782
5.2.3 Elementardaten 788
5.2.4 Gepackte Daten 811
5.2.5 Erweiterte Elementardaten 814
5.2.6 Gegenüberstellung der verschiedenen Datenbezeichnungen 816
5.3 Speicheradressierung 816
5.4 Ports 827
5.5 Befehls-Decodierung 832
5.5.1 Decodierung des/der Präfixe(s) 832
5.5.2 Decodierung des Opcodes 832
5.5.3 Decodierung eines ModR/M- und ggf. eines SIB-Byte 833
5.5.4 Decodierung einer Adresse oder Konstanten 833
5.6 Tabellen zur Single-Instruction-Multiple-Data-Technologie (SIMD) 844
5.6.1 Unter SIMD auf Intel-Prozessoren verfügbare Datenformate 844
5.6.2 Unter SIMD auf Intel-Prozessoren verfügbare Instruktionen 845
5.6.3 Unter SIMD auf AMD-Prozessoren verfügbare Datenformate 850
5.6.4 Unter SIMD auf AMD-Prozessoren verfügbare Instruktionen 851
5.6.5 Entsprechungen und Unterschiede der Intel- und AMD-SIMD-Befehle 855
5.7 Weitere Register der CPU 856
5.7.1 Kontroll-Register 856
5.7.2 Debug-Register 863

5.7.3	Modellspezifische Register (MSRs)	867
5.8	FPU-, MMX- und XMM-Umgebung	868
5.9	Historie	874
5.9.1	Pentium 4	874
5.9.2	Pentium III, Xeon	874
5.9.3	Pentium II, Pentium II Xeon, Celeron	875
5.9.4	Pentium Pro	875
5.9.5	Pentium	877
5.9.6	80486	879
5.9.7	80386 / 80387	881
5.9.8	80286 / 80287	890
5.9.9	80186/80188	895
5.9.10	8086 / 8087	896
5.9.11	16-Bit-Protected-Mode	898
5.10	Verzeichnis der Abbildungen und Tabellen	900
5.10.1	Abbildungen	900
5.10.2	Tabellen	906
5.11	ASCII- und ANSI-Tabelle	911
Stichwortverzeichnis		913

Vorwort

Das Assembler-Buch entstand eigentlich Ende der achtziger Jahre. Damals hat einer meiner Freunde meine Loseblattsammlung für mich selbst angefertigter Notizen zur hardwarenahen Programmierung gesehen und mich danach geradezu genötigt, daraus ein Manuskript zu machen, das veröffentlicht werden sollte. Ich zierte mich ein wenig, da ich mir nicht vorstellen konnte, dass jemand an so etwas Interesse haben könnte, ich also keinen Bedarf sah! Aber steter Tropfen höhlt den Stein und so erschien 1993 noch vor dem ersten Pentium das Assembler-Buch im Verlag Addison-Wesley. Damals wäre ich zufrieden gewesen, wenn die Auflage innerhalb der nächsten Jahre ausverkauft worden wäre. Doch es kam anders! Schnell wurde ein Nachdruck notwendig, dann eine zweite Auflage, deren Nachdruck, zweiter Nachdruck und so weiter.

Auf diese Weise entstand ein Buch, das seit acht Jahren und vier Auflagen sehr erfolgreich auf dem Markt ist – mit ungebrochener Nachfrage und Akzeptanz, was mich sehr freut und stolz macht. Wie bei Neuauflagen üblich, wurden in ihnen die jeweils aktuellen Änderungen der Prozessoren und ihrer Befehlssätze – und damit des Assemblers – berücksichtigt. Das führte dazu, dass Struktur und Gliederung des Buches bis zu der vorliegenden Auflage gleich blieben: Besprechung der ersten Prozessoren von damals und Ergänzung der Änderungen und Neuerungen aktueller Prozessoren in neu aufgenommenen Kapiteln.

Das kleine Jubiläum und die mittlerweile doch recht drastischen Unterschiede der Programmierung von heute (Standard: 32-Bit-Systeme im protected mode) verglichen mit der von damals (Standard: 16-Bit-Systeme weitestgehend im real mode) haben mich dazu veranlasst, eine vollständig neu bearbeitete Auflage mit der Nummer 5 auf den Markt zu bringen, die eine andere Struktur aufweist: Das vorliegende Buch basiert auf dem derzeit aktuellen Intel-Pentium-4-Prozessor und seinen Möglichkeiten (mit ein wenig Abschweifen zum AMD-Athlon mit seinem 3DNow!-Instruktionssatz). Änderungen bei den einzelnen voran-

gehenden Prozessorgenerationen werden lediglich kurz erläutert und in den Anhang verbannt, was *ein* Ergebnis des *Feedbacks* meiner Leser ist. Treu geblieben bin ich jedoch der Art und Weise, wie ich dem Leser das Assemblerwissen nahe bringen möchte. Es ist nämlich meine Überzeugung, dass es sehr wohl einen Unterschied macht, verstanden zu haben, was man liest, oder es einfach nur zur Kenntnis genommen zu haben und zu hoffen, andere erledigen einem die Programmierarbeit.

Hierzu verwende ich kleine Programmbeispiele. In den vorangehenden Auflagen waren dies eine Reihe von Prograämmchen, die z.B. eine Erkennung und Unterscheidung der verschiedenen Prozessoren und Co-Prozessoren ermöglichten. Sie hatten keine große, über das eigentliche, didaktische Ziel hinausgehende Funktion, sondern sollten lediglich anhand konkreter Beispiele den Einsatz der Assembler-Befehle und -Anweisungen darstellen. Deshalb zeigen Kommentare wie »Wer schreibt überhaupt noch Programme für den 286 oder gar den 8086?«, dass der Betreffende das Wesentliche nicht verstanden hat: Es geht nicht um die Prozessordetektion! Um aber auch auf diesem Sektor neuen Wind in das Buch zu bekommen und nicht »ewig lang auf dem Co-Prozessor und der Unterscheidung der verschiedenen Typen herumgeritten« zu haben, wurden neue Beispiele verwendet, wie z.B. die Erkennung, ob der aktuelle Prozessor über Multimedia-Erweiterungen (SIMD) verfügt.

Die meisten meiner Leser der vergangenen Auflagen scheinen dies auch so zu sehen, wie die folgenden Äußerungen zeigen: »Im Gegensatz zu vielen anderen Titeln gibt das Buch eine wirklich gut verständliche präzise Einführung. [...] Der Autor beschränkt sich denn auch, in durchaus gelungener Weise, auf die Erklärung der wichtigsten Details.« – »Als ich dieses Buch durchgearbeitet hatte, war ich in der Lage in Assembler zu programmieren und meine Assembler-Module in C-Programme einzubinden. [...] Sicher benötigt man Zeit und Ausdauer, aber das liegt einfach in der Natur der Sache. Programmieren lernt man nicht mal ebenso.« – »Schon nach wenigen Tagen konnte ich mit diesem Buch Assembler-Routinen programmieren, obwohl es das erste Mal war, dass ich mich mit dem Thema Assembler befasst hatte.« Ein Leser bringt es auf den Punkt: »Eine bessere Einführung bzw. Vertiefung in die Materie kann man sich kaum wünschen ... Kaufts einfach!« Ganz seiner Meinung ;-)

Assembler im Zeitalter von RISC und CRISC?

Wir leben im Zeitalter der RISC- und CRISC-Prozessoren, bei denen der Prozessor und die Hochsprachencompiler eine intensive und nur schwer zu ersetzende Symbiose eingegangen sind. Das heißt nichts anderes als: Optimierten Code und höchste Performance, wie sie die *Reduced Instruction Set Computers* und *Complexity Reduced Instruction Set Computers* versprechen und wie sie heute einfach gefordert werden müssen, kann man nur mit der Kombination Hardware – darauf abgestimmter Compiler erreichen. »Handoptimierung« per Assembler führt hier in der Regel zum genauen Gegenteil: Zum Verlust einmal erreichter Performance, da es äußerst schwer ist, am eigenen Computer nachzukochen, was die Profiköche der Hardware- und Compilerschmieden in monate-, oft jahrelanger, intensiver Zusammenarbeit kreiert haben – das gesamte Know-how steckt im Compiler!

Noch zu den Zeiten der ersten Auflage des Assemblerbuches war das anders: Damals waren sog. CISCs die beherrschenden Prozessoren im PC-Bereich. Diese *Complex Instruction Set Computer* zeichneten sich dadurch aus, dass sie einen Befehlssatz hatten, dessen Befehle aus so genanntem Microcode bestanden. Dies können Sie sich so vorstellen, dass die eigentlichen Prozessorbefehle, um die es in diesem Buch geht und die das Ziel der Assembler-Programmierung, das Ergebnis der Compilerläufe und gleichzeitig der Input für den Prozessor sind, selbst nur eine Art »Hochsprache« auf Maschinenebene waren, die prozessorintern in die eigentlich verdrahteten Microcode-Befehle umgesetzt wurden. Auf diese Weise konnten sehr einfache (»ADD«), aber auch sehr komplexe (»SCAS«) Instruktionen realisiert werden, weshalb es auch zu dem Wort »complex« in CISC kam. Und je nachdem, wie komplex der Microcode war, der hinter den einzelnen Befehlen stand, dauerte die Ausführung entsprechend lange. Gemessen wurde dies in »Taktzyklen«. Da diese Prozessoren noch nicht mit mehrstufigen, parallel arbeitenden Pipelines zur Befehlsverarbeitung arbeiteten (auch nicht der 80486, selbst wenn er bereits Ansätze in die neue Richtung aufwies!), konnte man sehr wohl durch »Handanlegen« einiges an Performance gewinnen. Nicht umsonst waren gerade in Profiprogrammen viele zeitkritische oder ressourcenfressende Programmteile in Assembler geschrieben.

Wir leben heute! Und doch: RISC und CRISC zum Trotz gibt es auch heute noch genügend Gründe, Assembler zu benutzen, unter anderem auch, da die weit verbreiteten, auf Intels IA32-Architektur basierenden Prozessoren keine reinen RISC-Prozessoren sind – auch der Pentium 4

nicht. Sie haben zwar sehr viele RISC-Anteile (man spricht vom RISC-Core), weisen aber auch noch sehr viele CISC-Merkmale auf. Assembler ist eine Programmiersprache. Ja! Aber Assembler ist auch etwas Besonderes, hat Elemente, die ihn weit über jede andere Programmiersprache stellen. Ein solches Element ist: Flexibilität. Die Flexibilität des Assemblers fußt auf seiner archaischen Einfachheit, seiner absoluten Nähe zur Hardware, der Fähigkeit, im wahrsten Sinne des Wortes jedes Bit im Rechenwerk des Computers gezielt ansprechen und verändern zu können. Es gibt einfach keine andere Möglichkeit, direkt und direkter mit der Hardware zu kommunizieren – es sei denn, man legt an die Chip-Pins selbst Spannung an! Dies ist der Grund, warum Assembler auch heute noch eine wesentliche Rolle spielen – heute, wo es nicht mehr wie im Computer-Pleistozän auch unter ökonomischen Gesichtspunkten um die Schonung von Ressourcen (Speicher und Geschwindigkeit) gehen kann. Denn sowohl die Speicher- und Prozessorpreise als auch die Größe ansprechbarer Adressräume und die Taktraten moderner Prozessoren lassen diese Art der Assembler-Nutzung – als Optimierungstool – unnötig und überkommen erscheinen. (Einer meiner ersten Rechner hatte stolze 1 MByte RAM, eine 40 MByte Festplatte und einen 12 MHz-Prozessor samt, welch Luxus!, 8 MHz Co-Prozessor – und kostete schlappe 12.000 DM! Welcher Rechner mit 1,6 GHz, natürlich inklusive FPU und SSE2, 80 GByte Festplatte und 128 MByte RAM samt netter Kleinigkeiten wie 32 MByte Videospeicher, DVD-Laufwerk etc. kostet heute 12.000 DM?)

C++ hat seinen Erfolg und seine Popularität nicht zuletzt seiner Flexibilität zu verdanken und damit (augenscheinlich) genau den gleichen Voraussetzungen, wie sie auch der Assembler bietet. C++ ist vielleicht die dem Assembler am nächsten kommende Hochsprache, die mit Assembler vieles gemeinsam hat. Doch selbst C++ kann vieles nicht, was mit Assembler möglich ist. Denn C++ ist auch nichts anderes als eine Hochsprache und damit verschiedenen Voraussetzungen, Konventionen und Restriktionen unterworfen, die moderne Hochsprachen systembedingt nun einmal haben. (Schauen Sie sich einmal den Quelltext von professionell mit C++ und Delphi erstellten Programmen, ja selbst von C++- oder Delphi-Modulen an! Sie werden sich wundern, wie viele »_asm«- bzw. »asm«-Bereiche dort zu finden sind. Sie glauben es nicht? Dann durchforsten Sie z.B. einmal die in den »Professional«-Versionen enthaltenen Quellcodes der Systembibliotheken von Delphi und C++!)

Wer glaubt, bei der Programmierung moderner Software auf Assembler verzichten zu können, rückt schnell in die Nähe von Idealisten und

solchen, die nicht wissen, was sie tun (sollten). Aber auch: Wer ernsthaft glaubt, ein Betriebssystem oder ein anspruchsvolles Anwendungsprogramm vollständig in Assembler entwickeln zu können, hat Mut und verdient Respekt – muss sich jedoch auch ein klein wenig Größenwahn und Wichtigtuerei vorwerfen lassen – es sei denn, er gehört zu den drei, vier Genies dieser Welt und ihren zwei Dutzend Jüngern. Die Kunst ist, zu wissen, wann und wie der Assembler heute sinnvoll eingesetzt werden kann. Und nach meiner Überzeugung kann das nur unterstützend im Rahmen von Code-Fragmenten und -modulen, eingebettet in die optimierten Resultate heutiger Compiler sein.

In diesem Buch wird es daher darum gehen, Sie in die Programmierung mit Assembler *einzuführen* und Ihnen zu zeigen, wie Assemblerteile sinnvoll in Hochsprachenprogramme eingebettet werden können. Hierzu benötigen Sie erheblich mehr Informationen als das einfache »So erstellt man eine Assembler-Routine«. Dieses Buch versucht daher, neben der Einführung in die Assembler-Programmierung so viele dieser Hintergrundinformationen wie möglich zu geben.

Für wen ist dieses Buch nicht geschrieben, was kann es nicht?

Alle hierzu notwendigen Kenntnisse und Informationen zu vermitteln ist dieses Buch jedoch *nicht* in der Lage! Wollte jemand auch nur andeutungsweise diese Aufgabe lösen, käme sehr schnell eine Enzyklopädie heraus, die niemand mehr lesen würde. Von Hegel stammt der Satz: »Wer etwas Großes will, der muss sich zu beschränken wissen. Wer dagegen alles will, der will in der Tat nichts und bringt es zu nichts.« Dieses Buch will daher nicht ein weiteres Standardbuch zur Programmierung sein mit vielen mehr oder weniger nützlichen Routinen und Tipps und Tricks, wie man sie aus dem Internet zu Hunderten holen kann. Es will und kann daher auch keine Anleitung oder gar ein Rezept dafür sein, Betriebssysteme, Anwendungsprogramme oder auch nur Teile davon in Assembler zu programmieren. Das muss jeder Programmierer selbst tun: Sie! Es will und kann auch nicht eine Anleitung sein, wie man in Assembler genauso »optimierend« programmiert wie mit C++ oder Delphi – dazu müsste es erheblich tiefer selbst in Hardwarebelange (Architektur!) eintauchen, als es das schon tut. Es will vielmehr in eine *andere Art* der Programmierung *einführen*. In eine Art, in der man sich sehr wohl Gedanken darüber machen muss, wo welche Aktion mit welchen Daten wie abläuft.

Dieses Buch ist keine »Eier legende Wollmilchsau«, also ein Buch, das jeden befriedigt, der auch nur andeutungsweise etwas mit Assembler

zu tun hat oder haben möchte. Oder jede Frage beantworten könnte. Das soll es auch nicht! Es lässt jede Menge Raum für andere Bücher und Veröffentlichungen, die sich mit der Thematik beschäftigen sollen und wollen. Wer daher glaubt, er hätte mit dem vorliegenden Werk die Lösung für genau seine spezifischen Probleme gefunden, wird wahrscheinlich irren. Dieses Buch ist kein Rezeptbuch. Es stellt keine Lösungswege dar, es hilft einem nicht einmal dabei, Lösungen zu finden. Im Gegenteil: Sobald die Sache knifflig wird, zu sehr ins Detail zu gehen droht oder bestimmte, von vielen als wesentlich verstandene Bereiche ankratzt (»Wie programmiert man ein Chiffrierungsprogramm in Assembler?« oder »Was muss ich tun, um einen MP3-Dekoder zu programmieren?«) zieht sich der Autor mit dem Hinweis auf Sekundärliteratur elegant aus der Affäre – und aus der Schusslinie. Und genau das ist beabsichtigt. Ich kann Ihnen zeigen, wie Werkzeuge funktionieren und wie man sie einsetzt – benutzen müssen Sie sie!

Für wen also ist dieses Buch geschrieben?

Dieses Buch richtet sich daher an Fortgeschrittene und Profis – wenn man von der Hochsprachenprogrammierung kommt. Es ist kein Lehrbuch für Anfänger oder Neulinge, die erste Erfahrungen mit dem Programmieren als solchem sammeln: Beim Leser werden im Gegenteil gute Programmierkenntnisse und -erfahrung vorausgesetzt. Gleichzeitig wendet es sich an Einsteiger, Neulinge und wenig Erfahrene – wenn es um Maschinensprache geht. Es will erfahrene Programmierer in die Lage versetzen, neben den mächtigen Hochsprachen der heutigen Tage zusätzliche Werkzeuge an die Hand zu bekommen, mit denen man hoch flexibel arbeiten kann und die man nutzen muss, um moderne Software von heute zu erstellen. Insofern wird keinerlei Erfahrung mit dem Assembler vorausgesetzt.

Doch auch derjenige, der bereits Erfahrungen mit Assembler hat, kann dieses Buch sinnvoll nutzen. Es vermittelt viele Zusammenhänge und Hintergrundinformationen, die beim Einsatz von Assembler, aber auch von Hochsprachen hilfreich sein können. Oder wissen Sie bereits, warum Sie selbst dann in der Regel kaum Gelegenheit dazu haben werden, Ihr Programm in Bedrängnis zu bringen, wenn Sie mit Datenstrukturen arbeiten, die deutlich größer sind als der verfügbare RAM? Tipp: Das hat mit der Art und Weise zu tun, wie unter Windows die Umsetzung einer in der Hochsprache benutzen »logischen« Adresse (also einer Konstante oder Variable - »for I := «) in eine an den Festplatten-

kontroller weitergegebene »physikalische« Adresse erfolgt (Stichwort Segmentierung und Paging).

Und auch der absolute Profi kann von diesem Buch profitieren: So gibt es eine ausführliche Referenz (Band 2, Die Assembler-Referenz, Addison-Wesley, ISBN 3-8273-2015-1) aller Instruktionen, die die Prozessoren von heute beherrschen – natürlich auch die Multimediaerweiterungen wie MMX, SSE/SSE2 und 3DNow! Und es vermittelt auch die Unterschiede, die zu älteren Prozessoren – bis hin zum 8086 – bestehen.

Auf jeden Fall sollte der Interessierte folgendes Zitat eines meiner Leser der vierten Auflage beherzigen: »Das Assemblerbuch ist ein echt harter Brocken, es ist nicht einfach zu lesen und alleine der Umfang des Buches zwingt einen, Stunden damit zu verbringen. Aber nach mehrwöchigem Lesen habe ich nun das Gefühl, Assembler und den Aufbau von Intel-basierenden Prozessoren besser zu verstehen. [...], das Buch verlangt vom Leser selber einfach viel Durchhaltevermögen und den Willen zum Lernen. Aber wer das wirklich hat, macht mit dem Buch einen sehr guten Kauf.«

Das Assembler-Buch – jetzt in zwei Bänden

Vor allem die Ergänzungen, die die Prozessoren durch SIMD erfahren haben, waren dafür verantwortlich, dass *Das Assembler-Buch* in der 5. Auflage in zwei Bände geteilt werden musste – der Umfang ist einfach zu groß geworden. Wir wollten eben kein monströses, schlecht handhabbares Werk bei Ihnen ablegen, wie es leider oft genug erfolgt.

Im vorliegenden Assemblerbuch werden daher die einzelnen Befehle (Instruktionen) und Anweisungen (Direktiven) besprochen, die die Assembler von Microsoft und Borland verstehen. Dieses Buch liefert Ihnen darüber hinaus die Zusammenhänge, die Sie benötigen, wenn Sie heute (nicht nur mit Assembler) programmieren wollen.

Das noch in Auflage 4 vorhandene Kapitel »Referenz« dagegen wurde in den zweiten Band, *Die Assembler-Referenz*, ausgelagert. Sinn macht das aus zwei Gründen: Wenn Sie (nach ausgiebiger Lektüre des ersten Bandes?) genügend Kenntnisse besitzen, werden Sie vermutlich häufiger die Referenz benötigen und nur noch gelegentlich in *Das Assembler-Buch* schauen. Auf diese Weise arbeiten Sie mit einem sehr schlanken Werk, das sie immer zur Hand haben können. Der Verlag und ich gehen davon aus, dass dies in Ihrem Sinne sein wird.

Danke schön!

Wir leben, gerade was das Thema Computer betrifft, in einer sehr schnelllebigen Zeit. Besonders bewusst wird einem das, wenn man sich nach acht Jahren daran macht, ein neues Buch zu schreiben – und es mit dem alten vergleicht. Man denke: 1993 kam langsam der erste Pentium auf den Markt. Heute, 2002, sind wir beim Pentium 4. Dazwischen lagen der Pentium Pro, der Pentium II und schließlich der Pentium III. Fünf neue Prozessoren in acht Jahren, das sind rein rechnerisch alle 1,5 Jahre ein neuer Prozessor!

Auch an den Betriebssystemen kann man das ablesen! 1993 spielte das Betriebssystem DOS noch eine große Rolle, Standard war das 16-Bit-Windows 3.x. Heute sind wir über Windows 95/98/SE bei Millennium angekommen bzw., im Non-Consumer-Bereich, ausgehend von Windows NT 3.x über verschiedene 4er-Stufen bei Windows 2000 – die Folgeversion XP, die alles vereinheitlicht, wird derzeit ausgeliefert. Auch hier kann grob festgestellt werden: Alle 1,5 Jahre ein neues Betriebssystem.

Ein letztes Beispiel: 1991 kam Turbo Pascal for Windows auf den Markt – der erste Pascal-Compiler für Windows. Delphi 1.0 als Weiterentwicklung kam 1995 auf den Markt, dann Delphi 2.0 (1996), Version 3.0 (1997) – die erste 32-Bit-Version des Compilers, Delphi 4.0 (1998) und 5.0 (1999). Delphi 6.0 ist in diesem Jahr auf den Markt gekommen. Im Schnitt: alle 1,5 Jahre ein neuer Compiler.

Wenn ich diese Entwicklung so betrachte, gibt es gute Gründe, danke zu sagen. Und mein größter Dank gilt meinen Lesern, die in verschiedenster Weise dazu beigetragen haben, dass Sie mit diesem Buch die Version 5 des Assembler-Buches in den Händen halten. Die Leser sind die »große Konstante«, die ein Autor braucht, um sich in diesem schnelllebigen Geschäft über einen langen Zeitraum hinweg so erfolgreich auf dem Markt behaupten zu können – vor allem, wenn er diesen Job nicht hauptberuflich ausübt.

Allerdings schulde ich auch vielen Menschen großen Dank, ohne die dieses Buch nicht möglich gewesen wäre. Allen voran sind hier die vielen Mitarbeiter des Verlages zu nennen, die wesentlich zu der Realisierung des Buches beigetragen haben und die einen großen Anteil am Erfolg des Buches haben. Stellvertretend für alle diese Menschen möchte ich speziell meiner Lektorin Susanne Spitzer danken, die das Buch von der ersten Idee 1992 bis zu ihrer Baby-Pause (herzlichsten Glückwunsch an dieser Stelle!) vor wenigen Wochen begleitet hat und mit der

die Zusammenarbeit niemals langweilig wurde, weil sie mich in ihrer charmanten Art mit viel Witz und Humor immer dahin gebracht hat, wo sie mich haben wollte – auch dann, wenn mir eigentlich andere Dinge vorschwebten. Nicht weniger effektiv in dieser Hinsicht und nicht weniger angenehm war die Zusammenarbeit mit Christiane Auf, die mich während des größten Teils dieses Projektes betreut hat. Herzlichen Dank auch an Simone Meißner für das Debuggen meines Manuskriptes.

Eine andere, wesentliche »große Konstante« waren neben meinen Lesern und meiner Lektorin auch Menschen, die ich ebenfalls seit 1993 kenne und sehr schätze und die wesentlichen Anteil am Erfolg des Buches über einen solch langen Zeitraum haben. Insbesondere nennen möchte ich Martina Prinz von Borland/Inprise und Corinna Kraft von Wüst, Hiller und Partner, die immer dann für mich da waren und mich prompt bedienten, wenn ich Fragen zu Borlands Produkten (TASM, C++-Builder, Delphi) hatte. Auf Microsofts Seite nahmen diese Position Rainer Römer und Thomas Baumgärtner ein. Rainer danke ich vor allem auch deshalb sehr herzlich, weil ich immer dann genervt habe, wenn er es überhaupt nicht gebrauchen konnte und eigentlich gar nicht dafür zuständig war – und mir trotzdem half.

München, Dezember 2001 *Trutz Eyke Podschun*

Einleitung

Wissen Sie, was ein AGI ist? Nein? Vielleicht hilft Ihnen dann weiter, dass AGI für *address generation interlock* steht? Auch nicht? Aber was der Unterschied zwischen einer *u-* und einer *v-pipeline* ist und dass es Pipelinehemmungen gibt und wann sie auftreten, ist klar – oder? Dann sind Ihnen auch die Begriffe »Befehlspaarung« und »Paarungsregeln« nicht fremd und Sie kennen die Ausnahmen hiervon. Eher weniger? Aber so grundlegende Dinge wie *delay slots* und *branch prediction* mit Hilfe der *branch trace buffer* samt den dazugehörigen *delayed branches* und *delayed loads* darf ich doch zumindest ebenso als bekannt voraussetzen wie *write-back* und *write-through* sowie die *cache lines*! Denn ich gehe davon aus, dass Sie auch ausgiebig *performance monitoring* betreiben.

Nein? Gut! Dann sind Sie hier richtig! Denn wenn Sie mit diesen Begriffen »auf du und du stehen«, gehören Sie höchstwahrscheinlich zu dem Kreis Programmierer, dem ich mit diesem Buch nicht viel Neues sagen kann. Ich will nun nicht zu sehr in Details gehen und Ihnen erklären, was es mit all dem auf sich hat. Dazu gibt es sehr gute und ausführliche Literatur. Nur so viel: Wenn Sie vorhaben, Assembler über den in diesem Buch dargestellten Rahmen hinaus zu benutzen, dann müssen Sie sich mit all diesen Begriffen (und vielen mehr!) sehr gut auskennen. Und der Rahmen, den dieses Buch aufspannt, heißt: Assembler als Hilfsmittel beim Programmieren mit einer Hochsprache! Mehr kann dieses Buch nicht leisten und mehr soll es auch nicht leisten.

Im Vorwort habe ich bereits einige Punkte als Grund angesprochen. So können Sie heute ein Maximum an Performance aus dem Prozessor nur dann herausholen, wenn Sie die zwei (oder mehr) Integer-Pipelines, mit denen die modernen Prozessoren von heute arbeiten, optimal einsetzen. Hierzu müssen Sie wissen, wie diese arbeiten und welche Befehle auf welcher Pipeline bearbeitet werden können. Sie müssen ferner wissen, wie Sie die verschiedenen Pipelines so beschicken können, dass sie optimal parallel arbeiten können, ohne durch *address generation interlocks* oder andere Abhängigkeiten ausgebremst zu werden. Dies ver-

steht man unter »Befehlspaarung«, die nach bestimmten »Paarungsregeln« zu erfolgen hat – natürlich mit den entsprechenden Ausnahmen.

Doch Befehlspaarung ist nicht alles! Bedingt durch ein ausgeklügeltes *instruction prefetching* mit optimierter *branch prediction* kann es notwendig werden, Instruktionen im Befehlsstrom umzustellen. Das kann teilweise sehr merkwürdig anzusehende Konsequenzen haben: Das Laden eines Registers erfolgt im Befehlsstrom nach einem Sprungbefehl, obwohl es im Quellcode davor angesiedelt ist und nachher bei der Ausführung auch davor erfolgen sollte. Grund dafür ist eine weitere Optimierung: Aufgrund von *delayed branches* erfolgt eine Programmverzweigung erst *nachdem* z.B. der *folgende* Ladebefehl ausgeführt wurde. Wie das möglich ist? Dadurch, dass die Pipelines mehrstufig sind (z.B. 5 Stufen haben) und die sich einem Sprungbefehl anschließenden Instruktionen bereits in der Dekodierungsstufe der Pipeline befinden, während noch die Zieladresse in den weiter oben stehenden Stufen berechnet wird. Daher kann das Laden des Registers noch vor dem Sprung erfolgen, auch wenn der entsprechende Befehl im Befehlsstrom hinter dem Sprungbefehl steht. Vorteil: Die Pipeline wird optimal ausgenutzt, da nicht auf die neue, gerade zu berechnende Zieladresse gewartet werden muss, um die Pipeline gefüllt zu halten.

Und auch RISC trägt seinen Teil dazu bei. Viele »einfache« Befehle sind fest verdrahtet, kommen also ohne Microcode aus, wie er Grundlage der CISC-Technologie war. Doch nicht zuletzt aufgrund der Abwärtskompatibilität haben auch RISC-Prozessoren Microcodes. Sie kommen bei selten benutzten oder »komplexen« Instruktionen zum Einsatz. Nun kann es vorkommen, dass es sinnvoller ist, einen »komplexen« Befehl wie LODS (*load string*) in eine Folge von »einfachen« MOV-Befehlen umzusetzen. Doch wann ist das wirklich sinnvoll?

Solche Optimierungen, die die einzige Ursache dafür sind, dass im (sicherlich theoretischen) optimalen Fall bis zu zwei oder mehr Instruktionen (bei zwei Pipelines) gleichzeitig pro Takt ausgeführt werden können, können Sie von Hand nur sehr schwer durchführen. Dazu brauchen Sie eine Menge Erfahrung, Detailkenntnisse und Insiderinformationen, die vom Hersteller des Prozessors kommen. Sie sind aufgrund der bei RISC-Systemen absolut notwendigen engen Zusammenarbeit von Hardware- und Compilerherstellern materialisiert in den modernen Hochsprachencompilern von heute. Und Sie müssen ausgiebig *performance monitoring* betreiben, was eventuell sogar spezielle Hardware voraussetzt, die Sie hierbei unterstützt, wollen Sie das kopieren. Das heißt nicht mehr und nicht weniger als: Wenn Sie versuchen,

an einem Compilat etwas durch vermeintliches Assembler-Optimieren zu verbessern, oder wenn Sie der Meinung sind, das alles selbst und in Assembler zu können, verschlimmbessern Sie das Ergebnis mit sehr hoher Wahrscheinlichkeit. Konsequenz: Verlust an Performance.

Warum dann überhaupt noch Assembler? Weil es viele Dinge gibt, die Sie dennoch tun können, um ein Programm zu optimieren. Denn auch die Nutzung von SCAS und LODS, solchen »komplexen« Befehlen, kann Performance steigern. Denn sie wurden vom Prozessorhersteller so optimiert, dass sie in den Pipelines optimal ausgeführt werden. Und manchmal (SIMD!) führt ja gar kein Weg daran vorbei ...

Soweit die Hardwareseite. Kommen wir noch kurz zum Betriebssystem. DOS hatte einen schlechten Ruf unter anderem deshalb, weil es keine Schutzkonzepte hatte. Auch die Customer-Versionen von Windows, Windows 3.xx, 9x und ME, werden von vielen verschmäht, weil sie »instabil« laufen und »leicht abzuschießen« sind. Wodurch? Durch »unsauber« programmierte Programme, die glauben, sich an Konventionen nicht halten zu müssen. Oder durch alte DOS-Programme, in denen sowieso jeder Programmierer das gemacht hat, was er wollte. Die Professional-Versionen Windows NT und 2000 haben da einen etwas besseren Ruf. Ursache: die Art und Tiefe der angelegten Schutzkonzepte. Es macht daher überhaupt keinen Sinn, Assembler nun einsetzen zu wollen, um diese betriebssystembedingten und notwendigen Errungenschaften auszuhebeln.

Dieses Buch wird Ihnen daher nicht dabei helfen, Programme oder Module zu entwickeln, die geeignet sind, die Schutzkonzepte zu umgehen. Wer sich darüber beklagt, dass ich kein Rezept dazu angeben werde, wie man wie zu guten alten DOS-Zeiten die Interrupts verbiegt und damit eigene Interrupts ermöglicht, oder wer moniert, dass ich keine Anleitung zum direkten Ansprechen von I/O-Ports gebe, hat nicht verstanden, worum es geht. Wer bemängelt, dass ich bestimmte Instruktionen oder Register nicht weiter erläutere oder beschreibe, ebenso wenig. Wer also unbedingt einen eigenen Exception-Handler schreiben oder das Betriebssystem »aufbohren« will, darf das gerne tun – jedoch muss er sich die Kenntnisse hierzu aus anderen Quellen holen. Denn Exception-Handler sind üblicherweise eine Angelegenheit des Betriebssystems und eng daran gebunden. (Wie eng, kann man z.B. daran erkennen, dass die Intel-Prozessoren gewisse SIMD-Instruktionen, obschon sie implementiert sind, nur dann unterstützen, wenn das Betriebssystem einen entsprechenden Exception-Handler zur Verfügung

stellt und dies in einem geschützten Bit eines geschützten Registers des Prozessors vermerkt! Wir werden darauf zurückkommen.)

Wenn ich Ihnen also über das eigentliche Thema »Assembler« hinaus noch weitergehende Informationen gebe, wie z.B. die Speichersegmentierung, den Paging-Mechanismus oder die Schutzkonzepte mit den privilege levels, so dient dies ausschließlich dem Zweck, Ihnen die Kenntnisse zu vermitteln, warum was wie funktioniert – oder eben nicht! Und wo Grenzen sind, die zu respektieren sind. Daher erhebe ich auch keinen Anspruch darauf, Ihnen *alle* Informationen zukommen zu lassen, die Sie interessieren könnten.

Der Rahmen, in dem sich alles in diesem Buch abspielen wird, ist der privilege level 3: Anwendungsprogramme. Kernel (privileg level 0) und andere Teile des Betriebssystems und/oder Module, die sich in niedrigeren levels als 3 ansiedeln, sind für mich tabu! Ansonsten könnten wir ja alle zurück zum DOS.

Noch einige Anmerkungen. Der Mensch ist ein optisches Wesen und arbeitet gerne mit Symbolen. Dem möchte ich dadurch Rechnung tragen, dass ich in diesem Buch mit vielen Abbildungen und Tabellen arbeiten werde und auch andere optische Stilmittel einsetze. Eines davon ist eine Marginalspalte, die innerhalb von Kapiteln als »Kompass« dienen soll, sich zurechtzufinden. Sie beherbergt auch ein zweites Stilmittel, nämlich die optische Hervorhebung einzelner Textpassagen. Ich verwende hierzu Icons mit bestimmten Bedeutungen. Diese stelle ich Ihnen nun vor.



Das »Stop«-Icon soll bewusst den Lesefluss unterbrechen. Es markiert Stellen mit der Beschreibung von Voraussetzungen, Einschränkungen, Ausnahmen oder schwerwiegender oder schwer aufzufindender Fallen.



Dieses Icon steht für »Achtung«. Es markiert Passagen, an denen der Leser besonders aufmerksam auf den Inhalt achten sollte. Stellen, die mit diesem Icon markiert sind, beinhalten häufig Fallen oder wesentliche Zusatzinformationen.



Das »Hinweis«-Icon ist an Stellen zu finden, an denen weitergehende, zusätzliche Informationen gegeben, Bezüge auf verwandte Themen(-kreise) hergestellt oder Sachverhalte angesprochen werden, die nur mittelbar mit der aktuellen Thematik zu tun haben.

Mit diesem Icon werden Textstellen markiert, in denen Tipps und Tricks angegeben werden. Das kann zum Beispiel die »Zweckentfremdung« von Assembler-Befehlen für Probleme sein, für die sie nicht konzipiert wurden, oder auch nur der Hinweis auf Lösungen für spezifische Fragestellungen.



Das C++-Builder-Icon markiert Textpassagen, die speziell auf Themen hinweisen, die unter C++ eine Rolle spielen. So sind z.B. alle Teile mit diesem Icon versehen, in denen Assembler-Routinen in CBuilder eingesetzt oder De-Compilate des C++-Compilers von Borland besprochen werden. Diese Textstellen sind selbstverständlich auch beim Einsatz des C++-Compilers von Microsoft interessant.



Analog markiert das Delphi-Icon die Stellen im Text, an denen Delphi-spezifische Themen behandelt werden, wie z.B. die Einbindung von Assembler-Modulen oder De-Compilate der Delphi-Compiler.



Mit Borlands Chip-Icon werden Textteile markiert, in denen es spezifisch um den Makro-Assembler (TASM) von Borland geht.



Mit dem Icon von Microsoft *programmers work bench* (PWB) werden Textteile markiert, in denen es spezifisch um den Makro-Assembler (MASM) von Microsoft geht.



Dieses Icon weist Sie auf ergänzende Daten hin, die Sie auf der beiliegenden CD finden.



Teil 1: Einführung in die Assembler-Programmierung

1 Assembler-Befehle – Oder: was macht ein Compiler mit »I := 0«?

Als Hochsprachenprogrammierer – egal, ob man nun in C++, Delphi oder den anderen hoch spezialisierten oder etwas angestaubten Programmiersprachen programmiert, ja selbst unter dem interpretierenden Basic ist das so – macht man sich keine Gedanken, was eigentlich im Herz des Computers abläuft, wenn man eine so simple Zuordnung einer Konstanten, hier »0«, an eine Variable, hier »I«, programmiert. Muss man auch nicht! Wichtig ist lediglich, dass man weiß, dass irgendwo in den Katakomben des Rechners ein kleines Stückchen dotiertes Silizium reserviert ist, das man dazu benutzen kann, ihm vorübergehend einen Namen (»I«) und einen Wert (»0«) zu geben, und das bereitwillig die Information wieder abgibt, so man es unter dem eben vergebenen Namen mit dem entsprechenden Hochsprachenbefehl dazu auffordert. Wie das dann in eine Form gebracht wird, die der Prozessor dann auch verstehen und entsprechend umsetzen kann, interessiert bereits nicht mehr: Das ist Sache der Compiler – wozu hat man die sonst? Hochsprachenprogrammierer haben ihr Augenmerk auf die drei wesentlichen Kernpunkte gerichtet, die jedem Programm gemein sind: Problem – Lösungsansatz – Realisierung. Und dies spielt sich hauptsächlich auf einer Ebene ab, die Spielwiese der modernen Hochleistungscompiler von heute ist. Details stören hier nur!

Doch unter bestimmten Gesichtspunkten wird es dann notwendig, tiefer hinabzusteigen in die Tiefen der Hardware und ihrer Programmierung. Und plötzlich, als hebe sich ein Vorhang, ist der Fokus ein ganz anderer. Plötzlich sind solche Fragen wichtig wie »Bearbeite ich die Fließkommazahl *F* nun in den FPU-Registern oder besser skalar in den XMM-Registern?« oder »Kann ich diesen bedingten Sprung irgendwie vermeiden? Er mindert die Performance!«

Moderne Prozessoren von heute bestehen, betrachtet man die Situation aus einem bestimmten Blickwinkel, aus drei Einheiten: der *Central Processing Unit* (CPU), deren Aufgabe im Bereich der Verarbeitung von Integer-Daten liegt (was an dieser Stelle ganz allgemein gehalten werden soll: Auch die Befehlsinstruktionen, mit denen der Prozessor arbeitet, sind Integer-Daten, Bytes genannt!), der *Floating Point Unit*, zuständig für Fließkomma-Berechnungen, und den Komponenten, die für Multimedia-Anwendungen erforderlich sind (SIMD). Alle diese drei Bereiche können als (mehr oder weniger) unabhängige, selbstständige Einheiten betrachtet und besprochen werden.

1.1 CPU-Operationen

CPU-Datenformate

Wenn Sie von der Hochsprachenprogrammierung herkommen, vergessen Sie bitte ab jetzt alle Datendefinitionen, die Ihnen dort untergekommen sind. Der Hintergrund ist ein einfacher: Jede Hochsprache und jede neue Version einer Hochsprache definiert Daten nach Kriterien, die im Rahmen der Hochsprache und ihren Randbedingungen Sinn machen, die aber im Rahmen des Assemblers nicht immer nachvollzogen werden können.

Ein Beispiel: Unter Delphi 2.x war die gute, alte *Integer* definiert als vorzeichenbehaftete Ganzzahl, der 16 Bit zur Codierung zur Verfügung standen. Diese 16 Bit resultierten aus der Breite der damals verwendeten Prozessorregister einerseits und dem darauf aufbauenden (16-Bit-)Betriebssystem andererseits. So konnte diese Integer-Werte zwischen -32.768 und +32.767 annehmen. Mit Aufkommen der 32-Bit-Prozessoren und den entsprechenden Betriebssystemen konnten nun vorzeichenbehaftete Ganzzahlen zwischen -2^{32} und $+2^{32}-1$ verwaltet werden. Und so wurden diese neuen 32-Bit-Ganzzahlen schnell zum neuen »Standard« erklärt. Damit nun die Portierung der »alten« 16-Bit-Programme in die »neue« 32-Bit-Umgebung möglichst schnell und problemlos erfolgen konnte, wurde kurzerhand in Delphi 3.x die *Integer* als vorzeichenbehaftete 32-Bit-Ganzzahl umdefiniert und damit der *LongInt* gleichgesetzt. (Und ich bin sicher: Mit Einführung des 64-Bit-Prozessors Itanium von Intel, dem bereits avisierten 64-Bit-Betriebssystem von Microsoft – wohl auch mit Namen Windows – und somit einer neuen Runde an Software-Updates ist dann unter Delphi X.x die *Integer* eine 64-Bit-Ganzzahl.) Der Rest war einfach: Die reine Neu-Compilierung des Quelltextes führte nun (zumindest theoretisch! Die Tücke lag wie immer im Detail.) zu einem vollständig kompatiblen 32-Bit-Programm. Ohne dass eine Befehlszeile (zumindest was die Integers be-

trifft) geändert werden musste. Denn nun lud die CPU den Wert 4711 nicht mehr als 16-Bit-Integer in ein 16-Bit-Register, sondern als 32-Bit-Integer in ein 32-Bit-Register!

Diesen unter den genannten Bedingungen sicherlich sinnvollen »Modeerscheinungen« kann der Assembler nicht folgen. So kennt er noch nicht einmal die Unterscheidung zwischen vorzeichenbehafteten und vorzeichenlosen Integers: Für ihn gibt es, abgeleitet von den Daten, die die Prozessoren kennen, nur Byte-Daten (*define bytes*; DB), Word-Daten (*define words*; DW), DoubleWord-Daten (*define double words*; DD) und QuadWord-Daten (*define quad words*; DQ), die man definieren kann. Ob die vorzeichenbehaftet sind, interessiert weder den Prozessor noch den Assembler – hier ist die Interpretationsfähigkeit des Programmierers gefragt. Wir werden darauf noch zurückkommen. Um aber das Lesen dieses Buches nicht zu einer Gewalttour zu machen, werden seit langem eingeführte und probate Datenformate verwendet. Es sind im Falle von vorzeichenlosen Ganzzahlen die *Bytes* (8 Bits), *Words* (16 Bits), *DoubleWords* (32 Bits) und *QuadWords* (64 Bits) sowie im Falle der vorzeichenbehafteten Ganzzahlen die *ShortInts* (7 Bits + Vorzeichen), die *SmallInts* (15 Bits + Vorzeichen) und die *LongInts* (31 Bits + Vorzeichen). Da die zu den QuadWords analogen *QuadInts* (63 Bits + Vorzeichen) noch nicht aufgetaucht sind (die CPU-Register sind »nur« 32 Bits breit!), gibt es diese Integer zurzeit nur im Rahmen von SIMD (siehe unten). Diese Daten werden in diesem Buch als »Elementardaten« bezeichnet. Es kommen noch die einfachen und gepackten BCDs hinzu – worum es sich hier handelt, entnehmen Sie bitte genauso wie weitere Einzelheiten über die Darstellung der genannten Daten dem Kapitel »Datenformate« auf Seite 778.



Bitte beachten Sie auch, dass der Begriff »Integer« mehrfach belegt ist: So dient er als Oberbegriff für alle vorzeichenbehafteten Ganzzahlen und darüber hinaus auch für alle Ganzzahlen schlechthin. Das ist zwar bedauerlich, resultiert jedoch aus dem englischen Sprachgebrauch (der üblicherweise nicht zwischen »signed integers« und »unsigned integers« unterscheidet) und sollte eigentlich aufgrund des jeweiligen Kontextes nicht zu Problemen führen.

CPU-Basis-Register

Zur Bearbeitung der eben besprochenen Daten besitzt der Prozessorchip Strukturen, die man gemeinhin als »Register« bezeichnet. Diese Register haben die unterschiedlichsten Aufgaben: Sie können Daten mit logischen oder arithmetischen Instruktionen bearbeiten, sie können Informationen über den aktuellen Zustand des Prozessors darstellen oder

Informationen entgegennehmen, die die Aktivitäten des Prozessors steuern, oder sie können Adressen und Indices aufnehmen, die bei der Kommunikation mit der Peripherie des Prozessors eine Rolle spielen.

Gemäß ihrer Aufgabe sind die Register des Prozessors eingeteilt in die

- Allzweckregister, die die Operanden für die arithmetischen oder logischen Operationen aufnehmen oder Zeiger, die bei gewissen Befehlen eine Rolle spielen, über die die Kommunikation mit der Peripherie erfolgt. Die modernen Prozessoren der Pentium-4-Familie (und deren Klone) besitzen acht solcher Register.
- Segmentregister, die beim Datenaustausch des Prozessors mit seinem Speicher zum Tragen kommen. Die Pentium-4-Prozessoren besitzen sechs dieser Register.
- Programm-Status- und -Kontroll-Register. Sie dienen der Steuerung des Programmablaufs sowie der Darstellung des aktuellen Programmzustands. Pentium-4-Prozessoren haben ein solches Register.
- Register, die die Adresse des nächsten auszuführenden Befehls im Programmablauf beinhalten. Prozessoren der Pentium-4-Familie besitzen ein solches *instruction pointer register*.

Abbildung 1.1 zeigt Ihnen die Basisregister eines Pentium 4:

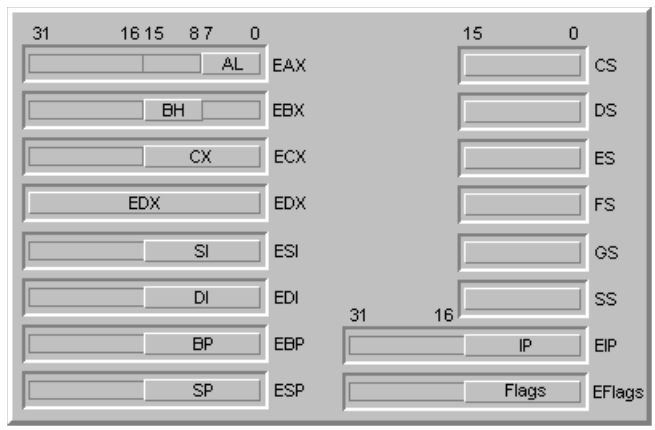


Abbildung 1.1: Die grundlegenden Register der CPU: Allzweck-, Segment-, Adressierungs- und Status-Register

Auf der linken Seite der Abbildung sind die acht 32-Bit-Allzweckregister dargestellt, die rechte zeigt die sechs 16-Bit-Segmentregister sowie das 32 Bit breite Status- und Kontrollregister EFlags und das ebenfalls 32 Bit breite »Befehlszeiger«-Register EIP.

Allzweckregister

Die Namen der Allzweckregister stammen traditionell noch aus der Zeit, in denen sie für bestimmte Aufgaben spezialisiert und lediglich 16 Bit breit waren. So ist der *Extended Accumulator* EAX aus dem *Accumulator* AX entstanden, dessen Hauptaufgabengebiet die arithmetischen Operationen waren. Das *Extended Base register* EBX entsprang dem *Base register* BX und diente als Heimat einer Basisadresse, die bei der indirekten Adressierung eine Rolle spielte (vgl. das Kapitel »Speicheradressierung«). ECX, das *Extended Counter register* diente in Form seines Vorläufers, des *Counter registers* CX, hauptsächlich der Steuerung von Programmschleifen, während das *Data register* DX, das dem *Extended Data register* EDX zugrunde liegt, zusätzliche Daten aufnahm, die entweder während verschiedener Zwischenstufen einer Berechnung entstanden oder im Rahmen verschiedener Instruktionen benötigt wurden.

Heute gibt es diese Unterscheidung nicht mehr – zumindest was die meisten Fähigkeiten betrifft. Alle acht Register, also EAX, EBX, ECX und EDX sowie ESI, EDI, EBP und ESP, sind absolut gleichberechtigt und können beliebig ausgetauscht und zu allen nur denkbaren Operationen (Arithmetik, Berechnung indirekter Adressen, Zeiger auf Speicherstellen) verwendet werden. Doch es existieren zwei Register, die mehr oder weniger als tabu gelten und für ganz bestimmte Zwecke eingesetzt werden: das *Extended Base Pointer register* EBP und das *Extended Stack Pointer register* ESP. Sie dienen der Verwaltung einer Datenstruktur, auf die der Prozessor häufig zurückgreift und ohne die gar nichts läuft: des Stacks. Was es damit auf sich hat, entnehmen Sie bitte dem Kapitel »Stack« auf Seite 385. Arbeiten Sie daher mit diesen Registern unter allen Umständen nur dann, wenn Sie genau wissen, was Sie tun!



Dennoch gibt es auch heute noch Spezialaufgaben für bestimmte Register, die andere Register nicht übernehmen können: So ist Kommunikation mit der Peripherie über Ports auch heute nur mit dem EDX- und EAX-Register möglich: EDX enthält die Adresse des Ports und EAX sendet oder empfängt das Datum. Auch können einige Befehle auf die Zusammenarbeit mit dem Akkumulator EAX hin optimiert sein und laufen mit diesem Register ggf. schneller ab als mit anderen Allzweckregistern. Und auch die letzten beiden der acht Allzweckregister, das *Extended Source Index register* ESI und das *Extended Destination Index register* EDI werden üblicherweise für bestimmte Zwecke reserviert: Sie spielen bei so genannten String-Befehlen eine entscheidende Rolle. Wir werden in diesem Kapitel noch darauf zu sprechen kommen.

Alias-Namen

Die 32-Bit-*Exx*-Register sind die physikalischen Strukturen, mit denen die Prozessoren aus der Pentium-4-Familie arbeiten. Wie bereits mehrfach geäußert, sind sie evolutionär aus den 16-Bit-Pendants der Vor-80386-Prozessoren entstanden. Nicht nur aufgrund der Abwärtskompatibilität zu diesen Prozessoren, sondern einfach auch aus praktischen Gründen gibt es jedoch die »alten« Registernamen weiterhin: Mit ihnen können eben auch Words oder Bytes in DoubleWord-Registern gezielt bearbeitet werden. Daher können Sie auch heute noch die »Register« AX, BX, CX, DX, SI, DI, BP und SP ansprechen! Allerdings stehen sie nur noch für die jeweils »unteren« sechzehn Bits 0 bis 15 der physikalisch vorhandenen Exx-Pendants, sind also nicht viel mehr als Alias-Namen bestimmter Teile des korrespondierenden 32-Bit-Registers. Analoges gilt für die 8-Bit-»Register« AH, AL, BH, BL, CH, CL sowie DH und DL, die jeweils die »oberen« (= *high*) 8 Bits der alten 16-Bit-Register repräsentieren, also die Bits 8 bis 15, oder die »unteren« (= *low*), also die Bits 0 bis 7.

Abbildung 1.1 versucht das darzustellen: Für eine Operation seien lediglich die Bits 0 bis 7 des EAX-Registers notwendig. Daher wird der Instruktion als Operand das »Register« AL (*accumulator – low byte*) übergeben. Die Operation erfolgt nun genau mit den Bits dieses »Registers«, den Bits 0 bis 7 des EAX-Registers. Alle anderen Bits bleiben »unsichtbar« und werden nicht verändert! Eine weitere Operation benötigt die Bits 8 bis 15 aus Register EBX. Der Instruktion wird daher als Operand das »Register« BH (*base register – high byte*) genannt. Auch in diesem Fall wirkt sich die Operation ausschließlich auf die Bits 8 bis 15 des EBX-Registers aus, alle anderen bleiben unverändert. Wird dagegen das niedrigerwertige Word im ECX-Register benötigt, spricht man es über CX an. Mit EDX schließlich wird dann das real existierende 32-Bit-Register EDX angesprochen.



Es gibt nur die Möglichkeit, das »untere« Word eines Registers oder die dieses Word bildenden Bytes gezielt anzusprechen! So gibt es *keine* Alias-Namen für das »obere« Word (Bits 16 bis 31) oder die dieses bildenden Bytes (Bit 16 bis 23 bzw. 24 bis 31). Auch lassen sich byteweise nur die vier Allzweckregister EAX, EBX, ECX und EDX, nicht aber alle anderen Register ansprechen. Immerhin gibt es mit »IP« bzw. »Flags« auch die Alias-Namen für das jeweils »untere« Word der Register EIP und EFlags. Analoges gilt für (E)SI, (E)DI, (E)BP und (E)SP (vgl. Abbildung 1.1).

Interpretation

Die Alias-Namen für bestimmte Registerteile der Allzweckregister erwecken den Eindruck, dass das Stichwort »Interpretation« unter Assembler eine bedeutende Rolle spielt: Das »Register« AH wird als Feld von acht bestimmten Bits des Registers EAX, den Bits 8 bis 15, *interpretiert*.

Dieser Eindruck stimmt! Ein Grund für die Flexibilität des Assembler besteht darin, dass er in Wirklichkeit nur wenige grundlegende Strukturen kennt und Annahmen macht. Den Gesamtzusammenhang im Auge zu behalten und sinnvolle Befehle auf sinnvolle Daten anzuwenden, ist Ihre Sache! Ganz besonders deutlich wird dieser Sachverhalt, wenn man einmal ein Allzweckregister genauer betrachtet, nehmen wir z.B. EAX. Wie jeder weiß, arbeitet der Prozessor ja binär, was bedeutet, dass er nur die Zustände »0« und »1« kennt. Er arbeitet also bitorientiert. Wen wundert daher, dass die Allzweckregister diesem Sachverhalt Rechnung tragen und 32 Bits realisieren, wie Abbildung 1.2 es zeigt? (Wer sich bei der binären Darstellung eines Datums noch ein wenig schwer tut, sei auf Kapitel »Datenformate« ab Seite 778 verwiesen).

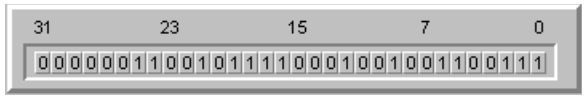


Abbildung 1.2: Binäre Darstellung eines DoubleWords mit dem dezimalen Wert 53.416.551

integer or not integer!

Doch was für ein Datum enthält EAX nun tatsächlich: Sind es 32 einzelne, von einander unabhängige Bits, die zwar gemeinsam in einer 32-Bit-Struktur gespeichert werden, die frappierend einem DoubleWord gleicht, die aber sonst wenig mit einander zu tun haben? Oder müssen diese 32 Bits im Zusammenhang gesehen werden, weil sie eine Zahl darstellen? In diesem Falle beinhaltete EAX die Ganzzahl 53.416.551.

signed or not signed!

Nächstes Problem: Ist das Datum tatsächlich eine Integer und kein Bit-Feld, erhebt sich die nächste Frage: Ist sie vorzeichenbehaftet oder nicht? Mit anderen Worten: Stellt Bit 31 das Vorzeichenbit einer Integer dar oder ist es deren höchste signifikante Stelle? Das ist, wenn man die Befehlsverarbeitung betrachtet, kein unwesentlicher Unterschied! Denn wäre in der Abbildung Bit 31 gesetzt, hätte die Zahl je nachdem, ob es ein Vorzeichen ist oder nicht, den Wert 2.200.900.199 (vorzeichenlos) bzw. -2.094.067.097 (mit Vorzeichen).

32, 16 oder 8 Bits?

Und noch ein Dilemma: Könnte es nicht sein, dass nur Teile des Registers eine Rolle spielen, da der vorangegangene Befehl einen der Alias-

Namen von oben verwendet hat? So könnte, wie Abbildung 1.3 das darstellt, der Wert »4711« (als Word) durch den vorangegangenen Befehl in das »Register« AX geschrieben worden sein und hätte damit ein anderes Datum überschrieben. Das bedeutet dann aber, dass die Bits 16 bis 31 des Registers EAX spätestens seit dem letzten Befehl Müll enthalten, der tunlichst künftig unberücksichtigt bleibt.

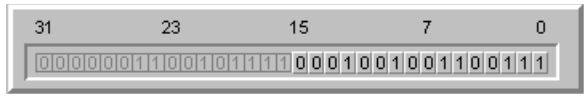


Abbildung 1.3: Binäre Darstellung eines Words mit dem dezimalen Wert 4711

Oder doch nicht? Haben diese Bits 16 bis 31 vielleicht trotz des »Überschreibens« eine Berechtigung? Denn immerhin könnten sie ja im Rahmen einer Adressberechnung durch eine Multiplikation eines Words (in den Bits 0 bis 15) mit der Konstanten 65.536 und damit ein DoubleWord als Resultat entstanden sein, zu der nun durch einfaches Überschreiben der vormals dort stehenden Nullen ein Offset addiert wird. Diese Art nicht nur der Adressenberechnung ist tatsächlich möglich, wir werden dies bei den entsprechenden Befehlen noch sehen!

nibble or not nibble!

Und schließlich: Betrachten wir einmal nur das niedrigstwertige Byte des Registers EAX, das über AL angesprochen werden kann. Zeigt der obere Teil in Abbildung 1.4 nun die Darstellung eines Bytes mit dem Wert 103, oder repräsentiert es eine *binary coded decimal*, eine BCD, mit dem Wert 7, wie es der untere Teil der Abbildung 1.4 nahe legt? (Falls Ihnen BCDs nicht geläufig sind, verweise ich auf den Abschnitt »Binary Coded Decimals« auf Seite 809.) Dann enthielten Bits 4 bis 7 wieder Müll!

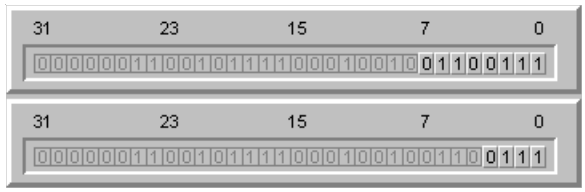


Abbildung 1.4: Binäre Darstellung eines Bytes mit dem dezimalen Wert 103 und einer BCD mit dem dezimalen Wert 7

Oder doch nicht – gibt es doch auch gepackte BDCs! Dann allerdings enthielte EAX die BCD 67 (vgl. oberer Teil der Abbildung). Wie und wodurch aber sollte die BCD 67 vom Byte 103 unterschieden werden?



Sie sehen, dass der Prozessor hier hoffnungslos überfordert wäre, müsste er diese Entscheidungen treffen. Denn mit welchen Daten Sie arbeiten – vorzeichenbehaftet oder vorzeichenlos, Ganzzahlen oder Bit-Felder, Binärdaten oder BCDs –, das weiß nur einer: Sie. Da Sie dem Prozessor diese Information aber nicht oder nur sehr eingeschränkt geben können, liegt es in Ihrer Verantwortung allein, die Ergebnisse von Berechnungen oder sonstigen Operationen korrekt zu *interpretieren*! Der Prozessor kann Ihnen hierbei nur helfen, indem er Ihnen signalisiert, was wäre, wenn eingegebenes Datum dieses oder jenes wäre. Und das tut er auch, wie wir bei der Besprechung der Flags gleich noch sehen werden. Die Entscheidungen treffen, was nun zu erfolgen hat, müssen jedoch Sie! Und dies unterscheidet Programmierung mit Assembler von Programmierung mit Hochsprachen. Denn in letzterer kann der Compiler meckern, wenn Sie versuchen, einem Byte eine Fließkommazahl zuzuordnen oder eine Routine mit einem Array als Parameter aufzurufen, die eine LongInt erwartet. Der Assembler kann das weniger stringent und lange nicht in dem Ausmaß, weil er, wie gesehen, z.B. nicht wissen kann, was in den Prozessorregistern für Daten hausen. *Assembler-Programmierung hat viel mit korrekter Interpretation dessen zu tun, was man sieht!*

Segmentregister

Die sechs Segmentregister enthalten Adressen, die beim Zugriff auf den Speicher eine wesentliche Rolle spielen. Sie sind auch nur für diesen Zweck nutzbar. Daher werden wir sie auch erst im Kapitel »Speicherverwaltung« ab Seite 394, wo es um die Speichersegmentierung geht, näher anschauen.



Das Segmentregister DS besitzt unter den Segmentregistern eine Sonderrolle, dient es doch bei Adressberechnungen zum Zugriff auf Daten als Standard-Bezugsregister. Die Nutzung der Register ES, FS oder GS zu diesem Zweck ist zwar möglich, verlangt aber einen so genannten *segment override prefix*, der ein zusätzliches Byte in der Instruktion darstellt und die Befehlsverarbeitung in den Pipelines entsprechend verzögert. Auch eine Sonderstellung nehmen die Segmentregister CS und SS ein, die für das Codesegment und den Stack reserviert sind.

Befehlszeiger

Der *instruction pointer* EIP bzw. sein 16-Bit-Alias IP werden lediglich der Vollständigkeit halber erwähnt. Ihnen als Programmierer ist ein Zugriff auf dieses Register vollständig verwehrt. Das Register untersteht ausschließlich der Kontrolle des Prozessors: Hier speichert er die Adresse

des nächsten auszuführenden Befehls im Programm. Der Inhalt des Registers wird vom Prozessor bei jeder Ausführung eines Befehls aktualisiert. So wird der Zeiger während der Befehlsdekodierung um die Anzahl Bytes erhöht, die der augenblicklich dekodierte Befehl zur Codierung benötigt. Oder es wird in ihn das Ziel eines Sprunges oder eines Unterprogrammaufrufs eingetragen.

Direkter Zugriff auf EIP (IP) ist auch nicht notwendig. Denn ein Schreiben in das EIP hätte ja zur Folge, dass der Prozessor an der eben eingeschriebenen Adresse mit der Programmausführung fortfahren soll. Das aber können Sie einfacher und komfortabler über die Sprung- oder Unterprogrammaufrufbefehle (JMP, Jcc, CALL) erreichen, die Ihnen die lästige und nicht einfache Adressberechnung abnehmen. Und sollten Sie wirklich einmal genötigt sein, die Position des nächsten Befehls im Programm zu erfahren – nichts anderes würden Sie ja durch das Auslesen von EIP erreichen –, so gibt es hierfür andere Möglichkeiten, z.B. über die Abfrage der aktuellen Position mittels eines vordefinierten Symbols des Assemblers.

EFlags-Register

Bleibt noch das EFlags-Register zu erklären. Dieses Register, entstanden aus dem 16-Bit-Flag-Register, ist (direkt) nur schwer zugänglich: Es gibt nur sehr wenige Befehle, die das EFlags-Register als Quelle oder Ziel einer Operation akzeptieren. Das Datum in EFlags wird in eindeutiger Weise interpretiert: als Feld von 32 Bits, wie Abbildung 1.5 zeigt:

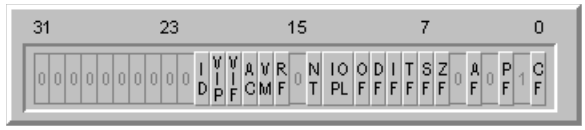


Abbildung 1.5: Speicherabbild des EFlag-Registers

Diese Bits sind vollständig unabhängig voneinander und beeinflussen sich gegenseitig nicht. Sie dienen drei Zwecken:

- Darstellung des derzeitigen Programmstatus (Condition Code),
- Steuerung gewisser Programmabläufe (Kontrollflags) und
- Darstellung bestimmter Systemparameter (Systemflags), die einen Einfluss auf die Funktion des Prozessors und das Betriebssystem haben.

Da diese Bits bestimmte Sachverhalte (entweder dem Programmierer oder dem Prozessor) signalisieren sollen, nennt man sie (der Schifffahrt entliehen) auch (Signal-)Flaggen oder »Flags«. Gemäß der drei genannten Aufgaben teilt man sie in Condition Code, Kontrollflags und Systemflags ein.

Wie Sie sehen können, sind nicht alle Flags definiert oder besser: dem Programmierer zugänglich. Die den grau dargestellten Bits 1, 3, 5, 15 und 22 bis 31 zugeordneten Flags gelten als reserviert und sollten tunlichst nicht angetastet werden. Das bedeutet, sie sollten nicht mit anderen als den jeweils aktuellen Werten belegt werden, wollen Sie unschöne Exceptions der Form »Allgemeiner Zugriffsfehler« vermeiden. Die oben gezeigten Nullen und Einsen sind die Standardwerte beim Pentium 4, andere Prozessoren können hier andere Werte haben.



Falls Sie also einmal Änderungen am Inhalt des EFlags-Register vornehmen müssen, die Sie nicht anders realisieren können – wir werden darauf zurückkommen –, so sollten Sie es zunächst auslesen, die Änderungen vornehmen und den geänderten Inhalt wieder zurückschreiben. Auf diese Weise stellen Sie sicher, dass die nicht zu verändernden Flags Prozessor-unabhängig den korrekten Standardwert enthalten.

Die wichtigsten und am häufigsten benutzten Flags sind die Statusflags (Abbildung 1.6, oben). Sie werden durch viele Instruktionen verändert oder dienen einigen Instruktionen als Input und signalisieren den Prozessorzustand nach einer arithmetischen Operation. Schon erheblich weniger häufig verwendet wird das einzige Kontrollflag (Abbildung 1.6, Mitte). Es hat lediglich bei den Stringbefehlen Wirkung und wird daher zusammen mit diesen besprochen. Mit den Systemflags (Abbildung 1.6, unten) werden Sie vermutlich selten in Berührung kommen. Sie spielen eine wesentliche Rolle bei der Verwaltung von sog. »Tasks« (NT, IOPL), in bestimmten Betriebsmodi des Prozessors (»virtual 8086 mode«; VIP, VIF, VM) sowie in speziellen Programmen (z.B. Debugger; TF, RF) oder zur Steuerung bestimmter Systemdienste (IF, AC) – Summa: Sie sind Sache des Betriebssystems oder sonstiger Spezialprogramme, die uns im Rahmen dieses Buches nicht interessieren. Daher werden wir sie an anderer Stelle besprechen.

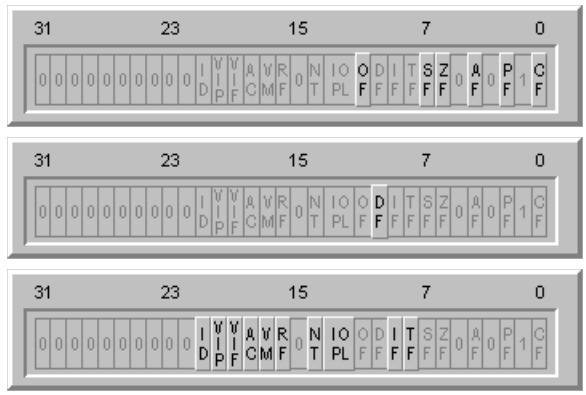


Abbildung 1.6: Status-, Kontroll- und Systemflags der CPU

ID-Flag Lediglich Bit 21, das System-Flag ID oder »*identification flag*«, könnte Sie interessieren: So können Sie anhand des Zustandes dieses Flags feststellen, ob der Prozessor über den äußerst wichtigen CPUID-Befehl verfügt. Interessant wird das aber nur bei Prozessoren vor dem Pentium (ja, die gibt's noch: mein alter 486 dient mir noch als Druckerserver!), da seither jedem Prozessor der Befehl CPUID implementiert wurde und künftig wohl auch wird.

Statusflags Die Statusflags sind genau die Hilfe, die Ihnen der Prozessor bei der Interpretation von Registerinhalten zur Verfügung stellt. Sie werden gebildet vom

- »carry flag« (CF). Dieses Flag ist ein sehr häufig und zu den verschiedensten Zwecken benutztes Flag. Seine eigentliche und Hauptaufgabe ist allerdings, einen Über- oder Unterlauf nach arithmetischen Operationen mit vorzeichenlosen Integers anzuzeigen. Es wird daher während einer Operation gesetzt, wenn z.B. die Addition zweier Daten den Wertebereich der verwendeten Daten überschreiten würde (z.B. bei Words: Überlauf von Bit 15 in das bei Words nicht vorhandene Bit 16) oder eine Subtraktion zweier Daten das untere Limit »0« unterschreiten würde (Unterlauf mit Borgen aus dem z.B. bei DoubleWords nicht vorhandenen Bit 32). Das carry flag nimmt sozusagen die Position des jeweils »fehlenden« Bits ein: Bit 32 bei DoubleWords, Bit 16 bei Words und Bit 8 bei Bytes.
- »parity flag« (PF). Dieses Flag wird immer dann gesetzt, wenn das niedrigstwertige Byte des Datums eine gerade Anzahl von gesetzten Bits hat, sonst wird es gelöscht. Bedeutung hat dieses Flag im Zusammenhang mit der Kommunikation über serielle Schnittstellen,

da ja Übertragungsprotokolle ebenfalls solche *parity bits* senden (können) und auf diese Weise recht schnell festgestellt werden kann, ob das empfangene Byte korrekt empfangen wurde (PF und gesendetes *parity bit* stimmen überein) oder nicht.

- »adjust flag«, auch »auxiliary carry flag« oder kurz »auxiliary flag« (AF). Dieses Flag kommt bei der BCD-Arithmetik zum Einsatz, da es wie das carry flag einen Über- oder Unterlauf anzeigt. Da BCDs einzelne Nibble (oder »half bytes«) und damit kleiner als die kleinste definierte Einheit (Byte) sind, kann das carry flag hier nicht die »Retterrolle« spielen; dies erfolgt durch das adjust flag: Es ist das bei BCDs nicht vorhandene »Bit 4«, in das oder aus dem ein Über-/Unterlauf erfolgt.
- »zero flag« (ZF). Es wird immer dann gesetzt, wenn das Ergebnis der Operation null ist, also kein Bit gesetzt ist. Andernfalls ist es gelöscht.
- »sign flag« (SF). Dieses Flag enthält, wie der Name schon vermuten lässt, fast immer das Vorzeichen des Ergebnisses einer Operation (Ausnahme im übernächsten Absatz!). Je nach eingesetztem Datum (ShortInt, SmallInt, LongInt) ist es eine Kopie des Bits 7, 15 oder 31 des Ergebnisses, das das Vorzeichen repräsentiert. Ist das sign flag gesetzt, signalisiert es ein negatives Vorzeichen, andernfalls ist das Datum positiv.
- »overflow flag« (OF). Dieses Flag ist das CF-Pendant für vorzeichenbehaftete Zahlen. Sobald das Ergebnis einer Operation nicht mehr im verwendeten Format (ShortInt, SmallInt oder LongInt) darstellbar ist, wird OF gesetzt, andernfalls gelöscht.



Achtung Falle! Das overflow flag signalisiert einen Übertrag in das/aus dem MSB, dem *most significant bit*. Bei LongInts handelt es sich hierbei wie bei DoubleWords um das Bit 31, bei SmallInts/Words um Bit 15 und bei ShortInts/Bytes um Bit 7. Während jedoch bei vorzeichenlosen Zahlen dieses MSB Teil der zur Zahlendarstellung verfügbaren Bits ist, repräsentiert es bei vorzeichenbehafteten Zahlen das Vorzeichen und besitzt somit im sign flag eine Kopie. Und dies führt zu Interpretationsproblemen, wenn der Wertebereich einer vorzeichenbehafteten Zahl über- oder unterschritten wird. Zur Illustration diene Abbildung 1.7:

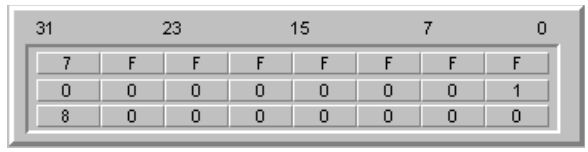


Abbildung 1.7: Darstellung eines Überlaufs nach Addition zweier vorzeichenbehafteter Zahlen

In der oberen Zeile ist (am Beispiel eines 32-Bit-DoubleWords) die größte positive, vorzeichenbehaftete Zahl dargestellt: $7FFF_FFFF. Addiert man zu dieser Zahl eine »1«, sieht man das Dilemma: Da $7FFF_FFFF, als vorzeichen*lose* Zahl interpretiert, noch nicht der Weisheit letzter Schluss ist, addiert der Prozessor dies brav zu $8000_0000. Denn schließlich ist das ja, vorzeichenlos interpretiert, auch korrekt. Damit ist aber Bit 31 und im Gefolge auch das sign flag gesetzt, was, vorzeichenbehaftet interpretiert, ein negatives Vorzeichen bedeutet. Damit steht im Register nun die kleinste negativ darstellbare Integer (*cave*: 2er-Komplement!). Das bedeutet: Was vorzeichenlos interpretiert absolut korrekt ist, ist vorzeichenbehaftet interpretiert falsch – die Überschreitung des positiven Wertebereichs führt zu einer negativen Zahl. Da der Prozessor nun nicht wissen kann, ob $7FFF_FFFF nun +2.147.483.647 (vorzeichenbehaftet) oder 2.147.483.647 (vorzeichenlos) ist, führt er die Addition so aus, als würden vorzeichenlose Zahlen verwendet. Um aber zu signalisieren, dass im Falle vorzeichenbehafteter Zahlen ein Überlauf stattgefunden hat (Übertrag von Bit 30 in das Vorzeichen-Bit 31!), setzt er OF. Das bedeutet: Ist OF gesetzt und gleichzeitig auch SF, so wurde, vorzeichenbehaftet interpretiert, durch die Operation der positive Wertebereich überschritten und SF zeigt das falsche, »entgegengesetzte«, hier also negative Vorzeichen an. Ist OF dagegen gelöscht, gibt SF das korrekte, hier positive Vorzeichen an.

Die gleiche Überlegung rückwärts zeigt auch den Sachverhalt an, wenn der negative Wertebereich unterschritten wird. Auch hier kann Abbildung 1.7 als Illustration herhalten: In der untersten Zeile steht die kleinste negative Zahl. Subtrahiert man von ihr »1«, so stellt sich aufgrund der für vorzeichenlose Zahlen korrekt durchgeführten Operation das in der obersten Zeile dargestellte Ergebnis ein. Dies ist analog der eben durchgeführten Betrachtung aber die größte positive Zahl. Somit spiegelt auch hier die Stellung des sign flag einen falschen Sachverhalt wider: Nach Subtraktion einer Zahl von der kleinsten negativen Zahl wird das Vorzeichenbit gelöscht, was »positiv« heißen würde. OF ist auch in diesem Fall gesetzt, da ein Borgen aus Bit 31 in Bit 30 notwen-

dig wurde. Das aber bedeutet: Ist OF gesetzt und SF gelöscht, so wurde durch die Operation der negative Wertebereich unterschritten und SF zeigt das falsche, »entgegengesetzte« Vorzeichen. Ist OF dagegen gelöscht, so gibt SF wiederum das Vorzeichen korrekt an.

Anhand der Definition der Flags können Sie schon erkennen, dass ihre Funktion untrennbar mit den verschiedenen einsetzbaren Daten verknüpft ist: Das carry flag unterstützt die Interpretation vorzeichenloser Zahlen, sign und overflow flag die der vorzeichenbehafteten und adjust flag die der BCDs. Das zero flag kann für alle Zahlenarten verwendet werden, während das parity flag in diesem Zusammenhang keine Funktion hat.



Da diese Entscheidungshilfen von so großer Bedeutung sind, wurden Mnemonics (zur Definition des Begriffs siehe Kapitel »Mnemonics, Befehlssequenzen, Opcodes und Microcode« auf Seite 768) geschaffen, die Teil von bestimmten Befehls-Mnemonics sind und jeweils für eine ganz bestimmte Kombination von Statusflags gelten. Tabelle 1.1 zeigt die 30 Mnemonics, die aufgrund bestimmter Redundanzen und Beziehungen untereinander durch nur 16 unterschiedliche Prüfungen realisiert werden.

	Mnemonics			
Bedingung	**Negierung**		**Identität zu**	**Prüfung**
vorzeichenlos:				
A above	NBE	not below or equal		CF = 0 und ZF = 0
AE above or equal	NB	not below	NC no carry	CF = 0
B below	NAE	not above or equal	C carry	CF = 1
BE below or equal	NA	not above		CF = 1 oder ZF = 1
vorzeichenneutral:				
E equal			Z zero	ZF = 1
NE not equal			NZ not zero	ZF = 0
vorzeichenbehaftet:	NLE	not less or equal		OF = SF und ZF = 0
G greater	NL	not less		OF = SF
GE greater or equal	NGE	not greater or equal		OF ≠ SF
L less	NG	not greater		OF ≠ SF oder ZF = 1
LE less or equal				
allgemein:			PO parity odd	
NO no overflow				OF = 0
NP no parity				PF = 0
NS no sign				SF = 0
O overflow				OF = 1
P parity			PE parity even	PF = 1
S sign				SF = 1

Tabelle 1.1: Mnemonics für die Kombination bestimmter Statusflags nach vergleichenden Befehlen

So ist, wie eben gesehen, eine vorzeichenbehaftete Zahl dann größer als eine andere, wenn nach Bildung der Differenz das zero flag gelöscht und sign und overflow flag gleich sind. Diese Bedingung und die dahinter stehende Prüfung besitzt das Mnemonic »G« (»*greater*«), das Teil von so genannten bedingten Befehlen ist (z.B. J*G*, jump *if greater*). Diese Befehle werden wir weiter unten kennen lernen.



Die Statusflags sind, bis auf eine Ausnahme, nicht direkt veränderbar. Das heißt: Es gibt keine Befehle, die sie direkt einzeln und gezielt verändern! Wie gesagt: Bis auf eine Ausnahme, und die betrifft das carry flag. Seiner Bedeutung nicht nur als Statusflag nach arithmetischen Operationen entsprechend können Sie es mit bestimmten Befehlen setzen, löschen oder »umdrehen«. Die hierzu notwendigen Befehle werden wir im Kapitel »Instruktionen zur gezielten Veränderung des Flagregisters« auf Seite 127 besprechen.



Es gibt aber sehr wohl eine Möglichkeit, die Statusflags indirekt gezielt zu verändern. Dazu muss aber das gesamte EFlags-Register ausgelesen und in ein Allzweckregister transportiert werden. Hier lässt/lassen sich dann das/die zu verändernden Flag(s) mit logischen oder Bit-orientierten Instruktionen verändern und wieder in das EFlags-Register zurücktransferieren. Diese Methode werden wir in Teil 2 des Buches kennen lernen.

CPU-Befehle

Soweit im Folgenden nicht ausdrücklich anders vermerkt, lassen sich alle CPU-Befehle mit DoubleWords (32 Bits), Words (16 Bits) oder Bytes (8 Bits) bzw. ihren vorzeichenbehafteten Pendants (LongInt, SmallInt, ShortInt) durchführen. Die Unterscheidung erfolgt ausschließlich durch die Angabe des entsprechenden »Register«-Namen (z.B. ADD AL, 3 - Byte; SUB BH, BL - Byte; INC CX, 1 - Word; DEC EDX, EAX - DoubleWord), soweit (mindestens) ein Register involviert ist. Ist dagegen kein Register beteiligt, kommt eine Speicherstelle zum Einsatz. Diese muss daher vorab in geeigneter Weise definiert worden sein, damit der Assembler weiß, mit welchen Datenbreiten er arbeiten muss. Wir werden darauf in Teil 2 des Buches zurückkommen.



Da in den folgenden Betrachtungen mit verschiedenen Beispielen gearbeitet wird, empfiehlt es sich, zunächst zum besseren Verständnis das Kapitel »Datenformate« ab Seite 778 zu konsultieren, in dem wichtige Aspekte der prozessorinternen Darstellung verschiedener Daten be-

schrieben werden, von deren Kenntnis im Folgenden Gebrauch gemacht wird.

Der CPU-Befehlssatz umfasst Befehle zu

- arithmetischem Manipulieren von Daten
- logischem Manipulieren von Daten
- Datenvergleich
- bitorientierten Operationen
- Datenaustausch
- Datenkonversion
- Sprungbefehlen
- Flagmanipulationen
- Stringoperationen
- Verwaltungsoperationen
- speziellen Operationen



Die meisten der folgenden CPU-Befehle haben Operanden, also Parameter, die ihnen übergeben werden. Diese Operanden müssen in einer speziellen Art und Weise angegeben werden, um korrekt zu arbeiten. Sollten Sie mit der Angabe dieser Operanden (Befehlssemantik) nicht vertraut sein, konsultieren Sie bitte das Kapitel »Befehlssemantik« auf Seite 763, bevor Sie weiterlesen.

1.1.1 Arithmetische Operationen

Die CPU ist Integer-orientiert. Das bedeutet, dass sie nur mit Ganzzahlen arbeiten kann. Es verwundert daher nicht sonderlich, dass sich die Arithmetik der CPU auf die Grundrechenarten und wenig mehr beschränkt, dafür aber mit einigen Variationen, die unterschiedliche Bedingungen berücksichtigen.

ADD
SUB

Beginnen wir mit den grundlegendsten Berechnungen. Natürlich kann die CPU Integers addieren (ADD) und subtrahieren (SUB). Das Ergebnis der Operation kann – abgesehen vom Wert, zu dessen Berechnung wohl nicht viel zu sagen ist – mit Hilfe der Statusflags gemäß der eingesetzten Daten interpretiert werden.

Statusflags

So signalisiert das zero flag, wenn gesetzt, dass das Ergebnis »Null« ist, unabhängig davon, ob die betrachteten Zahlen vorzeichenlos oder vorzeichenbehaftet sind. (Es gibt also hier keine »negative Null« wie bei Fließkommazahlen, wie wir noch sehen werden!) Bei vorzeichenlosen Zahlen signalisiert das carry flag darüber hinaus, ob ein Über- oder Unterlauf stattgefunden hat, der gültige Wertebereich somit über- oder unterschritten wurde. Ob es ein Über- oder Unterlauf war, entscheidet die Operation: Eine Unterschreitung des Wertebereichs mit ADD ist bei vorzeichenlosen (und somit immer positiven) Zahlen genauso wenig möglich wie eine Überschreitung mittels SUB.

Dies kann nur bei vorzeichenbehafteten Zahlen erfolgen. Hier übernimmt daher das overflow flag die Funktion des carry flag. Ist OF gelöscht, hat durch die Operation kein Überlauf in das oder Borgen aus dem Vorzeichenbit (sign flag oder MSB, most significant bit; Bit 31 bei LongInts, Bit 15 bei SmallInts, Bit 7 bei ShortInts) stattgefunden. Im gesetzten Zustand wurde das sign flag aufgrund des Über- bzw. Unterlaufs verändert. Wie das overflow flag in Verbindung mit dem sign flag zu interpretieren ist, wurde bereits weiter oben geschildert (Seite 41 ff.).

Handelte es sich dagegen weder um vorzeichenlose noch um vorzeichenbehaftete Zahlen, sondern um BCDs, hat das adjust flag seinen Auftritt. Es zeigt analog zum carry flag an, dass ein Überlauf bei der Addition zweier ungepackter BCDs stattgefunden hat, hier allerdings von Bit 3 in Bit 4, da BCDs ja 4-Bit-Integers sind. Bitte beachten Sie, dass nach Addition zweier ungepackter BCDs der Korrekturbefehl AAA und nach Subtraktion zweier BCDs der Korrekturbefehl AAS aufgerufen werden muss, um ein korrektes Ergebnis zu erhalten.



Auch das carry flag kann bei BCDs eine Rolle spielen. Neben den ungepackten BCDs, die mit AAA und AAS korrigiert werden können, können auch gepackte BCDs addiert und subtrahiert werden. In diesem Fall fungiert CF als AF des zweiten Nibbles, also der zweiten BCD. Nach Addition/Subtraktion von gepackten BCDs muss die Korrektur DAA bzw. DAS aufgerufen werden. Einzelheiten zu diesen Korrekturbefehlen finden Sie weiter unten.

Bleibt noch das parity flag. Es signalisiert wiederum die Parität des niedrigstwertigen Byte des Ergebnisses, also seiner Bits 7 bis 0: Liegt eine gerade Anzahl von gesetzten Bits vor (»gerade Parität«), so ist PF gesetzt, andernfalls gelöscht.

ADD und SUB erlauben verschiedene Arten von Operanden (XXX dient im Folgenden als Platzhalter für ADD bzw. SUB): *Operanden*

- Addition/Subtraktion einer Konstanten zum/vom Akkumulatorinhalt
  `XXX AL, Const8; XXX AX, Const16; XXX EAX, Const32`
- Addition/Subtraktion einer Konstanten zu/von einem Registerinhalt
  `XXX Reg8, Const8; XXX Reg16, Const16; XXX Reg32, Const32`
- Addition/Subtraktion einer Konstanten zu/von einem Speicheroperand
  `XXX Mem8, Const8; XXX Mem16, Const16; XXX Mem32, Const32`
- Addition/Subtraktion einer Byte-Konstanten zu/von einem Registerinhalt mit Vorzeichenerweiterung
  `XXX Reg16, Const8; XXX Reg32, Const8`
- Addition/Subtraktion einer Byte-Konstanten zu/von einem Speicheroperand mit Vorzeichenerweiterung
  `XXX Mem16, Const8; XXX Mem32, Const8`
- Addition/Subtraktion eines Registerinhaltes zu/von einem Registerinhalt
  `XXX Reg8, Reg8; XXX Reg16, Reg16; XXX Reg32, Reg32`
- Addition/Subtraktion eines Speicheroperanden zu/von einem Registerinhalt
  `XXX, Reg8, Mem8; XXX Reg16, Mem16; XXX, Reg32, Mem32`
- Addition/Subtraktion eines Registerinhalts zu/von einem Speicheroperanden
  `XXX, Mem8, Reg8; XXX Mem16, Reg16; XXX Mem32, Reg32`

Sie sehen, die grundlegenden arithmetischen Befehle sind so grundlegend, dass mit ihnen praktisch jede Datenquelle (Konstante, Register, Speicher) und praktisch jedes Ziel (Register, Speicher) verwendet werden kann.



Beachten Sie bitte, dass der Akkumulator (also das EAX-Register bzw. seine AX- bzw. AL-Form) auch bei den modernen Prozessoren mit gleichberechtigten Allzweckregistern immer noch in der Form eine Sonderrolle spielt, dass es sich bei der Addition/Subtraktion von Konstanten zu/vom Akkumulator um Ein-Byte-Befehle handelt, während alle anderen Versionen mindestens zwei Bytes umfassen.

MUL
IMUL

Ganz so einfach wie bei Addition und Subtraktion ist die Sache bei der Multiplikation zweier Zahlen nicht. Das fängt mit der Feststellung an, ob ein Vorzeichen existiert oder nicht. Wie jeder mit Papier und Bleistift nachvollziehen kann, ist die Frage, ob das höchstwertige Bit des Datums in die Berechnung einbezogen werden muss (kein Vorzeichenbit) oder nicht (Vorzeichenbit), also ob die Zahlen durch 31 oder 30 Bits (DoubleWords/LongInts) dargestellt werden, ganz entscheidend für das Ergebnis. (Analoges gilt natürlich für Words/SmallInts und Bytes/ShortInts.) Man kann also in diesem Fall nicht einfach anhand von Flagstellungen und *nachträglicher* Interpretation des Wertes zu einem korrekten Ergebnis kommen: Die durchgeführten Operationen sind unterschiedlich! Daher existieren für die Multiplikation jeweils zwei Befehle, die entweder vorzeichenlose Ganzzahlen verarbeiten (MUL) oder vorzeichenbehaftete Integers im engeren Sinne (*integer multiplication* IMUL).

Statusflags

Da für jeden Fall ein eigenständiger Befehl existiert, spielen die Flagstellungen bei MUL/IMUL eine untergeordnete Rolle, wenn überhaupt. Nach MUL und IMUL haben nur CF und OF eine Bedeutung. Sie zeigen an, ob das Ergebnis der Multiplikation den Wertebereich der Operanden überschritten hat oder nicht.

Was heißt das? Bei MUL ist das einfach zu erklären. Wenn beispielsweise zwei Words mit einander multipliziert werden, so kann das Ergebnis Werte im Bereich eines DoubleWords annehmen (z.B. \$1000 · \$00FF = \$000F_F000). Muss aber nicht: Es kann auch im Wertebereich eines Words bleiben (z.B. \$0100 · \$00FF = \$0000_FF00). Und genau dieser Sachverhalt wird durch CF und OF signalisiert: Wird durch die Multiplikation der Wertebereich der Operanden (hier Words) überschritten, so werden CF und OF gesetzt. In diesem Fall ist das höherwertige Word des Ergebnis-DoubleWords nicht »0«. Bleibt dagegen das Ergebnis im Wertebereich Word, so ist das höherwertige Word des Ergebnisses »0« und CF und OF werden gelöscht.

Bei IMUL ist das zwar grundsätzlich gleich. Doch nachdem IMUL mit vorzeichenbehafteten Integers arbeitet, kann das Ergebnis auch negativ sein. In diesem Falle ist der höherwertige Anteil der resultierenden LongInt von Null verschieden, selbst wenn das Ergebnis vom absoluten Betrag her in ein Word passen würde. (Stichwort: »sign extension«. Im höherwertigen Teil steht dann der Wert \$FFFF, der aus der Vorzeichenerweiterung einer SmallInt in eine LongInt resultiert.) Daher wird bei IMUL dann CF und OF gelöscht, wenn der höherwertige Anteil des Ergebnisses entweder »0« ist (positive Zahl) oder »\$FFFF« (negative

Zahl). Andernfalls sind beide Flags gesetzt. (Es versteht sich, glaube ich, von selbst, dass der hier an der Multiplikation von SmallInts dargestellte Sachverhalt analog mit den anderen Daten – ShortInts und LongInts – funktioniert!)

Operanden

Als Operatoren für die Befehle kommen lange nicht so viele Möglichkeiten wie bei der Addition/Subtraktion in Betracht. Hinzu kommt, dass die Befehle den ersten Operanden, der Ziel- und ersten Quelloperanden angibt, schlichtweg implizieren. Insofern gibt es nur zwei Möglichkeiten (XXX steht für MUL/IMUL):

- Expliziter (zweiter) Quelloperand ist ein Register
  `XXX Reg8; XXX Reg16; XXX Reg32`
- Expliziter (zweiter) Quelloperand ist eine Speicherstelle
  `XXX Mem8; XXX Mem16; XXX Mem32`

In allen Fällen ist der erste Quelloperand (= Multiplikand) und damit auch das Ziel (= Produkt) vorgegeben: der Akkumulator. (Wieder eine »Verletzung« des Gleichheitsprinzips für Allzweckregister!) Je nach Größe des explizit angegebenen Operanden (Quelloperand 2 = Multiplikator!) ist damit die implizierte Quelle (Quelloperand 1) und das ebenfalls implizierte Ziel vorgegeben, wie Tabelle 1.2 zeigt:

expliziter Operand (Source-Operand #2)	durch expliziten Operanden festgelegte Datengröße	impliziter Operand (Source-Operand #1)	impliziter Zieloperand
Reg8 / Mem8	Byte	AL	AX
Reg16 / Mem16	Word	AX	DX:AX
Reg32 / Mem32	DoubleWord	EAX	EDX:EAX

Tabelle 1.2: Explizite und implizite Operanden des MUL-/IMUL-Befehls



Beachten Sie bitte, dass bei der Verwendung von Words als Operanden das resultierende DoubleWord auch bei 32-Bit-Prozessoren *nicht* in EAX abgelegt wird, sondern in das höherwertige Word in DX und das niedrigerwertige Word in AX aufgeteilt wird: *DX := HiWord(AX * Mem16/Reg16), AX := LoWord(AX * Mem16/Reg16)*. Dies ist in der Abwärtskompatibilität zu den 16-Bit-Prozessoren begründet. Leider gibt es keine MUL-Version, die ein DoubleWord-Ergebnis in EAX ablegt.



Bei IMUL dagegen sieht das (scheinbar) ein wenig erfreulicher aus. Der IMUL-Befehl existiert in drei Formen: der Ein-Operanden-Form, der Zwei-Operanden-Form und sogar in einer Drei-Operanden-Form. Durch die Erweiterungen können auch erster Quell- und Zieloperand

explizit vorgegeben werden. Doch erkauft man sich dies mit einem Nachteil: Das Multiplikationsergebnis kann eventuell nicht korrekt sein, wie wir gleich sehen werden.

In der Ein-Operanden-Form verhält sich der IMUL-Befehl analog zu MUL, mit der Ausnahme, dass er vorzeichenbehaftete Integers verwendet. Es gilt also auch hier die Tabelle und die Aufteilung eines Ergebnis-DoubleWords in zwei Word-Register selbst bei 32-Bit-Prozessoren.

In der Zwei-Operanden-Form sind folgende Operanden erlaubt:

- Multiplikation eines Registerinhaltes mit einer vorzeichenerweiterten Konstante
  `IMUL Reg16, Const8; IMUL Reg32, Const8`
- Multiplikation eines Registerinhaltes mit einer Konstanten
  `IMUL Reg16, Const16; IMUL Reg32, Const32`
- Multiplikation eines Registerinhaltes mit einem Registerinhalt
  `IMUL Reg16, Reg16; IMUL Reg32, Reg32`
- Multiplikation eines Registerinhaltes mit einem Speicheroperanden
  `IMUL Reg16, Mem16; IMUL Reg32, Mem32`

Bitte beachten Sie, dass bei den Zwei-Operanden-Sequenzen das Ziel (und damit auch die erste Quelle) immer ein Register sein muss. Dessen Inhalt kann entweder (unter Vorzeichenerweiterung) mit einer Byte-Konstanten oder einer Word- bzw. DoubleWord-Konstanten multipliziert werden, mit einem anderen (passenden) Registerinhalt oder dem Inhalt einer Speicherstelle.



Bei Multiplikationen der Zwei-Operanden-Form müssen Quell- und Zieloperanden die gleiche Größe besitzen. Das bedeutet, dass das Ergebnis einer Multiplikation ggf. nicht korrekt ist – dann nämlich, wenn es den Wertebereich der eingesetzten Operanden überschreitet. In diesem Falle wird im Ziel lediglich der niedrigerwertige Anteil des Ergebnisses abgelegt, der höherwertige Teil schlichtweg verworfen und CF und OF gesetzt. Passte dagegen das Multiplikationsergebnis in das Ziel, werden CF und OF gelöscht.

In der Drei-Operanden-Form bezeichnet der erste Operand das Ziel (= Produkt), das immer ein Register sein muss. Allerdings dient dieser Operand *nicht* als Quelloperand. Vielmehr wird das Produkt aus den beiden folgenden Operanden gebildet: Operand 1 := Operand 2 · Operand 3, wobei Operand 2 (= Multiplikand) ein Register oder eine Spei-

cherstelle sein kann und Operand 3 (= Multiplikator) grundsätzlich eine Konstante ist. Diese kann entweder ein Byte sein, was dann vorzeichenerweitert verwendet wird, oder eine Konstante mit der gleichen Größe wie die beiden anderen Operatoren (Word bzw. DoubleWord). Es sind folgende Operationen definiert:

- Multiplikation eines Speicheroperanden mit einer vorzeichenerweiterten Konstanten und Ablage in einem Register
  `IMUL Reg16, Mem16, Const8; IMUL Reg32, Mem32, Const8`
- Multiplikation eines Registers mit einer vorzeichenerweiterten Konstanten und Ablage in einem anderen Register
  `IMUL Reg16, Reg16, Const8; IMUL Reg32, Reg32, Const8`
- Multiplikation eines Speicheroperanden mit einer Konstanten und Ablage in einem Register
  `IMUL Reg16, Mem16, Const16, IMUL Reg32, Mem32, Const32`
- Multiplikation eines Registers mit einer Konstanten und Ablage in einem anderen Register
  `IMUL Reg16, Reg16, Const16; IMUL Reg32, Reg32, Const32`

Auch in diesem Fall gilt, dass das Ergebnis ggf. um den höherwertigen Anteil »beschnitten« wird, da Zieloperand und alle Quelloperanden (Vorzeichenerweiterung!) die gleiche Größe haben. Daher signalisieren CF und OF, ob das Resultat in das Zielregister passte (CF = OF = 0) oder nicht.

DIV
IDIV

Ähnlich wie beim Paar MUL/IMUL verhält es sich bei der Integerdivision. Auch hier gibt es zwei Befehle, die entweder auf vorzeichenlose (DIV) oder vorzeichenbehaftete Integers (IDIV) angewendet werden.

Statusflags

Bei diesen beiden Befehlen spielen die Statusflags überhaupt keine Rolle, gelten also als undefiniert und sollten tunlichst nicht ausgewertet werden.



Bitte beachten Sie, dass sowohl DIV als auch IDIV trotz der verwirrenden Namensgebungen so genannte »Integerdivisionen« sind, also Divisionen, die keinen Nachkommateil erzeugen (sonst wäre das Resultat ja eine Realzahl!)! Das bedeutet, dass 3 DIV 2 = 3 IDIV 2 = 1, genauso wie 2 DIV 2 und 2 IDIV 2, und dass -3 IDV 2 = -1 ergibt, genauso wie -2 IDIV 2. Lediglich die ebenfalls berechneten Reste unterscheiden sich entsprechend.

Operanden

Auch bei den Divisionen kommen analog zu den Multiplikationen nur vergleichsweise wenige Möglichkeiten der Operatorenwahl in Betracht (XXX steht für DIV/IDIV):

- Expliziter (zweiter) Quelloperand ist ein Register
  `XXX Reg8, XXX Reg16, XXX Reg32`
- Expliziter (zweiter) Quelloperand ist eine Speicherstelle
  `XXX Mem8, XXX Mem16, XXX Mem32`

Auch hier ist der Dividend (= erste Quelloperand) und damit auch das Ziel (= Quotient) vorgegeben: der Akkumulator. Und auch hier sind je nach Größe des explizit angegebenen Divisors (Quelloperand 2!) damit die Quelle und das Ziel vorgegeben, wie die folgende Tabelle 1.3 zeigt:

expliziter Operand (Source-Operand #2)	durch expliziten Operanden festgelegte Datengröße	impliziter Operand (Source-Operand #1)	impliziter Zieloperand
Reg8 / Mem8	Byte	AX	AL; AH
Reg16 / Mem16	Word	DX:AX	AX; DX
Reg32 / Mem32	DoubleWord	EDX:EAX	EAX; EDX

Tabelle 1.3: Explizite und implizite Operanden des DIV-/IDIV-Befehls

Die Division eines (implizit in AX übergebenen) Dividenden durch den explizit über ein Byte-Register bzw. eine Byte-Speicherstelle übergebenen Divisor resultiert in einem ganzzahligen Divisionsergebnis, das im Byte-Akkumulator (AL) übergeben wird. AH enthält den Divisionsrest, ebenfalls in Byte-Form. Analog führt die Division des implizit in DX/EDX (höherwertiges Word/DoubleWord) und AX/EAX (niedrigerwertiges Word/DoubleWord) übergebene Dividenden durch den explizit übergebenen Word/DoubleWord-Divisor zur einem Word/DoubleWord-Divisionsergebnis in AX/EAX und einem Divisionsrest in DX/EDX.



Falls der Divisor bei DIV/IDIV »0« ist oder das Ergebnis der Division nicht in das Zielregister passt (z.B. $FFFF / $04 = $3FFF > $FF!), wird eine *divide error exception* (#DE) ausgelöst! Es werden also *nicht* wie bei den korrespondierenden Multiplikationen CF und OF verändert!



Das Vorzeichen des Divisionsrestes ist immer das gleiche wie das des Dividenden, es sei denn der Divisionsrest ist »0«, da bei der Integerdarstellung eine »-0« nicht existiert. Dies ist auch logisch, da ja die Umkehrrechnung (Quotient · Divisor + Rest = Dividend) gelten muss und

die Quotientenbildung durch »Runden in Richtung 0« erfolgt, was praktisch einem Abschneiden des imaginären Nachkommateils der Division entspricht. Konsequenterweise führt auch die Division von 3 durch -2 mittels IDIV zu -1 Rest 1 ($-1 \cdot -2 + 1 = 3$).



Der IDIV-Befehl hat anders als der IMUL-Befehl im Laufe der Evolution der Intel-Prozessoren keine wie auch immer geartete Erweiterung erfahren. Insbesondere können nicht Ziel- und erster Quelloperand explizit angegeben werden und es gibt auch keine Zwei- oder Drei-Operanden-Formen.

BCD-Korrekturen

Nachdem auch BCDs mit einander multipliziert und dividiert werden können, gibt es analog der Korrekturbefehle bei Addition und Multiplikation auch für diese Operationen Korrekturbefehle. Sie heißen AAM und AAD und werden etwas weiter unten besprochen.

ADC
SBB

Häufiger kommt es vor, dass man den Wertebereich von Integers gerne über die zur Verfügung stehenden Grenzen hinaus erweitern möchte, zumindest was Additionen und Multiplikationen betrifft. So gab es zu Zeiten der 16-Bit-Prozessoren den Wunsch, auch DoubleWords mit 32 Bits addieren oder subtrahieren zu können, seit den 32-Bit-Prozessoren sollten es 64-Bit-QuadWords sein. Hardwareseitig war das nicht sehr schwer, ließen sich doch 32-Bit-DoubleWords in zwei 16-Bit-Words aufteilen, die in zwei Allzweckregister passten. Und nachdem die Prozessoren vier solcher Allzweckregister hatten, konnte man auf diese Weise tatsächlich zwei DoubleWords bearbeiten. Gleiches gilt natürlich heute bei QuadWords und 32-Bit-Registern.

Will man nun zwei Zahlen auf diese Weise mit einander addieren, so muss zunächst der niedrigerwertige Teil beider Zahlen addiert werden (also z.B. das jeweilige niedrigerwertige DoubleWord der QuadWords). Hierbei kann ein Überlauf stattfinden. Dieser Überlauf muss bei der Addition der beiden höherwertigen Anteile berücksichtigt werden. Hierzu dient der Befehl ADC, *add with carry*. Nachdem der Überlauf nach der Addition zweier vorzeichenloser Zahlen mit dem carry flag signalisiert wird, ist dieses Flag der richtige Partner für ADC: Ist carry gesetzt, wird einfach zur Summe der beiden Operanden eine »1« addiert, andernfalls nicht.

Analoges gilt für die Subtraktion: Ergibt sich bei der Subtraktion der beiden niedrigerwertigen Anteile der QuadWords ein Unterlauf, wird

er im carry flag signalisiert. SBB, *subtract with borrow*, subtrahiert dann von der Differenz der beiden Operanden eine »1«. Andernfalls nicht.

Eine Addition und Subtraktion zweier QuadWords (z.B. in EDX:EAX und ECX:EBX) ist also ganz einfach zu erreichen:

```
ADD EAX, EBX  ; niedrigerwertige Anteile
ADC EDX, ECX  ; höherwertige Anteile mit Überlauf

SUB EAX, EBX  ; niedrigerwertige Anteile
SBB EDX, ECX  ; höherwertigen Anteile mit Unterlauf
```

Mit der jeweils ersten Instruktion werden die niedrigerwertigen DoubleWords addiert/subtrahiert, in der jeweils zweiten Zeile dann die höherwertigen, wobei ein Über-/ Unterlauf aus der ersten Operation im carry flag signalisiert und vom zweiten Befehl berücksichtigt wird.



Diese Kombinationen funktionieren sowohl bei vorzeichenlosen wie auch bei vorzeichenbehafteten Zahlen. Da nämlich im jeweils ersten Schritt die niedrigerwertigen Anteile der Daten addiert bzw. subtrahiert werden, spielen die Vorzeichen keine Rolle: Die niedrigerwertigen Anteile haben als vorzeichenlos interpretiert zu werden, weshalb das carry flag zur Erkennung eines Über-/Unterlaufs das richtige Flag ist. ADC und SBB arbeiten vollständig analog zu ADD und SUB mit der einzigen Ausnahme, dass das CF quasi ein impliziter dritter zu berücksichtigender Operand ist. Somit kann mit Hilfe der nach ADC/SBB gesetzten Flags die korrekte Interpretation erfolgen, je nachdem ob die QuadWords vorzeichenbehaftet waren oder nicht. (Zu den Flags nach ADC/SBB siehe ADD/SUB.)

Statusflags Die Statusflags werden durch ADC und SBB genauso behandelt wie durch die Zwillingsbefehle ADD und SUB.

Operanden ADC und SBB erlauben die gleichen Arten von Operatoren wie die korrespondierenden Befehle ADD und SUB (XXX dient im Folgenden als Platzhalter für ADC bzw. SBB):

- Addition/Subtraktion einer Konstanten zum/vom Akkumulatorinhalt
  `XXX AL, Const8; XXX AX, Const16; XXX EAX, Const32`
- Addition/Subtraktion einer Konstanten zu/von einem Registerinhalt
  `XXX Reg8, Const8; XXX Reg16, Const16; XXX Reg32, Const32`

- Addition/Subtraktion einer Konstanten zu/von einem Speicheroperand
  `XXX Mem8, Const8; XXX Mem16, Const16; XXX Mem32, Const32`
- Addition/Subtraktion einer vorzeichenerweiterten Byte-Konstanten zu/von einem Registerinhalt
  `XXX Reg16, Const8; XXX Reg32, Const8`
- Addition/Subtraktion einer vorzeichenerweiterten Byte-Konstanten zu/von einem Speicheroperand
  `XXX Mem16, Const8; XXX Mem32, Const8`
- Addition/Subtraktion eines Registerinhaltes zu/von einem Registerinhalt
  `XXX Reg8, Reg8; XXX Reg16, Reg16; XXX Reg32, Reg32`
- Addition/Subtraktion eines Speicheroperanden zu/von einem Registerinhalt
  `XXX, Reg8, Mem8; XXX Reg16, Mem16; XXX, Reg32, Mem32`
- Addition/Subtraktion eines Registerinhalts zu/von einem Speicheroperanden
  `XXX, Mem8, Reg8; XXX Mem16, Reg16; XXX Mem32, Reg32`

INC
DEC

INC, *increment*, und DEC, *decrement*, sind im Prinzip als »einfache« Additionen und Subtraktionen aufzufassen, die eine Zahl um 1 erhöhen oder erniedrigen. Insofern sind sie von den Aktionen her nichts Besonderes. Doch vom Ergebnis her unterscheiden sie sich ein wenig von ADD bzw. SUB.



Aus den Hochsprachen werden Sie die gleichnamigen Befehle kennen. Dort gibt es allerdings die Möglichkeit, zu dem zu inkrementierenden/dekrementierenden Wert eine beliebige Konstante, nicht notwendigerweise »1«, addieren/subtrahieren zu können. Dies ist beim Assembler nicht der Fall. INC und DEC können nur mit der (implizierten) Konstanten »1« arbeiten!

Statusflags

Das carry flag (CF) wird von beiden Befehlen nicht verändert, alle anderen Statusflags (OF, SF, AF und PF) werden anhand des Ergebnisses gesetzt: ZF, wenn das Ergebnis »0« ist, PF, wenn in den Bits 7 bis 0 eine gerade Anzahl gesetzter Bits vorliegt, und OF, wenn ein Über-/Unterlauf in/aus Bit 31 (DoubleWords), 15 (Words) oder 7 (Bytes) erfolgte, da dieses Bit ja das Vorzeichen repräsentiert. Das Vorzeichenbit wird in SF kopiert.



Der Grund dafür, dass das carry flag unangetastet bleibt, ist, dass auf diese Weise Schleifenzähler realisiert werden können, ohne den Zustand des carry flag zu verändern. Auf diese Weise können z.B. weitere Abbruchbedingungen der Schleife realisiert werden, die abhängig von einer anderen Prüfung in der Schleife sind:

```
Start: :
       :
       CMP  AL, BL ; setzt carry flag, wenn AL < BL
       :
       :
       DEC  CL     ; setzt zero flag, wenn CL = 0
       JNBE Start  ; springt zurück, solange AL > BL (CF =
       :           ; 0) und CL > 0 (ZF = 0)
       :
```

Das Haupteinsatzgebiet von INC/DEC ist daher auch die Realisierung eines Schleifenzählers.



Da INC/DEC das carry flag nicht verändern, müssen Sie ADD/SUB mit einem Operanden »1« verwenden, wenn Sie nicht-vorzeichenbehaftete Zahlen um 1 inkrementieren oder dekrementieren und das carry flag zwecks Feststellung eines Über-/Unterlaufs auswerten wollen.

Operanden

INC und DEC erlauben das Inkrementieren und Dekrementieren von Registerinhalten oder Speicherstellen (XXX dient im Folgenden als Platzhalter für INC bzw. DEC):

- Inkrementieren/Dekrementieren eines Registerinhaltes
  `XXX Reg8; XXX Reg16; XXX Reg32`
- Inkrementieren/Dekrementieren einer Speicherstelle
  `XXX Mem8; XXX Mem16; XXX Mem32`



Beachten Sie bitte, dass es für die Codierung der Registervarianten zwei Opcodes gibt, wenn ein 16-Bit- oder ein 32-Bit-Register inkrementiert/dekrementiert wird: Ein-Byte-Opcodes und Zwei-Byte-Opcodes. In der Regel wird aber der Assembler den optimalen Code erzeugen. ACHTUNG: Die Existenz von Zwei-Byte-Codes ist leider nicht analog den Befehlen AAD und AAM zu sehen, indem das zweite Byte die Inkrementationskonstante codiert (s.u.). Das zweite Byte ist tatsächlich als Teil des Opcodes für die Codierung »Inkrementieren/Dekrementieren um 1« notwendig.

NEG

NEG, *negate*, ist ein sehr einfacher arithmetischer Befehl: Er bildet den »arithmetisch negierten« Wert – das »2er-Komplement« –, also quasi das Ergebnis der Operation (0 – Wert) und hat daher nur dann einen Sinn, wenn die Daten als vorzeichenbehaftet interpretiert werden.

Statusflags

Daher hat das carry flag hier auch eine leicht andere Bedeutung: Es ist nur dann gelöscht, wenn der Operand den Wert »0« hat; somit hat es den entgegengesetzten Wert zum zero flag. Dies ist auch sinnvoll, da ja außer bei 0, wo kein »Borgen« notwendig wird, bei jedem anderen Wert ein Unterlauf erfolgen *muss*, den das carry flag logischerweise signalisiert. Alle anderen Statusflags werden anhand des Ergebnisses wie gewohnt gesetzt: zero flag, wenn der Inhalt des Operanden nach der Negierung 0 ist (was nur dann der Fall sein kann, wenn der Operand vorher ebenfalls den Wert 0 hatte), sign flag, wenn das MSB gesetzt ist. Das overflow flag wird immer gelöscht, da es niemals eine Über- oder Unterschreitung des Wertebereichs geben kann. Das parity flag wird gesetzt, wenn die Anzahl gesetzter Bits in niedrigstwertigen Byte des Operanden gerade ist, und das adjust flag ist ebenso wie das carry flag immer dann gesetzt, sobald der zu negierende Operand einen Wert größer als 0 besitzt, da es dann grundsätzlich einen Unterlauf aus Bit 4 in Bit 3 des Operanden geben *muss*.

Operanden

Als Operanden kommen bei NEG nur Registerinhalte oder Speicherstellen in Frage:

- Negieren eines Registerinhaltes
  `NEG Reg8; NEG Reg16; NEG Reg32`
- Negieren einer Speicherstelle
  `NEG Mem8; NEG Mem16; NEG Mem32`

Weitere Operanden machten auch keinen Sinn!

AAA
AAS
AAM
AAD

AAA, *ASCII adjust after addition*, AAS, *ASCII adjust after subtraction*, AAM, *ASCII adjust after multiplication*, und AAD, *ASCII adjust before division*, sind eigentlich keine »echten« arithmetischen Befehle, sondern eher »Korrekturbefehle«. Sie dienen dazu, die Fehler zu korrigieren, die entstehen, wenn man binär ausgerichtete Operationen auf »dezimale« Daten anwendet (die ja eigentlich nicht wirklich dezimal, sondern nur dezimal codiert und ansonsten binär sind). Da sie aber im Kontext zu arithmetischen Befehlen zu sehen sind, werden sie auch an dieser Stelle behandelt. AAA, AAS und AAM sind Operationen, die eine *vorangegangene* arithmetische Operation korrigieren, während AAD eine Division *vorbereitet*, damit das Ergebnis eine korrekte BCD ist. (Falls Sie In-

formationen zu BCDs benötigen, konsultieren Sie bitte das Kapitel »Datenformate« auf Seite 778.)

Die Namen der Korrekturbefehle sind nicht gerade selbsterklärend! Sie resultieren aus einer der Hauptanwendungen nicht gepackter BDCs. So sind solche BCDs recht einfach in die ASCII-Codes der korrespondierenden Zeichen überführbar, indem man zu der in den Bits 3 bis 0 eines Bytes codierten BCD-Ziffer den Wert $30 (und damit einen Inhalt der Bits 4 bis 7 des Bytes) addiert: Das ASCII-Zeichen »0« hat den Code $30, das ASCII-Zeichen »9« den Code $39. AAA, AAS, AAM und AAD wurden (und werden heute noch manchmal) daher eingesetzt, um ASCII-Ziffern zu erzeugen.

Operanden Alle Befehle haben keinen expliziten Operanden. Vielmehr implizieren sie den Akkumulator als Quell- und Zieloperanden. Sie gehen von folgenden Voraussetzungen aus:

- Es wurden/werden *ungepackte* BCDs (eine Ziffer pro Byte) manipuliert.
- Das Ergebnis bzw. der Dividend der mathematischen Operation steht in AL (eine Ziffer) bzw. AH und AL (zwei Ziffern), wobei in AH die höherwertige und in AL die niedrigerwertige Ziffer steht.
- Das adjust flag AF wurde an die Situation angepasst (nicht bei AAD und AAM).

Die einzelnen Korrekturbefehle führen nun folgende Operationen aus:

```
AAA:
if (AL > 9) or AF = 1
   then (AL := AL + 6) mod 16; AH := AH + 1; AF := 1; CF := 1;
   else AF := 0; CF := 0;

AAS:
if (AL > 9) or AF = 1
   then (AL := AL – 6) mod 16; AH := AH – 1; AF := 1; CF := 1;
   else AF := 0; CF := 0;

AAM:
AH := AL div 10; AL := AL mod 10;

AAD:
AL := AH * 10 + AL; AH := 0;
```

Nach einer Addition zweier ungepackter BCD-Ziffern können drei Fälle auftreten:

- Das Ergebnis ist eine BCD-Ziffer im Bereich 0 bis 9. Dann ist alles OK, weshalb AAA lediglich AF und CF löscht, um diesen Sachverhalt zu signalisieren.
- Das Ergebnis liegt zwischen 10 und 15. Dann ist AF zwar nicht gesetzt, da kein Überlauf in Bit 4 stattgefunden hat, aber AL ist größer als 9. In diesem Fall wird 6 (binär) addiert und das Ergebnis modulo 16 genommen. Es liegt somit zwischen 0 ($0A + 6 = $10 = 16; 16 mod 16 = 0) und 5 ($0F + 6 = $15 = 21; 21 mod 16 = 5). Die zweite Ziffer (immer eine 1!) wird zum Inhalt in AH addiert und AF und CF gesetzt, um den Überlauf in AH zu signalisieren.
- AF ist gesetzt, da die Addition einen Überlauf in Bit 4 erzeugte. Dies ist der Fall, wenn das Ergebnis zwischen 16 und 18 liegt (18 ist der maximal mögliche Wert, wenn zwei BCD-Ziffern mit maximalem Wert 9 addiert wurden!). In diesem Fall erfolgt der gleiche Vorgang wie zuvor.

Analoges gilt für die Korrektur nach Subtraktion, nur mit umgekehrtem Vorzeichen (im wahrsten Sinne des Wortes!):

- Liegt das Ergebnis zwischen 9 und 0, werden lediglich die Flags AF und CF gelöscht, da eine gültige BCD-Ziffer vorliegt.
- Wenn AF gesetzt ist, da binäre Werte zwischen $FF (= 0 – 1) und $F7 (= 0 – 9), die nach der Subtraktion entstehen können und korrigiert werden müssten, nur durch Borgen aus Bit 4 entstehen können und damit einen Unterlauf hervorrufen, der durch ein gesetztes AF angezeigt wird, oder/und das Ergebnis > 9 ist, wird der Wert 6 vom Ergebnis abgezogen und dieses modulo 16 genommen, was zu Werten zwischen 9 ($FF – 6 = $F9; $F9 mod 16 = 9) und 1 ($F7 – 6 = $F1; $F1 mod 16 = 1) führt. Dann wird 1 von AH abgezogen (Borgen!) und AF und CF zum Signalisieren des Unterlaufs gesetzt.

Die Korrektur nach Multiplikationen ist relativ klar: Da die eigentliche Multiplikation mittels MUL erfolgte (IMUL darf nicht verwendet werden, da CPU-BCDs per definitionem kein Vorzeichen besitzen), ist das Ergebnis eine binär codierte Zahl im Bereich 0 bis 81 (= $00 bis $51), die in zwei Dezimalteile aufgeteilt werden muss. Dies erfolgt durch Division mit 10, wobei das Divisionsergebnis (höherwertige Ziffer) in AH abgelegt wird, der Divisionsrest (niedrigerwertige Ziffer) in AL.

Auch die Vorbereitung einer Division ist klar, sie erfolgt umgekehrt zur Multiplikation: Die in AH und AL abgelegten Ziffern einer zweistelligen, ungepackten BCD werden durch Multiplikation der höherwertigen Ziffer (AH) mit 10 und Addition der niedrigerwertigen Ziffer (AL) in eine Binärzahl umgeformt, die dann als Dividend der Division fungieren kann.

Statusflags Das overflow flag, das sign flag sowie die Flags zero und parity sind nach AAA und AAS undefiniert. Das carry flag sowie das adjust flag wurden gesetzt, wenn durch die Korrektur ein dezimaler Über-/Unterlauf in das/vom AH-Register erfolgte, andernfalls sind sie gelöscht. Nach AAM und AAD sind das carry flag, das adjust flag sowie das overflow flag undefiniert, zero flag, sign flag und parity flag werden aufgrund des binären Resultates der Operation entsprechend gesetzt.



Alle hier vorgestellten Korrekturbefehle gehen davon aus, dass die Randbedingungen für das korrekte Arbeiten von AAA, AAS, AAM und AAD gegeben und vom Programmierer sichergestellt sind. Aus diesem Grunde erfolgt auch keine weitergehende Korrektur oder Kontrolle der eingesetzten Operanden oder resultierenden Ergebnisse. So erwartet AAA, dass zwei einziffrige BCDs addiert wurden. Zwar ist aufgrund der Tatsache, dass die Korrektur durch Inkrementieren des AH-Registers erfolgt, auch die Addition einer Ziffer zu einer zweiziffrigen (ungepackten!) BCD möglich, doch darf dann das Resultat den Wert $0909 (ungepackte BCD-Ziffern befinden sich in AH und AL!) nicht überschreiten: So führt z.B. die Addition von 1 zu 99 mit BCD-Ziffern zunächst zum vorläufigen binären Ergebnis $090A ($99_{BCD} + 1_{BCD}$= $0909 + $01 = $090A), das durch AAA in $0A00 und damit ein falsches Ergebnis »korrigiert« wird: Die niedrigerwertige Ziffer stimmt, aber der Überlauf wird lediglich ungeprüft und unkorrigiert zu AH addiert. Analog geht AAS davon aus, dass in AH eine von »0« verschiedene Ziffer existiert, wenn eine größere von einer kleineren Ziffer abgezogen wird. Denn AAS verarbeitet den entstehenden Überlauf durch unkorrigiertes und ungeprüftes Dekrementieren von AH. Steht dort 0, resultiert der falsche Wert $FF! Schließlich geht auch AAM davon aus, dass maximal zwei einziffrige BCD-Zahlen miteinander multipliziert werden. Ist dies nicht gewährleistet, kann das zu vollkommen unerwarteten oder gar falschen Ergebnissen führen.

Auch die vorbereitende Erzeugung eines divisionsfähigen Dividenden durch AAD erfordert die Einhaltung der Randbedingungen. Dazu ein Beispiel: Die Division der nicht gepackten BCD 98 (bei der die Ziffer 9

in AH und die Ziffer 8 in AL steht!) durch die BCD 2 führt zunächst durch das vorbereitende AAD zu der binären Zahl $62 ($98_{BCD}$ = $0908 = 9 · 10 + 8 = 98 = $62) in AL. Diese Zahl durch 2 dividiert ergibt $31 = 31_{BCD} = $0301, was weit weg ist vom korrekten Ergebnis 49_{BCD} = $0409. Grund: Das Ergebnis erfordert mehr als eine Ziffer, ist also ohne Korrektur nicht darstellbar. Dies ist aber nicht im Sinne der BCD-Arithmetik-Philosophie! Nur dann, wenn die BCD-Division als Umkehrung der maximal möglichen Multiplikation mit zwei einzelnen BCD-Ziffern angesehen wird, wobei der maximal erlaubte Wert des Dividenden durch den Divisor vorgegeben wird, kann ein korrektes Ergebnis herauskommen: 9 · 9 = 81; daher transformiert AAD 81_{BCD} in $51 und die Division von $51 durch $09 ergibt $09 oder 9_{BCD}. Somit sind bei einem Divisor von 9 alle Dividenden oberhalb 81 = 9 · 9 verboten. (Anderes Beispiel: 9 · 5 = 45, daher 45_{BCD} → $2D; $2D / $05 = $09 = 9_{BCD}; somit sind bei einem Divisor von 5 alle Dividenden > 9 · 5 = 45 verboten.) Die korrekten Randbedingungen für die Rechnung mit BCDs sicherzustellen und einzuhalten, liegt in der Verantwortung des Programmierers.



AAM und AAD sind Zwei-Byte-Instruktionen, was bedeutet, dass ihr Opcode im Gegensatz zu den Ein-Byte-Instruktionen AAA und AAS aus zwei Bytes besteht. Sie werden das nur dann feststellen, wenn Sie das Assemblat im Debugger betrachten. Dann nämlich werden Sie sehen, dass AAM mit $D40A und AAD mit $D50A übersetzt wird. $D4 steht hierbei für AAM und $D5 für AAD. $0A ist in beiden Fällen eine Konstante (mit dem Wert $0A = 10), die der Assembler automatisch als expliziten (zweiten!) Operanden einfügt. Wer nun ein bisschen nachdenkt, wird schnell feststellen, dass die Konstante 10 gerade der Wert ist, der bei der Multiplikation/Division, die die beiden Befehle durchführen, verwendet wird. Damit erhebt sich die Frage, ob auch andere Werte als $0A verwendet werden können.

Ja, sie können. Es kann anstelle von $0A jeder beliebige Byte-Wert als Operand übergeben werden, der dann für die Multiplikation/Division herangezogen wird. Das bedeutet, dass die verallgemeinerten Versionen von AAM und AAD eine einfache Art der Umrechnung einer Ziffer der Basis 2 (binäre Zahlen) in eine Ziffer der Basis A und umgekehrt ermöglichen. Mit verschiedenen Kombinationen der erweiterten AAM-/AAD-Instruktionen sind sogar Umrechnungen von Zahlen der Basis A in Zahlen der Basis B möglich oder das Packen oder Entpacken von BCDs.



Allerdings gibt es hierzu kein Mnemonic, sodass der Assembler dies nicht unterstützt: Die Befehlssequenz muss »von Hand« erzeugt werden. Das ist aber sehr einfach, indem man die DB- oder DW-Instruktionen des Assemblers verwendet. Ein Beispiel dafür finden Sie auf der beiliegenden CD-ROM.

DAA
DAS

Was für ungepackte BCDs gilt, sollte, zumindest teilweise, auch für gepackte BCDs gelten. Daher gibt es mit DAA, *decimal adjust after addition*, und DAS, *decimal adjust after subtraction*, zwei Befehle, die AAA und AAS sehr ähnlich sind. Einziger Unterschied: Sie erwarten zwei BCD-Ziffern pro Byte.

Im Einzelnen machen die Befehle Folgendes:

```
DAA:
if (AL[3..0] > 9) or AF = 1
   then AL := AL + 6; AF := 1; CF := CF or AF;
   else AF := 0;
if (AL[7..4] > 9) or CF = 1
  then AH := AL + $60; CF := 1;
   else CF := 0;

DAS:
if (AL[3..0] > 9) or AF = 1
   then AL := AL – 6; AF := 1; CF := CF or AF;
   else AF := 0;
if (AL[7..4] > 9) or CF = 1
   then AL := AL - $60; CF := 1;
   else CF := 0;
```

Im Prinzip ist DAA ein zweimal hintereinander ausgeführtes AAA, wobei einmal die niedrigerwertige Ziffer in den Bits 3 bis 0 von AL korrigiert wird und danach die höherwertige Ziffer in den Bits 7 bis 4 von AL. Da die BCDs hier aber gepackt sind, muss keine Restbildung (*mod*) erfolgen mit Übertrag vom/ins AH-Register. Vielmehr wird der Korrekturfaktor 6 addiert, sobald die Ziffer in AL[3..0] größer als 9 oder AF gesetzt ist. Ein eventuell stattfindender Überlauf erfolgt genau da, wo er hin soll: in die Bits 7..4 der zweiten Ziffer. Dieser Überlauf wird durch Setzen des AF signalisiert. Gleichzeitig aber wird auch CF gesetzt, sodass CF nun immer dann gesetzt ist, wenn entweder die Addition vor DAA einen Überlauf erzeugte (der ja in der zweiten Ziffer der gepackten BCD korrigiert werden muss!) oder die Korrektur der niedrigerwertigen Ziffer einen Überlauf (AF!) erzeugte.

In einem zweiten Schritt wird nun die in AL[7..4] stehende, zweite BCD korrigiert. Und zwar nur dann, wenn CF gesetzt ist und damit einen additions- oder korrekturbedingten Überlauf signalisiert oder der Wert größer als 9 ist. Dann wird der Korrekturfaktor $60 addiert und CF gesetzt. Ein Überlauf in ein anderes Register, wie bei AAA, erfolgt nicht! Ein solcher Überlauf muss via CF behandelt werden.

Das gleiche Prinzip liegt auch bei DAS vor, das als zweifaches Ausführen von AAS mit den beiden gepackten BCD-Ziffern aufgefasst werden kann.

Statusflags

Das overflow flag ist nach DAA und DAS undefiniert. Ein gesetztes adjust flag signalisiert eine erfolgte Korrektur des niedrigerwertigen Nibbles (Bit 3 bis 0), carry flag eine Korrektur des höherwertigen Nibbles (Bit 7 bis 4) der gepackten BCD. Das sign flag ist gesetzt, wenn Bit 7 auch gesetzt ist, spiegelt hier aber *nicht* die Stellung eines Vorzeichens wider, da CPU-BCDs definitionsgemäß vorzeichenlos sind! Das parity flag wird anhand der Gegebenheiten ebenso gesetzt wie das zero flag.



Korrekturbefehle für Multiplikation und Division gepackter BCDs gibt es nicht, da eine Multiplikation/Division mit gepackten BCDs via MUL/DIV nicht möglich ist! Hierzu müssten zunächst die gepackten BCDs entpackt und in eine binäre Zahl konvertiert werden, bevor die Multiplikation/Division erfolgen könnte. Das Ergebnis müsste dann zunächst wieder in die Form entpackter BCDs gebracht werden, die dann gepackt werden müssten. Der Aufwand hierzu rechtfertigt aber aufgrund der beschränkten Anwendungsgebiete für gepackte BCDs nicht die Entwicklung entsprechender Befehle.

1.1.2 Logische Operationen

Wie in der Einleitung zum Kapitel der CPU-Operationen bereits gesagt, können die Daten, mit denen die CPU arbeitet, sehr unterschiedlich interpretiert werden. Alle arithmetischen Operationen betrachten die Bits der Operanden als nicht voneinander unabhängig: Sie codieren eine wie auch immer geartete Zahl. Veränderungen an einzelnen Bits (z.B. die Addition zweier gesetzter Bits 0 zweier Zahlen) hat bei solchen Instruktionen immer Auswirkungen auf die anderen Bits (weil z.B. Überträge berücksichtigt werden).

Die »logischen« Operationen dieses Abschnitts dagegen fassen die Bits der Operanden als eigenständige, logische Zustände auf, die nur die Werte »0« und »1« für die Inhalte »falsch« und »wahr« annehmen können. Überträge gibt es nicht! Dementsprechend operieren die folgenden Operationen bitweise und der Inhalt der Operanden wird als Bitfeld von unabhängigen Bits interpretiert.

AND
OR
XOR
NOT

Die CPU kennt mit AND, OR, XOR und NOT die vier grundlegenden Operationen aus dem Bereich der »Logik«. AND führt eine bitweise AND-Verknüpfung durch, bei der ein Ergebnisbit dann und nur dann gesetzt ist, wenn beide korrespondierenden Ausgangsbits ebenfalls gesetzt waren. Andernfalls wird es gelöscht. Bei der ODER-Verknüpfung ist das Ergebnisbit dann gesetzt, wenn eines oder beide Ausgangsbits ebenfalls gesetzt waren. Die »Ausschließende Oder«-Verknüpfung (*eXclusive OR*) liefert ein gesetztes Ergebnisbit, wenn eines der beiden Ausgangsbits gesetzt war, nicht aber beide. Und die logische Negierung NOT bildet das 1er-Komplement, bei dem das Ergebnisbit gesetzt wird, wenn das Ausgangsbit gelöscht war und umgekehrt. AND, OR und XOR verknüpfen damit zwei Bits miteinander, während NOT lediglich den Zustand eines Bits »umdreht«: Aus »1« wird »0« bzw. aus »0« wird »1«. Die Ergebnisse lassen sich in Tabelle 1.4 zusammenfassen:

Bit 2:	0	1	0	1	0	1	
Bit 1:	AND		OR		XOR		NOT
0	0	0	0	1	0	1	1
1	0	1	1	1	1	0	0

Tabelle 1.4: Darstellung der Bitstellungen nach den logischen Operationen AND, OR, XOR und NOT

Operanden

AND, OR und XOR erlauben verschiedene Arten von Operanden (XXX dient im Folgenden als Platzhalter für AND, OR und XOR):

- Logische Verknüpfung des Akkumulatorinhalts mit einer Konstanten
  ```
  XXX AL, Const8; XXX AX, Const16; XXX EAX, Const32
  ```
- Logische Verknüpfung eines Registerinhalts mit einer Konstanten
  ```
  XXX Reg8, Const8; XXX Reg16, Const16; XXX Reg32, Const32
  ```
- Logische Verknüpfung eines Speicheroperands mit einer Konstanten
  ```
  XXX Mem8, Const8; XXX Mem16, Const16; XXX Mem32, Const32
  ```

- Logische Verknüpfung eines Registerinhalts mit einer vorzeichenerweiterten Byte-Konstanten
  `XXX Reg16, Const8; XXX Reg32, Const8`
- Logische Verknüpfung eines Speicheroperands mit einer vorzeichenerweiterten Byte-Konstanten
  `XXX Mem16, Const8; XXX Mem32, Const8`
- Logische Verknüpfung eines Registerinhaltes mit einem Registerinhalt
  `XXX Reg8, Reg8; XXX Reg16, Reg16; XXX Reg32, Reg32`
- Logische Verknüpfung eines Registerinhalts mit einem Speicheroperanden
  `XXX, Reg8, Mem8; XXX Reg16, Mem16; XXX, Reg32, Mem32`
- Logische Verknüpfung eines Speicheroperanden mit einem Registerinhalt
  `XXX, Mem8, Reg8; XXX Mem16, Reg16; XXX Mem32, Reg32`

Naturgemäß sind die Möglichkeiten bei der logischen Verneinung NOT eingeschränkt, da durch diesen Befehl keine Verknüpfung erfolgt sondern eine Veränderung bestehender Bits, somit nur ein Quelloperand in Frage kommt, der gleichzeitig auch Zieloperand ist:

- Logische Verneinung eines Registerinhalts
  `NOT, Reg8; NOT Reg16; NOT, Reg32`
- Logische Verneinung eines Speicheroperanden
  `NOT, Mem8; NOT Mem16; NOT Mem32`



Beachten Sie bitte, dass die genannten Bitverknüpfungen immer zwischen den korrespondierenden Bits der beiden Operanden der Befehle AND, OR und XOR erfolgen, nie aber »innerhalb« eines Operanden! Die einzelnen Bits eines Operanden sind und bleiben voneinander unabhängig:

```
AND:
for I := 0 to Length(Destination – 1) do
  Destination[I] := Source1[I] and Source2[I]

OR:
for I := 0 to Length(Destination – 1) do
  Destination[I] := Source1[I] or Source2[I]

XOR:
for I := 0 to Length(Destination – 1) do
  Destination[I] := Source1[I] XOR Source2[I]
```

```
NOT:
for I := 0 to Length(Destination – 1) do
  Destination[I] := not Source1[I]
```

Statusflags Durch AND, OR und XOR werden das carry und overflow flag explizit gelöscht. Dies signalisiert richtigerweise, dass es nach logischen Verknüpfungen weder einen vorzeichenlosen (carry flag) noch vorzeichenbehafteten (overflow flag) Über- bzw. Unterlauf geben kann: Die Operanden der Befehle sind einzelne, von einander unabhängige Bits, keine Zahlen. Aus diesem Grunde spielt auch das adjust flag keine Rolle: Es ist nach den logischen Verknüpfungen undefiniert.

Zero flag, sign flag und parity flag dagegen werden entsprechend dem Ergebnis gesetzt: Sind alle Bits gelöscht, so ist zero flag gesetzt, andernfalls gelöscht. Das sign flag ist eine Kopie des Bits 31, 15 oder 7 – je nach Größe des eingesetzten Operanden. Somit erhebt es die genannten Bits in einen Sonderstatus, da sie durch die Verknüpfung mit dem sign flag »ein wenig gleicher« sind als die anderen Bits des Bitfeldes, die ja eigentlich alle untereinander gleich sind! Und das parity flag zeigt im gesetzten Zustand eine gerade Parität an, die sich in einer geraden Anzahl gesetzter Bits äußert.

NOT dagegen verändert keine Flags – warum auch?



Die logischen Verknüpfungen wirken, wie gesagt, auf einzelne Bits, weshalb die Inhalte der Operanden auch als Felder voneinander unabhängiger Bits und nicht als Zahl interpretiert werden (müssen). Allerdings gibt es eine Ausnahme: Wenn man eine Integer (im Beispiel ein DoubleWord) darstellt als Summe fallender Potenzen von 2:

$$I = Bit_{31} \cdot 2^{31} + Bit_{30} \cdot 2^{30} + ... + Bit_1 \cdot 2^1 + Bit_0 \cdot 2^0,$$

so lassen sich zwar nicht die Komponenten dieser Reihe als unabhängig voneinander betrachten (weil sie addiert werden) und verändern, wohl aber die Koeffizienten (Bit_x) der jeweiligen 2er-Potenzen. Auf diese Weise lassen sich auch zwei »Zahlen« logisch verknüpfen – mit teilweise frappierendem Ergebnis. So können einzelne Bits dieser Zahlen gezielt verändert werden. Zum Beispiel lassen sich die letzten acht Bits durch AND-Verknüpfung mit einer Zahl, bei der die letzten acht Bits gelöscht sind, löschen. Zu beachten ist dabei jedoch, dass die restlichen 24 Bits gesetzt sein müssen, sollen sie unverändert bleiben. Dies führt dazu, dass die Zahl, mit der verknüpft werden soll, die Konstante $FFFF_FF00 ist (Bits 31 bis 8 gesetzt, Bits 7 bis 0 gelöscht). Verknüpft man obige Integer mit dieser »Maske«, wie man solche Bit-Konstanten

nennt, mit einer AND-Verknüpfung, so sind die Bits 31 bis 8 des Ergebnisses je nach der Stellung in I gesetzt oder gelöscht und die Bits 7 bis 0 auf jeden Fall gelöscht. Betrachtet man das Resultat wiederum als Zahl, so wurde praktisch die Integer zunächst durch 256 dividiert und der resultierende Quotient der Integerdivision mit 256 multipliziert. I wurde also um den Rest einer Division durch $0000_0100 vermindert:

(I and $FFFF_FF00) ≡ I := 256 · (I div 256) = I – (I mod 256)

Erzeugt man dagegen eine Maske, in der alle Bits außer den letzten 8 Bits gelöscht sind ($0000_00FF), und führt damit eine AND-Verknüpfung durch, so sind im Ergebnis nur diejenigen der letzten acht Bits gesetzt, die auch in I gesetzt waren. Arithmetisch betrachtet handelt es sich also um eine Restbildung nach Division mit $0000_0100, auch Modulo-Bildung genannt:

(I and $0000_00FF) ≡ I := I – (256 · (I div 256)) = I mod 256



Beachten Sie bitte, dass nicht jede logische Verknüpfung und nicht jede »Maske« Sinn machen. So ergibt die Modulo-Bildung nur dann korrekte Ergebnisse, wenn als Maske »Zahlen« verwendet werden, die mit Bit 0 beginnend lückenlos bis zu dem gewünschten Divisor gesetzt sind. Damit kommen (bei Byte-Betrachtung!) nur die Zahlen 1 (Bit 0 gesetzt), 3 ($03: Bit 0 und 1 gesetzt), 7 ($07: Bit 0, 1 und 2 gesetzt), 15 ($0F), 31 ($1F), 63 ($3F), 127 (7F) und 255 ($FF) in Frage. Sie entsprechen der Modulo-Bildung mit (Maske + 1), also 2 ($01 + 1 = $02), 4 ($04), 8 ($08), 16 ($10), 32 ($20), 64 ($40), 128 ($80), 256 ($100). Der Versuch, eine Zahl mit der Maske $00F5 UND-zuverknüpfen und anzunehmen, das Ergebnis wäre der Modulus 246 ($F5 + 1 = 245 + 1) einer Zahl, scheitert! Vielmehr ist das Ergebnis eine Zahl, bei der die Bits 3 und 1 explizit gelöscht sind, was nicht zwingend nur beim Modulus 246 der Fall ist. Der Grund dafür ist, dass die Bits, aus denen die Zahlen zusammengesetzt sind, eben alles andere als unabhängig voneinander sind.

Auch die Verwendung der XOR-Verknüpfung von »echten« Zahlen miteinander oder mit Masken wird in den seltensten Fällen Sinn machen. So kann man mittels XOR eine einfache Form der Datenverschlüsselung durchführen. Die Operation C = D XOR Maske verschlüsselt das DoubleWord D mit Maske zur Chiffre C, die ihrerseits mit Hilfe von Maske zum ursprünglichen DoubleWord D entschlüsselt werden kann: D = C XOR Maske. Wie gesagt: ein einfaches Verschlüsselungsverfahren, das sicherlich nicht mit PGP und anderen professionellen Verschlüsselungsprogrammen konkurrieren kann, jedoch für manche Fälle

durchaus brauchbar ist. Eine andere Anwendung von XOR auf Zahlen ist die Bildung von Prüfsummen zum Zwecke der Verifizierung von Datenströmen, bei der dann die Daten nicht mit Masken, sondern mit folgenden Daten geXORt werden.

Und auch OR macht nur dann Sinn, wenn im Ergebnis bestimmte Bits auf jeden Fall gesetzt sein sollen. So kann die Umrechnung einer BCD in das korrespondierende ASCII-Zeichen entweder durch Addition der Konstanten $30 erfolgen oder (was gleichbedeutend ist!) durch ODER-Verknüpfung mit $30.



Allerdings gibt es noch je einen Anwendungsfall für OR und XOR, der in Assembler-Quelltexten häufig zu finden ist. So führt die OR-Verknüpfung eines Operanden mit sich selbst etwa der Form OR EAX, EAX zu einem Ergebnis, das zunächst nicht sehr sinnvoll erscheint: Es hat sich am Inhalt nichts verändert. Doch darf nicht vergessen werden, dass OR auch Flags setzt. Und das kann immer dann eine wertvolle Hilfe sein, wenn (mit Hilfe der Flags) eine Programmverzweigung aufgrund des Inhaltes des Operanden erfolgen soll, die vorangegangene Operation aber keine Flags gesetzt hat:

```
      MOV EAX, [Mem32] ; MOV verändert keine Flags
      OR  EAX, EAX     ; Flags setzen anhand des Wertes
      JZ  Zero         ; verzweigt, wenn EAX = 0.
      :
      :
Zero: :
      :
```

Zwar könnte man dies auch mittels eines arithmetischen Vergleiches mit CMP erreichen und hätte dann sogar die Möglichkeit, andere Bedingungen zu prüfen (größer, kleiner etc.). Doch ist OR kürzer, schneller, effektiver und häufig absolut ausreichend.

Einen ähnlichen Trick kann man mit XOR machen. Verknüpft man analog den Operanden mit sich selbst, wie in XOR EAX, EAX, so kommt als Resultat 0 heraus. Denn die XOR-Verknüpfung von 0 mit 0 und 1 mit 1 ergibt jeweils 0. Dies ist eine schnelle, einfache und effiziente Art, einen Operanden zu löschen. Andernfalls müsste man den ineffektiveren Weg über MOV EAX, 0 nehmen ...

1.1.3 Operationen zum Datenvergleich

Datenvergleich ist sowohl bei »arithmetischen« Daten, also Integers, wichtig wie auch bei logischen Werten, also Feldern aus »Wahrheitswerten«. Dementsprechend gibt es zwei Befehle, die genau das ermöglichen: CMP und TEST.

CMP

CMP ist der »arithmetische« Datenvergleich. Mit Hilfe dieses Befehles lassen sich zwei Zahlen mit einander vergleichen. Als Ergebnis werden die Statusflags gesetzt, die dann ein Auswerten des Ergebnisses in Form von Programmverzweigungen ermöglichen.



Der CMP-Befehl ist eigentlich ein verkappter SUB-Befehl, da er tatsächlich den zweiten Quelloperanden vom ersten abzieht. Der Unterschied zu SUB besteht nun allein darin, dass dieses Ergebnis nicht in ein Ziel transferiert wird; vielmehr werden analog SUB lediglich die Flags gesetzt und das Ergebnis danach verworfen. CMP ist somit einer der wenigen Befehle, die zwar Quelloperanden besitzen, aber keine Zieloperanden – noch nicht einmal implizit! CMP verändert die Inhalte der Operanden nicht.

Operanden

Nachdem CMP eigentlich ein SUB-Befehl ist, verfügt er auch über die analogen Möglichkeiten der Operandennutzung:

- Vergleich des Akkumulators mit einer Konstanten
  `CMP AL, Const8; CMP AX, Const16; CMP EAX, Const32`
- Vergleich eines Registerinhalts mit einer Konstanten
  `CMP Reg8, Const8; CMP Reg16, Const16; CMP Reg32, Const32`
- Vergleich eines Speicheroperands mit einer Konstanten
  `CMP Mem8, Const8; CMP Mem16, Const16; CMP Mem32, Const32`
- Vergleich eines Registerinhalts mit einer vorzeichenerweiterten Byte-Konstanten
  `CMP Reg16, Const8; CMP Reg32, Const8`
- Vergleich eines Speicheroperanden mit einer vorzeichenerweiterten Byte-Konstanten
  `CMP Mem16, Const8; CMP Mem32, Const8`
- Vergleich eines Registerinhaltes mit einem Registerinhalt
  `CMP Reg8, Reg8; CMP Reg16, Reg16; CMP Reg32, Reg32`
- Vergleich eines Registerinhalts mit einem Speicheroperanden
  `CMP, Reg8, Mem8; CMP Reg16, Mem16; CMP, Reg32, Mem32`
- Vergleich eines Speicheroperanden mit einem Registerinhalt
  `CMP, Mem8, Reg8; CMP Mem16, Reg16; CMP Mem32, Reg32`

Statusflags Wie kann man die Flags heranziehen, um festzustellen, welcher Operand nun größer war, wenn überhaupt? Dazu müssen zwei Fälle unterschieden werden, je nachdem, ob vorzeichenlose oder vorzeichenbehaftete Integers betrachtet werden:

vorzeichenlose Integer Fall 1: Die Operanden sind vorzeichenlose Zahlen. Dann richten wir unser Augenmerk auf das zero und carry flag, da ja das carry flag einen möglichen Überlauf vorzeichenloser Zahlen signalisiert. Ein solcher Überlauf müsste berücksichtigt werden. In diesem Fall gibt es drei Möglichkeiten:

- Operand1 > Operand2. Dann ist (Operand1 – Operand2) > 0, weshalb weder zero noch carry flag gesetzt sind.
- Operand1 = Operand2. Dann ist (Operand1 – Operand2) = 0, weshalb zero flag gesetzt ist, carry flag gelöscht.
- Operand1 < Operand2. Dann ist (Operand1 – Operand2) < 0, weshalb carry, nicht aber zero flag gesetzt ist. Das gesetzte carry flag signalisiert das »Borgen« aus dem nicht vorhandenen, dem MSB (most significant bit) der Zahl folgenden Bit.

Das bedeutet, dass bei vorzeichenlosen Integers immer dann Operand1 größer als Operand2 ist, wenn das carry flag gelöscht ist. Ist es gesetzt, ist Operand2 größer als Operand1. Sind beide Operanden gleich groß, so ist das zero flag gesetzt.

vorzeichenbehaftete Integer Fall 2: Die Operanden sind vorzeichenbehaftete Zahlen. Dann spielen neben dem zero flag auch noch das overflow und das sign flag eine Rolle. Die Lage ist damit ein wenig komplizierter. Hier unterscheiden wir die Fälle:

- Operand1 > Operand2. Dann ist ZF = 0 sowie SF = OF, wie die weitere, folgende Fallunterscheidung zeigt, die aufgrund der unterschiedlichen Vorzeichenkombinationen erforderlich ist:
  - Operand1 > 0; Operand2 ≥ 0. Dann ist (Operand1 – Operand2) > 0 und es hat kein Unterlauf stattgefunden. Damit ist SF = 0 und OF = 0 und somit SF = OF.
  - Operand1 > 0; Operand2 ≤ 0. Dann ist zwar (Operand1 – Operand2) > 0. Je nach absoluter Größe von Operand1 und Operand2 können aber zwei Situationen auftreten: Das Ergebnis der Addition (Subtraktion eines negativen Wertes ist identisch mit Addition des Absolutwertes!) passt in den Wertebereich. Dann ist SF = 0 und OF = 0. Überschreitet es dagegen den Werte-

bereich, so ist OF = 1 und SF = 1). In jedem Falle ist wiederum SF = OF.

  - Operand1 ≤ 0; Operand2 < 0. Dann ist (Operand1 – Operand2) > 0 (da ja Operand1 > Operand2, s.o.) und es hat kein Überlauf stattgefunden, da der maximal mögliche positive Wert durch die Subtraktion eines kleineren negativen Wertes von einem größeren negativen Wert nicht überschritten werden kann. Damit ist SF = 0 und OF = 0 und ebenfalls SF = OF.

- Operand1 = Operand2. Dann ist (Operand1 – Operand2) = 0 und somit ZF = 1 und SF und OF = 0.
- Operand1 < Operand2. Hier ist wiederum ZF = 0, jedoch ist SF ≠ OF, wie die analoge weitere Fallunterscheidung zeigt:
  - Operand1 ≥ 0; Operand2 > 0. Dann ist (Operand1 – Operand2) < 0 und es hat kein Unterlauf stattgefunden, da das Ergebnis negativ ist, niemals aber den maximalen negativen Wert überschreiten kann. Damit ist SF = 1 und OF = 0 und somit SF ≠ OF.
  - Operand1 < 0; Operand2 ≥ 0. Dann ist (Operand1 – Operand2) < 0, und es muss wiederum unterschieden werden, ob das Ergebnis in den Wertebereich passt. Tut es das, ist SF = 1 und OF = 0 und damit SF ≠ OF. Andernfalls ist OF = 1 und SF = 0 und wiederum SF ≠ OF.
  - Operand1 < 0; Operand2 < 0. Dann ist (Operand1 – Operand2) > 0, ohne dass ein Überlauf stattfinden kann, da, Absolutwerte betrachtet, der negative Operand2 immer um den Absolutbetrag des Operand1 vermindert wird. Somit ist SF = 1, OF = 0 und SF ≠ OF.

Bei vorzeichenbehafteten Integers ist also immer dann Operand1 größer als Operand2, wenn das overflow flag und das sign flag den gleichen Wert haben. Haben sie dagegen entgegengesetzte Werte, ist Operand1 kleiner als Operand2. Auch bei diesen Zahlen zeigt ein gesetztes zero flag an, dass beide Operanden gleich groß sind.



Wer die Fallunterscheidungen von eben mitverfolgt hat, wird eventuell auf eine »Ungereimtheit« gestoßen sein, die mich auch eine Weile überlegen ließ, weil ich das Problem zu sehr von der mathematischen Seite betrachtet habe. Sie betrifft die Fälle bei vorzeichenbehafteten Zahlen,

in denen die Subtraktion des Operand2 vom Operand1 zu Wertüberschreitungen geführt hat, also wenn

1. entweder (Operand1 > 0) > (Operand2 < 0)
2. oder (Operand1 < 0) < (Operand2 > 0) ist.

Die Frage ist, warum in 1. das sign flag gesetzt wird, obwohl doch das temporäre Ergebnis positiv ist (die Subtraktion einer negativen Zahl von einer positiven ist immer positiv!) bzw. in 2. gelöscht wird, obwohl doch das temporäre Ergebnis negativ ist (die Subtraktion einer positiven Zahl von einer negativen ist immer negativ!). Wie gesagt: Diese Ungereimtheit liegt an der allzu mathematischen Betrachtung. Rekapitulieren Sie bitte, was ich bei der Besprechung des EFlags-Registers geschrieben habe: Das sign flag enthält schlicht den Inhalt des MSB, das most significant bit, des Datums. Das bedeutet, dass ein Übertrag in das MSB erfolgt und erfolgen muss, da es ja auch sein könnte, dass die vorliegenden Zahlen vorzeichenlose Integers sind! Ein gesetztes overflow flag zeigt nun das »negierte« Vorzeichen an: Wenn also kein Über-/Unterlauf stattgefunden hat, ist OF immer gelöscht und SF zeigt das Vorzeichen des temporären Ergebnisses korrekt an. Ist es daher positiv (wie in 1.), so ist Operand1 > Operand2, ist es negativ (wie in 2.), so ist Operand1 < Operand2. Ist dagegen OF gesetzt, so hat ein Über-/Unterlauf stattgefunden und das sign flag signalisiert *nicht* den Zustand des Vorzeichens, sondern sein Gegenteil, weil der Übertrag in das MSB erfolgte (MSB = 1: gelöschtes sign flag wurde gesetzt, weil positiver Wertebereich überschritten wurde) oder aus dem MSB erfolgen musste (MSB = 0: gesetztes sign flag wurde gelöscht und damit der negative Wertebereich unterschritten). Somit ist in diesem Fall Operand1 > Operand2, wenn SF gesetzt ist (1.), andernfalls ist Operand1 < Operand2 (2.)

TEST TEST ist das »logische Pendant« zu CMP. Es prüft, ob zwei Bitfelder sich von einander unterscheiden. Auch bei TEST werden als Ergebnis die Statusflags gesetzt, sodass mit Programmverzweigungen auf die bestehende Situation reagiert werden kann. Allerdings ist die Flag-Auswertung lange nicht so kompliziert wie bei CMP.



Auch TEST ist eigentlich eine Verknüpfung, bei der das Resultat verworfen wird. Der Vergleich nutzt eine logische AND-Verknüpfung der beiden Operanden und setzt anhand des temporären Ergebnisses die Statusflags analog zum AND-Befehl. Dann wird das Ergebnis verwor-

fen, ohne in ein Ziel transferiert zu werden. TEST ist damit wie CMP einer der wenigen Befehle ohne Zieloperand.

Operanden

Mit einer Ausnahme sind somit bei TEST alle Operandenkombinationen erlaubt, die auch AND gestattet:

- Vergleich des Akkumulatorinhalts mit einer Maske
  `TEST AL, Const8; TEST AX, Const16; TEST EAX, Const32`
- Vergleich eines Registerinhalts mit einer Maske
  `TEST Reg8, Const8; TEST Reg16, Const16; TEST Reg32, Const32`
- Vergleich eines Speicheroperands mit einer Maske
  `TEST Mem8, Const8; TEST Mem16, Const16; TEST Mem32, Const32`
- Vergleich eines Registerinhalts mit einer vorzeichenerweiterten Byte-Maske
  `TEST Reg16, Const8; TEST Reg32, Const8`
- Vergleich eines Speicheroperands mit einer vorzeichenerweiterten Byte-Maske
  `TEST Mem16, Const8; TEST Mem32, Const8`
- Vergleich eines Registerinhaltes mit einem Registerinhalt
  `TEST Reg8, Reg8; TEST Reg16, Reg16; TEST Reg32, Reg32`
- Vergleich eines Speicheroperanden mit einem Registerinhalt bzw. Vergleich eines Registerinhalts mit einem Speicheroperanden
  `TEST Mem8, Reg8; TEST Mem16, Reg16; TEST Mem32, Reg32`

Die Ausnahme ist *TEST Reg, Mem*. Diese Register-Speicheroperand-Kombination ist unter TEST nicht möglich, was bei genauerem Hinsehen auch gerechtfertigt ist: Vom Ergebnis her ist *TEST Reg, Mem* das Gleiche wie *TEST Mem, Reg*, da die AND-Verknüpfung in beiden Fällen die gleichen Bitstellungen erzeugt (AND und somit auch TEST sind von der logischen Operation her kommutativ!). Die beiden »AND«-Fälle würden sich also lediglich darin unterscheiden, in welches Ziel das Ergebnis gespeichert wird: *Reg* bzw. *Mem*. Da aber, anders als bei AND, das Ergebnis bei TEST nach dem Setzen der Statusflags verworfen wird, sind beide Fälle gleich und daher einer redundant, weshalb auf *TEST Reg, Mem* verzichtet wird.

Statusflags

Analog AND werden bei TEST die Flags gesetzt: OF und CF sind explizit gelöscht, da es weder einen vorzeichenlosen noch einen vorzeichenbehafteten Über- oder Unterlauf geben kann. AF gilt als undefiniert, lässt also keine Auswertung zu. SF dient hier nicht als Signal für die Stellung eines Vorzeichens, sondern ist wie bei logischen Operationen

gewohnt je nach Stellung des Bits 31 (DoubleWords), 15 (Words) oder 7 (Bytes) gesetzt; und ZF ist gesetzt, wenn alle Bits gelöscht sind, das Bitfeld somit unbesetzt. PF ist nach TEST wie üblich gesetzt, wenn in Bits 7 bis 0 eine gerade Anzahl gesetzter Bits vorgefunden wird.

1.1.4 Bitorientierte Operationen

Neben den »logischen« Operationen, die auf Bitfelder wirken, gibt es noch weitere bitorientierte Prozessorbefehle. Sie dienen zum einen dazu, einzelne Bits in den Bitfeldern gezielt anzusprechen (BT, BTS, BTR, BTC) oder zu suchen (BSF, BSR), zum anderen dazu, die Reihenfolge der Bits in den Bitfeldern zu ändern (SHL, SHR, SAL, SAR, SHLD, SHRD, ROL, ROR, RCL, RCR).



Wie bereits im Abschnitt über die Logischen Operationen geäußert, können auch Zahlen als Bitfelder aufgefasst werden. So lässt sich die *Zahl* $4711_d = 1267_h$ binär darstellen als

$1 \cdot 2^{12} + 0 \cdot 2^{11} + 0 \cdot 2^{10} + 1 \cdot 2^9 + 0 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$.

In dieser Summe fallender 2er-Potenzen lassen sich die Koeffizienten der 2er-Potenzen als eigenständige, von einander unabhängige Ziffern interpretieren, die einzeln und von einander unabhängig manipuliert werden können. Sie bilden somit ein Bitfeld. Dass diese Sichtweise durchaus sinnvoll sein kann, haben wir beim »Missbrauch« des AND- bzw. OR-Befehls bereits gesehen. Auch in diesem Kapitel werden wir Befehle kennen lernen, die für *Zahlen* »missbraucht« werden können, ja die sogar eigens dafür geschaffen wurden.

Zu den grundlegenden bitorientierten Befehlen gehören die »Verschiebe«-Befehle. Dies sind Befehle, bei denen die Bits um eine gewisse Anzahl von Stellen innerhalb des Bitfeldes nach links oder rechts verschoben werden. Je nachdem, wie das Schicksal der am einen Ende das Bitfeld verlassenden Bits aussieht, kennt man verschiedene Arten von Verschiebebefehlen: die »Shift«-Befehle, bei denen die »aus dem Bitfeld herausgeschobenen« Bits verworfen werden, und die »Rotationsbefehle«, bei denen die am einen Ende herausgeschobenen Bits das Bitfeld über das andere Ende wieder betreten, also im Bitfeld »rotieren«.

Die folgenden, erläuternden Abbildungen gehen von einer Situation aus, die in Abbildung 1.8 dargestellt ist.

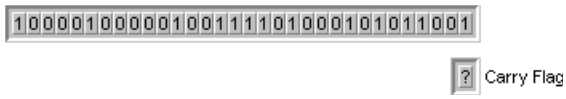


Abbildung 1.8: Speicherabbild eines Bitfeldes als Ausgangssituation vor einem Bit-Schiebebefehl

Im ersten Quelloperanden der Befehle liege ein Bitfeld vor, in dem die Bits 31, 26, 20, 17 bis 14, 12, 8, 6, 4, 3 und 0 gesetzt sind. Der Inhalt des carry flags sei unbestimmt.

SHL
SHR

Shift logical left, SHL, und *shift logical right*, SHR, sind zwei Vertreter der ersten Kategorie. Bei SHL erfolgt das Verschieben der Bits um eine anzugebende Anzahl von Positionen nach links, sodass am linken Ende die gleiche Anzahl Bits das Bitfeld verlassen und verworfen werden. Bei SHR verlassen diese Bits das Bitfeld rechts und werden ebenfalls verworfen.

Was aber passiert mit den frei gewordenen Positionen rechts (bei SHL) und links (bei SHR)? In der Grundversion der Shift-Befehle, SHL und SHR, werden die freien Plätze mit Nullen aufgefüllt. Basta! Abbildung 1.9 zeigt das Ergebnis nach einem Shift um 5 Positionen nach rechts (oben) bzw. um 3 Positionen nach links (unten). Die grau dargestellten Ziffern repräsentieren die von der »Nullhalde« aufgefüllten Ziffern.

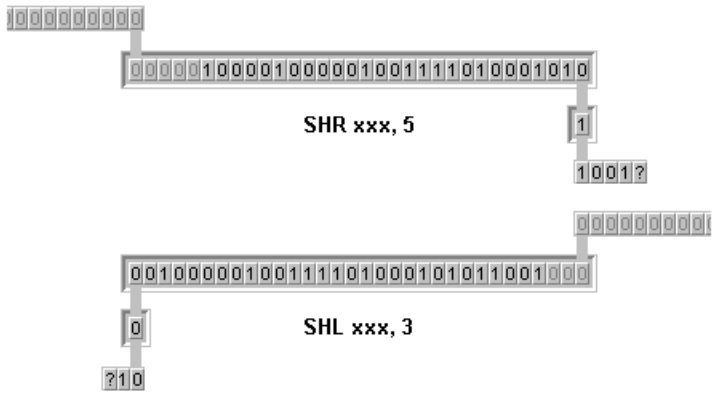


Abbildung 1.9: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »einfachem« Verschieben nach rechts (SHR; oben) bzw. links (SHL; unten)

SHLD
SHRD

Bei SHLD, *double precision shift left*, und SHRD, *double precision shift right*, gibt es eine Quelle, aus der die nachrückenden Bits rekrutiert werden. Diese Quelle ist ein weiteres Bitfeld, das als zweiter Operand übergeben wird. Das bedeutet, die am einen Ende des ersten Bitfeldes auftretenden Lücken werden mit Bits aus dem anderen Ende des zweiten Bitfeldes gefüllt. Abbildung 1.10 demonstriert das.

In der Abbildung herrscht jeweils die gleiche Situation wie in Abbildung 1.9, jedoch werden hier die Bits aus einem weiteren Bitfeld (»yyy«) gewonnen, in dem in diesem Beispiel vor der Operation jedes gerade Bit gesetzt war. Wie man erkennt, »saugen« die frei werdenden Positionen im Zieloperanden die Bits aus dem zweiten Quelloperanden, was dazu führt, dass auch dort die Bits verschoben werden. Dies erfolgt allerdings nur formal, da sich der Inhalt des zweiten Quelloperanden nicht ändert.

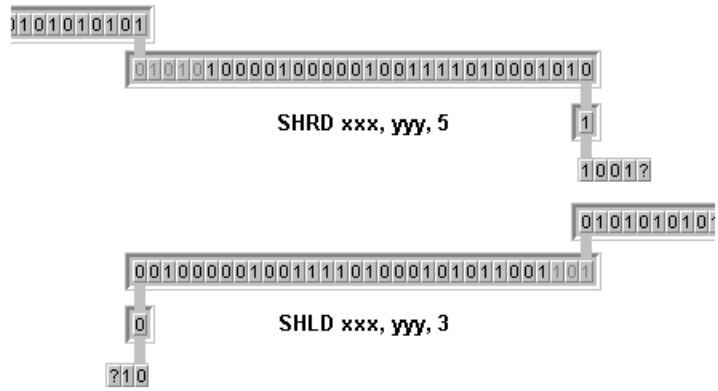


Abbildung 1.10: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »doppelt präzisem« Verschieben nach rechts (SHDR; oben) bzw. links (SHDL; unten)

SAL
SAR

Bei SAR, *shift arithmetic right*, dagegen ist Quelle das most significant bit, also Bit 7 bei Bytes, Bit 15 bei Words und Bit 31 bei DoubleWords. Das bedeutet, die links frei werdenden Stellen werden alle mit dem Bit aufgefüllt, das vor der Verschieberei einmal das MSB (Vorzeichen!) war. Nützlich und Grund für seine Existenz ist bei SAR daher, dass eine automatische sign extension durchgeführt wird, wenn vorzeichenbehaftete Zahlen nach rechts verschoben werden. Hier haben wir einen solchen »arithmetischen« Bit-orientierten Befehl. Abbildung 1.11 zeigt ein Beispiel.

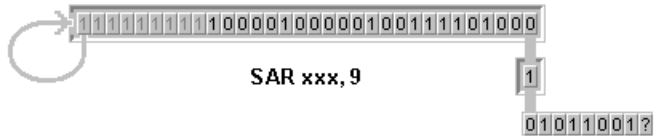


Abbildung 1.11: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »arithmetischem« Verschieben nach rechts (SAR)

Und im Falle von Verschiebungen nach links? Bleibt das Vorzeichen (MSB) der Zahlen bei SAL, *shift arithmetic left*, ebenfalls erhalten? Leider nein! SAL ist nur ein Alias für SHL und wird durch die Assembler in die gleiche Befehlssequenz übersetzt.

Alle sechs Shift-Befehle, also SHL, SHR, SAL, SAR, SHLD und SHRD, involvieren das carry flag. So wird bei allen Befehlen das letzte Bit, das aus dem Bitfeld geschoben wird, in das CF kopiert.



ROL
ROR

Lässt man dagegen die am einen Ende austretenden Bits am anderen Ende wieder eintreten, also die Bits »nur rotieren«, so hat man mit den Befehlen *rotate left*, ROL bzw. *rotate right*, ROR, die zweite Kategorie an Verschiebebefehlen. ROL und ROR involvieren das CF wie die Shift-Befehle, indem sie eine Kopie des zuletzt aus dem Bitfeld rotierten Bits in CF ablegen. (Bei ROL ist somit CF eine Kopie des LSB, da das LSB das MSB vor dem letzten Rotationszyklus war, somit das zuletzt nach links herausgeschobene und rechts aufgenommene Bit ist. Analog ist bei ROR das CF eine Kopie des MSB.)

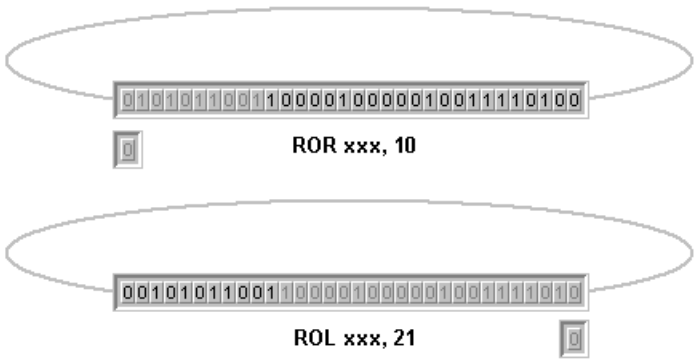


Abbildung 1.12: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »einfachem« Rotieren nach rechts (ROR) bzw. links (ROL)

Abbildung 1.12 zeigt die beiden Rotationsbefehle. Im oberen Teil erfolgt eine Rotation nach rechts um 10 Positionen, im unteren eine nach links um 21 Positionen. Die jeweils grau dargestellten Bits sind die an einem Ende ausgetretenen und am anderen wieder eingetretenen Bits.

RCR
RCL

Diese »Rotationsbefehle« gibt es auch in einer Version, die das carry flag direkt in die Rotation mit einbezieht und nicht nur als Kopie des zuletzt rotierten Bits auffasst: RCL, *rotate left with carry*, und RCR, *rotate right with carry*, rotieren »über« das CF (siehe Abbildung 1.13). In diesem Fall muss man sich das carry flag als imaginäres Bit 32 (RCR) bzw. -1 (RCL) vorstellen (hier werden DoubleWords zugrunde gelegt, Analoges gilt natürlich für Words und Bytes!). Das bedeutet: Das erste in eine frei werdende Position hineingeschobene Bit stammt aus dem carry flag, das letzte herausgeschobene Bit landet im carry flag.

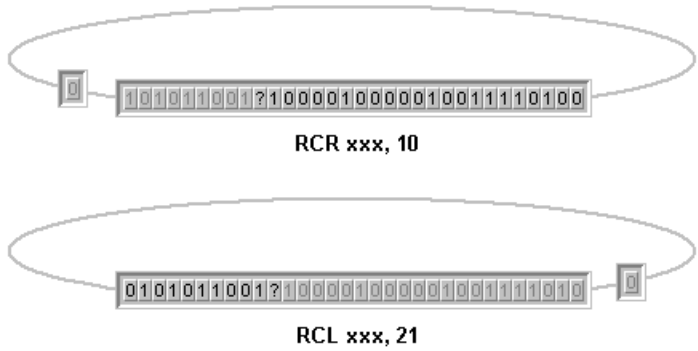


Abbildung 1.13: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach Verschieben »über carry« nach rechts (RCL) bzw. links (RCL)

Das Fragezeichen zeigt die Position, an der der Inhalt des unbestimmten carry flag »eingereiht« wurde. Es ist jeweils die erste »aufgefüllte« Position. Das CF selbst beinhaltet jeweils das letzte aus dem Bitfeld geschobene Bit.



Da bei den Befehlen RCR und RCL das carry flag das erste nachrückende Bit enthält, ist es notwendig, ihm vor dem Aufruf von RCR oder RCL einen definierten Inhalt zu geben (in den Abbildungen somit eine »0« oder eine »1« zuzuordnen). Hierzu gibt es Befehle, die es explizit setzen, löschen oder gezielt verändern können. Diese Befehle werden wir im Abschnitt über »Instruktionen zur gezielten Veränderung des Flagregisters« weiter unten kennen lernen.



Die Shift-Befehle werden häufig für einfache und schnelle Multiplikationen bzw. Divisionen mit Multiplikatoren bzw. Divisoren verwendet, die eine Potenz zur Basis 2 sind. So ist die Verschiebung des Bitfeldes um eine Position nach rechts de facto eine Division durch $2^1 = 2$, während die Verschiebung um 4 Positionen nach links eine Multiplikation mit $2^4 = 16$ ist. Mit SAR lässt sich so eine vorzeichenbehaftete Division vom Typ IDIV erreichen, während SHR eine DIV-ähnliche Division durchführt. SAL und SHL entsprechen dem Befehl MUL. Ein Pendant für IMUL gibt es nicht.



Eine Division mittels SAR führt nicht zum gleichen Ergebnis wie IDIV! Treten bei der Division mittels IDIV Divisionsreste auf, so werden sie (zumindest was das Ergebnis als Integer betrifft) verworfen, es erfolgt somit eine Rundung »in Richtung Null«. Eine Division durch Bitverschiebung mittels SAR dagegen rundet »in Richtung negative Unendlichkeit«. Beispiel: -9 IDIV 4 ergibt -2 (-2.25 Richtung 0 gerundet). -9 SAR 2 ergibt -3 (-2.25 Richtung -∞ gerundet)! (Wer's nicht glaubt – cave: 2er-Komplement: $-9_d = F7_h = 11110111_b$; zwei Bits nach rechts mit sign extension: $11111101_b = FD_h = -3_d$). Diese Unterschiede treten jedoch nur bei IDIV und SAR mit negativen Zahlen auf. Für positive Zahlen oder bei DIV und SHR sind die Ergebnisse gleich, da in diesem Fall bei DIV eine Rundung »in Richtung 0« und bei SHR eine Rundung »in Richtung -∞« und somit jeweils in die gleiche Richtung erfolgt!

Operanden

Die Shift-Befehle SHL, SHR, SAL und SAR sowie die Rotationsbefehle ROL, ROR, RCL und RCR verwenden die gleichen Operandenstrukturen. Die Bitfelder werden immer als erster Operand übergeben, sind somit erster Quell- und Zieloperand. Sie können sowohl in Registern als auch an Speicherstellen stehen und 8, 16 oder 32 Bits umfassen. Als zweiten Quelloperanden erwarten die Befehle eine Zahl, die die Anzahl der zu verschiebenden Positionen angibt. Dies kann eine Konstante sein, allerdings kann sie auch über ein Register angegeben werden. Dieses Register ist immer und muss immer das 8-Bit-Register CL sein. Somit ergeben sich folgende möglichen Befehlsfolgen (XXX steht für SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR):

- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das Bitfeld in einem Register vorliegt:
  ```
  XXX Reg8, Const8; XXX Reg16, Const8; XXX Reg32, Const8
  ```
- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das Bitfeld in einer Speicherstelle vorliegt:
  ```
  XXX Mem8, Const8; XXX Mem16, Const8; XXX Mem32, Const8
  ```

- Indirekte Angabe der Anzahl zu verschiebender Positionen via CL, wobei das Bitfeld in einem Register vorliegt:
  `XXX Reg8, CL; XXX Reg16, CL; XXX Reg32, CL`
- Indirekte Angabe der Anzahl zu verschiebender Positionen via CL, wobei das Bitfeld in einer Speicherstelle vorliegt:
  `XXX Mem8, CL; XXX Mem16, CL; XXX Mem32, CL`

Die Befehle SHLD und SHRD benötigen, wie bereits angedeutet, drei Operanden. Der erste Quell- und damit auch Zieloperand ist wiederum das Bitfeld, das verändert werden soll. Hier kommen wiederum Register oder Speicherstellen als Ort für das Bitfeld in Frage, allerdings sind Byte-Operanden nicht möglich. Der zweite Quelloperand ist immer ein Register. In ihm ist das Bitfeld angegeben, das zur »Auffüllung« dienen soll – weshalb er immer die gleiche Größe wie der erste Quelloperand haben muss. Der Inhalt dieses Registers wird nicht verändert! Dritter Quelloperand ist eine Konstante wie bei den übrigen Verschiebebefehlen, die die Anzahl der zu shiftenden Positionen angibt. Somit sind folgende Befehlsfolgen möglich (XXX steht für SHLD oder SHRD):

- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld in einem Register vorliegt:
  `XXX Reg16, Reg16, Const8; XXX Reg32, Reg32, Const8`
- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld an einer Speicherstelle vorliegt:
  `XXX Mem16, Reg16, Const8; XXX Mem32, Reg32, Const8`
- Indirekte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld in einem Register vorliegt:
  `XXX Reg16, Reg16, CL; XXX Reg32, Reg32, CL`
- Indirekte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld an einer Speicherstelle vorliegt:
  `XXX Mem16, Reg16, CL; XXX Mem32, Reg32, CL`



Der Wert, der zur Bestimmung der zu verschiebenden Positionen benutzt wird, ist immer ein 8-Bit-Wert. Er wird mit der Maske $1F UND-verknüpft, um nur die fünf niedrigerwertigen Bits zuzulassen. Dies beschränkt den Wertebereich des »Positionszählers« auf [0, 31]. Damit ist gewährleistet, dass maximal um 31 Positionen verschoben werden kann. Diese Reduktion wird im Falle der Rotationsbefehle ggf. noch weiter getrieben: Werden 16-Bit-Operanden verwendet, so wird der auf den Bereich [0,31] skalierte Wert nochmals modulo 1/ und bei 8-Bit-Operanden modulo 9 genommen. Dies reduziert den Wertebereich auf

die jeweilige Operandengröße ([0,16] bei Word-Operanden, [0,8] bei Byte-Operanden) und verhindert, dass mehrere unnötige Rotationszyklen mit identischem Ergebnis durchgeführt werden. ACHTUNG: Bei Word- und Byte-Operanden ist es möglich, bei den Befehlen RCL und RCR um eine Position mehr zu rotieren, als der Operand Bits hat! Grund: Das carry flag hat hier die Funktion eines »zusätzlichen« Bits, sodass formal Words hier 9 Bits breit sind und Bytes 8 – zumindest, was die Rotation angeht.



Ungeachtet der Reduktion der Anzahl zu verschiebender Bits auf den maximalen Wertebereich [0,31] bei den Shift-Befehlen kann es (nur bei diesen!) vorkommen, dass er dennoch zu groß ist: Wenn 16- oder 8-Bit-Operanden zum Einsatz kommen. In diesem Falle sind sowohl der Inhalt des Zieloperanden wie auch alle Statusflags undefiniert.

Statusflags

Ist die im zweiten (bzw. bei SHLD und SHRD: dritten) Operanden übergebene Zahl zu verschiebender Positionen Null, so werden keine Flags verändert.

Bei allen Verschiebebefehlen nach links (ROL, RCL, SHL, SAL und SHLD) um eine Position zeigt das overflow flag einen »Vorzeichenwechsel« durch die Verschiebung an. Dies erfolgt, indem vor der Verschiebung das MSB und das benachbarte, niedrigerwertige Bit XOR-verknüpft werden und das Ergebnis im OF abgelegt wird. Ein Beispiel: In einem 32-Bit-Feld ist das Bit 31 das »Vorzeichen«, wenn man die Bits als Koeffizienten einer LongInt auffasst. Durch die Verschiebung nach links um eine Position (Multiplikation mit 2) wird dieses MSB herausgeschoben und Bit 30, das ursprünglich benachbarte Bit, avanciert zum »neuen« Vorzeichenbit. Haben somit Bit 31 und Bit 30 des unveränderten Bitfeldes den gleichen Zustand, ändert sich das »Vorzeichen« durch die Verschiebung nicht, andernfalls sehr wohl. Die XOR-Verknüpfung von Bit 31 und Bit 30 erzeugt das korrekte Ergebnis: OF = Bit_{31} XOR Bit_{30}.

(In manchen Dokumentationen ist zu lesen, dass für das OF das MSB und das CF geXORt werden! Das ist zweifellos richtig, wenn man den Zustand nach der Operation zur Definition heranzieht. Da das CF in diesem Fall das ursprüngliche MSB – und somit das ursprüngliche Bit 31 im Beispiel – enthält und das neue MSB das ursprünglich dem MSB benachbarte Bit – Bit 30 im Beispiel –, sind beide Aussagen identisch. Ich selbst bevorzuge die Definition anhand des Ausgangszustands, da

mir auf diese Weise der Effekt, »Vorzeichenwechsel«, deutlicher nachvollziehbar erscheint, als wenn man ein MSB mit einem CF verknüpft!)

Auch bei den Verschiebebefehlen um eine Position nach rechts (ROR, RCR, SHR, SAR und SHRD) zeigt das OF nach der Operation einen »Vorzeichenwechsel« an, indem das MSB und das niedrigerwertige, benachbarte Bit XOR-verknüpft werden (also z.B. Bit 31 und 30 in einem 32-Bit-Feld; ACHTUNG: hier wird tatsächlich der Zustand *nach* der Operation betrachtet, also wenn das »neue Vorzeichen« bereits im MSB vorliegt und das »alte« rechts daneben; hier würde die Erklärung mit dem Zustand *vor* der Operation weniger anschaulich sein!). Dieser »Vorzeichenwechsel« tritt bei SAR *niemals* ein, da ja das »alte« MSB durch die Operation in das »neue« MSB kopiert wird, die XOR-Verknüpfung also »0« ergibt. Dies ist absolut korrekt, da ja SAR eine »Vorzeichenerweiterung nach Division« durchführt, das Vorzeichen also gleich bleibt. Bei SHR dagegen hat OF immer den Wert des »alten Vorzeichens (=MSB)«, da immer eine »0« nachgeschoben wird und eine XOR-Verknüpfung eines beliebigen Zustandes mit »0« immer den Zustand selbst ergibt. (0 XOR 0 = 0 ➞ OF: »positiv« bleibt »positiv«, kein Wechsel; 1 XOR 0 = 1 ➞ OF: »negativ« wird »positiv«, Wechsel!).

Zusammengefasst heißt das: Das overflow flag zeigt nach den Verschiebebefehlen um eine Position einen ggf. aufgetretenen »Vorzeichenwechsel« (bei SAR also nie) an, bei Verschiebungen um mehr als eine Position ist OF undefiniert.

Die Flags SF, ZF, AF und PF bleiben bei allen Verschiebebefehlen außer SHLD und SHRD unverändert. Bei SHRD und SHLD werden SF, ZF und PF anhand des Ergebnisses gesetzt, AF ist undefiniert.

Das carry flag enthält grundsätzlich eine Kopie des zuletzt aus dem Bitfeld geschobenen Bits (im Falle von RCR und RCR enthält es *das* zuletzt aus dem Bitfeld geschobene Bit).

BT
BTS
BTR
BTC

Die BTx-Familie ist eine Gruppe von Bit-orientierten Befehlen, die den Zustand eines bestimmten Bits in einem Bitfeld prüfen. Der Zustand des überprüften Bits wird im carry flag gespeichert, sodass nach dem BTx-Befehl unmittelbar eine Programmverzweigung mittels bedingtem Sprung oder anderen bedingten Operationen (SETcc) erfolgen kann. Anschließend wird entweder gar nichts mehr unternommen (BT, *bit test*), das eben geprüfte Bit gesetzt (BTS, *bit test and set*), gelöscht (BTR, *bit test and reset*) oder »umgedreht« (BTC, *bit test and complement*).

Das Bitfeld kann entweder Word- bzw. DoubleWord-Größe besitzen und damit vollständig in einem Register abbildbar sein. Es ist jedoch auch möglich, »Bitstrings« zu verwenden, also Strukturen, die erheblich größer als eine Word-/DoubleWord-Variable sind. In diesem Fall müssen diese Strukturen jedoch eine Größe aufweisen, die ein ganzzahliges Vielfaches von Words bzw. DoubleWords ist.

Operanden

Alle BTx-Befehle erwarten als ersten Quelloperanden (und damit auch Zieloperanden) das Bitfeld. Dies kann entweder direkt in einem Register oder indirekt in einem Speicheroperanden enthalten sein. Der zweite Quelloperand gibt dann an, welches Bit in diesem Feld verwendet werden soll. Das kann wiederum entweder direkt durch Angabe einer Konstanten erfolgen oder indirekt über ein Register. Somit sind folgende Operandenkombinationen möglich:

- Direkte Angabe des zu prüfenden Bits durch eine Konstante, das Bitfeld liegt direkt in einem Register vor:
  `BTx Reg16, Const8; BTx Reg32, Const8`
- Direkte Angabe des zu prüfenden Bits durch eine Konstante, das Bitfeld liegt indirekt in einem Speicheroperanden vor:
  `BTx Mem16, Const8; BTx Mem32, Const8`
- Indirekte Angabe des zu prüfenden Bits durch eine Register-Konstante, das Bitfeld liegt direkt in einem Register vor:
  `BTx Reg16, Reg16; BTx Reg32, Reg32`
- Indirekte Angabe des zu prüfenden Bits durch eine Register-Konstante, das Bitfeld liegt indirekt in einem Speicheroperanden vor:
  `BTx Mem16, Reg16; BTx Mem32, Reg32`

Wird ein Register als erster Quelloperand verwendet, so enthält dieses Register bereits das gesamte Bitfeld. Daher können in diesem Fall nur die Bits 0 bis 15 (bei Word-Registern) bzw. 0 bis 31 (bei DoubleWord-Registern) angesprochen werden. Die BTx-Befehle würdigen diese Tatsache, indem sie als zu prüfende Bitposition den Modulus 16 (Word-Register) bzw. Modulus 32 (DoubleWord-Register) des im zweiten Quelloperanden (Const8, Reg16, Reg32) übergebenen Wertes als gewünschte Bitposition nehmen.

Wird dagegen als erster Quelloperand eine Speicherstelle verwendet, so wird dem Befehl die Adresse auf ein Word bzw. DoubleWord übergeben. Diese Adresse kann jedoch nicht nur als Zeiger auf ein Word/DoubleWord aufgefasst werden, sonder als Adresse auf das erste Word/DoubleWord eines Feldes aus Words/DoubleWords, in dem wei-

tere Bits verzeichnet sind. Daher wird sie als »Bitbasis« (*bit base*) bezeichnet. Das aber bedeutet, dass erheblich mehr als 16 bzw. 32 Bits angesprochen werden können. Die im zweiten Quelloperanden übergebene Bitposition wird daher nicht mehr auf den Wertebereich eines Words bzw. DoubleWords beschränkt. Die bedeutet aber, dass sie nun umgerechnet werden muss in einen als Offset (»Bitoffset«, *bit offset*) bezeichneten Wert, der angibt, im wievielten Word/DoubleWord beginnend mit der bit base das zu prüfende Bit steht und an welcher Position in diesem Word/DoubleWord es sich befindet.

Mit dem im zweiten Operanden über ein Register übergebenen Wert lassen sich 2^{16} = 65.536 Bits (Word-Register) und 2^{32} = 4.294.967.296 Bits (DoubleWord-Register) adressieren. Da die BTx-Befehle den Inhalt dieser Register als vorzeichenbehaftete Integer interpretieren, lassen sich somit Bitpositionen von -32.768 bis +32.767 (Word-Register) bzw. -2.147.483.648 bis +2.147.483.647 angeben (was bedeutet, dass Bits »vor« und »hinter« der bit base angesprochen werden können). Die bereits angesprochene Berechnung des Offset erfolgt nun bei Word-Operanden durch

$$\text{Offset} := 2 \cdot (\text{Operand div } 16); \quad \text{Position} := \text{Operand mod } 16.$$

Die Integer-Division des im zweiten Operanden übergebenen Wertes durch 16 gibt an, in welchem Word das gewünschte Bit verzeichnet ist (Bits 0 bis 15 in Word 0, Bits 16 bis 31 in Word 1 etc.). Dieser Wert mit 2 multipliziert gibt an, wie viele Bytes zur bit base addiert werden müssen, um das Word zu identifizieren, das das gewünschte Bit enthält. Der Modulus 16 des gewünschten Bits, also quasi der Rest, der bei der Offset-Berechnung mittels DIV übrig bleibt, gibt dann die Position des Bits in diesem Word an. Bitte beachten Sie, dass der Offset je nach übergebenem Wert positiv oder negativ sein kann, die Position jedoch nicht!

Analog erfolgt die Berechnung bei DoubleWord-Operanden:

$$\text{Offset} := 4 \cdot (\text{Operand div } 32); \quad \text{Position} := \text{Operand mod } 32$$

Nach dieser Berechnung addieren die BTx-Befehle nun den Offset zur bit base und laden das so identifizierte Word/DoubleWord in die internen Arbeitsregister. Dann wird das Bit an der Stelle *Position* geprüft und ggf. verändert, bevor das Resultat ggf. wieder zurückgeschrieben wird.



Zweierlei gibt es noch zu berücksichtigen. Erstens: Bei Verwendung der Byte-Konstanten als zweitem Operator erfolgt diese Adressberechnung nicht! Vielmehr wird das an der im ersten Operanden übergebenen Stel-

le stehende Word/DoubleWord geladen und der Modulus 16 bzw. 32 der Konstante als Bitposition verwendet.

Zweitens: Diese Art der Bit-Identifizierung ist gefährlich! Nachdem bei Verwendung von DoubleWord-Operanden 4.294.967.296 Bits angesprochen werden können, sind Bitfeld-Strukturen bis 536.870.912 Byte = 512 MByte möglich (8 Bits pro Byte)! Das bedeutet, dass es sehr leicht zu Schutzverletzungen kommen kann, falls man nicht erheblich aufpasst! Diese können darin begründet sein, dass z.B. die Datensegmente nicht groß genug sind, negative Offsets verwendet werden, ohne die bit base entsprechend anzupassen, oder sog. »memory-mapped I/O register« angesprochen werden: Schnell ist ein falscher Wert in das Register geschrieben, vor allem, wenn er berechnet wird! Intel empfiehlt daher, die Adressberechnung nach den obigen Formeln selbst vorzunehmen, das entsprechende Word/DoubleWord mittels MOV in ein Register zu laden und dann die »Register-Version« der BTx-Befehle zu nutzen.



Manche Assembler erlauben, dass bei der indirekten Bitprüfung (Speicherstellen) mittels direkter Angabe des Prüfbits (Const8) auch Bitpositionen größer 15 (Word-Speicherstellen) bzw. größer 31 (DoubleWord-Speicherstellen) angegeben werden können. Dazu berechnet der Assembler anhand der oben angeführten Formeln ein Offset und eine Position aus der Konstanten. Die Position ist dann wieder im Bereich 0 bis 15 bzw. 0 bis 32 und wird zur »neuen« Const8. Der Offset wird zur bit base addiert. Dies ist gleichbedeutend mit einer automatischen Verschiebung der bit base um den Offset; die neue bit base ist dann die neue Adresse des Speicheroperanden. Das alles ist für den Programmierer vollkommen transparent. Man kann dieses Verhalten des Assemblers nur anhand der Tatsache erkennen, dass sich jeweils Speicheradresse und Wert der Const8 im Assembler-Quelltext und im Assemblat unterscheiden.

Statusflags

Das carry flag erhält den Zustand des geprüften Bits. Alle anderen Flags sind undefiniert, sollten also nicht ausgewertet werden. Nach Setzen des carry flag löscht BTR das geprüfte Bit, BTS setzt es und BTC negiert es.

BSF
BSR

BSF und BSR sind zwei Operationen, die in einem Bitfeld das erste gesetzt Bit suchen. BSF, *bit scan forward*, beginnt hierbei am LSB, dem *least significant bit*, an Position 0 im Bitfeld und sucht in Richtung MSB, dem *most significant bit* an Position 15 (Word-Operanden) bzw. 31 (DoubleWord-Operanden). BSR, *bit scan reverse*, sucht umgekehrt beginnend

mit dem MSB in Richtung LSB. Die Suche bricht ab, wenn entweder ein gesetztes Bit gefunden oder das gesamte Bitfeld durchsucht wurde. Wurde ein gesetztes Bit gefunden, wird dessen Position in den Zieloperanden eingetragen. Andernfalls gilt der Inhalt des Zieloperands als undefiniert.

Operanden Die Befehle sind zwei der Ausnahmen, in denen Operand #1 nicht gleichzeitig Quell- und Zieloperand sind, sondern nur Zieloperand. Hierbei kann es sich nur um ein Register handeln. Er nimmt die Position auf, an der das erste gesetzte Bit gefunden wurde. Der erste und einzige Quelloperand ist somit Operand #2. Über ihn kann das zu prüfende Bitfeld entweder direkt via Allzweckregister oder indirekt über eine Speicherstelle übergeben werden. Somit sind folgende Instruktionen möglich:

- Direkte Angabe des Bitfeldes über ein Register
  `BSx Reg16, Reg16; BSx Reg32, Reg32`
- Indirekte Angabe des Bitfeldes über einen Speicheroperanden
  `BSx Reg16, Mem16; BSx Reg32, Mem32`

Statusflags Wenn der Wert des Bitfeldes »0«, d.h. kein Bit gesetzt ist, wird das zero flag gesetzt, andernfalls gelöscht. Alle anderen Statusflags gelten als undefiniert, können also nicht ausgewertet werden.

1.1.5 Operationen zum Datenaustausch



Da in diesem Abschnitt die Kommunikation mit dem Speicher beschrieben wird, empfiehlt es sich, die Kapitel »Speicherverwaltung« auf Seite 394 und »Adress- und Operandengrößen« auf Seite 765 sowie »Stack« auf Seite 385 und »Ports« auf Seite 827 durchgelesen zu haben.

MOV Einer der wohl wichtigsten Befehle des Befehlssatzes überhaupt ist der MOV-Befehl. Er ist dafür verantwortlich, ein Datum von einer Stelle zu einer anderen zu bewegen. Das Mnemonic MOV, *move data*, ist allerdings etwas missverständlich: Bewegt wird nur eine Kopie des Datums, das Datums selbst bleibt an seinem Ursprungsort unverändert erhalten.

Operanden Gemäß seiner Bedeutung akzeptiert der Befehl sehr viele Operandentypen und -kombinationen.

- Kopieren einer Konstanten in ein Allzweckregister
  `MOV Reg8, Const8; MOV Reg16, Const16; MOV Reg32, Const32`

- Kopieren einer Konstanten an eine Speicherstelle
  `MOV Mem8, Const8; MOV Mem16, Const16; MOV Mem32, Const32`
- Kopieren eines Allzweckregisterinhaltes in ein Allzweckregister
  `MOV Reg8, Reg8; MOV Reg16, Reg16; MOV Reg32, Reg32`
- Kopieren des Inhalts einer Speicherstelle in ein Allzweckregister
  `MOV Reg8, Mem8; MOV Reg16, Mem16; MOV Reg32, Mem32`
- Kopieren eines Allzweckregisterinhaltes an eine Speicherstelle
  `MOV Mem8, Reg8; MOV Mem16, Reg16; MOV Mem32, Reg32`
- Kopieren eines Speicherinhalts in den Akkumulator
  `MOV AL, Mem8; MOV AX, Mem16; MOV EAX, Mem32`
- Kopieren eines Speicherinhalts in den Akkumulator
  `MOV Mem8, AL; MOV Mem16, AX; MOV Mem32, EAX`
- Kopieren eines Allzweckregisterinhaltes in ein Segmentregister
  `MOV SReg, Reg16;`
- Kopieren eines Speicherinhaltes in ein Segmentregister
  `MOV SReg, Mem16;`
- Kopieren eines Segmentregisterinhaltes in ein Allzweckregister
  `MOV Reg16, SReg;`
- Kopieren eines Segmentregisterinhaltes an eine Speicherstelle
  `MOV Mem16, SReg;`



Es gibt noch Erweiterungen des MOV-Befehls, die den Zugriff auf die Kontroll- und Debug-Register des Prozessors ermöglichen. Dieser Zugriff ist jedoch nur unter Privilegstufe 0 möglich, sodass die entsprechenden Erweiterungen als privilegierte Befehle gelten und im Abschnitt »Verwaltungsbefehle« besprochen werden.



Der MOV-Befehl kann *nicht* dazu benutzt werden, Daten in das CS-Segmentregister zu schreiben. Der Versuch, dies zu tun, erzeugt eine invalid opcode exception (#UD). Das CS-Register kann nur im Rahmen von JMP, CALL, RET und IRET-Befehlen von außen verändert werden.



Falls mit dem MOV-Befehl ein Selektor in ein Segment-Register geladen werden soll, muss dieser valide sein. Im protected mode heißt das, dass er auf einen gültigen Eintrag in der global descriptor table (GDT) oder der aktuellen local descriptor table (LDT) zeigen muss. Der Prozessor kopiert in diesem Fall die Daten aus dem Deskriptor in den nicht zugänglichen Cache des Segmentregisters und führt die erforderlichen

Validierungen durch. Nullselektoren dürfen in Segmentregister geladen werden. Das bedeutet, es wird keine Exception ausgelöst, wenn das Segmentregister beschrieben wird. Der nächste Zugriff auf das so selektierte »Nullsegment« jedoch erzeugt eine general protection exception (#GP).



Gemäß der im Anhang unter »Standard-Adress- und Operandengrößen« genannten Bedingungen erzeugen Assembler in 32-Bit-Umgebungen Befehlssequenzen mit einem in diesem Fall überflüssigen operand size override prefix, wenn Daten zwischen einem Segmentregister und einem Allzweckregister ausgetauscht werden, da hier Nicht-Standard-Daten (16-Bit-Word in 32-Bit-Umgebung) Verwendung finden. Dies ist zwar kein Problem und der Prozessor arbeitet absolut korrekt; dennoch stellt das Präfix hier ein cycle penalty dar, das immerhin einen Takt kostet. Man kann dies verhindern, indem man als Allzweckregister ein 32-Bit-Register angibt. Die meisten Assembler gehen dann von Standarddaten aus und kopieren die unteren 16 Bits aus dem Allzweckregister in das Segmentregister oder umgekehrt. Prozessoren vor dem Pentium Pro lassen bei dem Kopieren des Segmentregister-Inhalts in das niedrigerwertige Word des Allzweckregisters den höherwertigen Anteil unangetastet, Prozessoren ab dem Pentium Pro dagegen setzen ihn auf Null.



Falls mit dem MOV-Befehl das SS-Register verändert werden soll, werden bis nach der Ausführung des folgenden Befehls alle Interrupts unterbunden. Dies erfolgt, um nach einem Neuladen des SS-Registers auch das dazugehörige ESP-Register neu laden zu können, ohne von Interrupts, die ja vom Stack Gebrauch machen, gestört zu werden. Die Nutzung eines »alten« Stack-Pointers in einem »neuen« Stacksegment würde mit sehr hoher Wahrscheinlichkeit zu Problemen führen.

Statusflags Die Statusflags werden durch MOV nicht verändert.

MOVSX
MOVZX Die Befehle *move with sign extension*, MOVSX, und *move with zero extension*, MOVZX, sind Abarten des MOV-Befehls, die im Rahmen des Kopierens den Wertebereich des Datums vorzeichenerweitert (MOVSZ) bzw. vorzeichenlos (MOVZX) auf den des nächst»höheren« Datums erweitern (ShortInt zu SmallInt, SmallInt zu LongInt bzw. Byte zu Word, Word zu DoubleWord).

Operanden

Als Zieloperanden kommen Allzweckregister in Frage, als Quelloperand ein Allzweckregister oder Speicherstellen:

- Kopieren eines Allzweckregisterinhaltes in ein Allzweckregister unter Erweiterung des Wertebereiches
  `MOV Reg16, Reg8; MOV Reg32, Reg8; MOV Reg32, Reg16`
- Kopieren des Inhaltes einer Speicherstelle in ein Allzweckregister unter Erweiterung des Wertebereiches
  `MOV Reg16, Mem8; MOV Reg32, Mem8; MOV Reg32, Mem16`

Statusflags

Statusflags werden nicht verändert.

XCHG

MOV hat einen Nachteil: Es überschreibt den Inhalt des Zieloperanden mit dem Inhalt des Quelloperanden. Sollen dagegen die Inhalte zweier Operanden ausgetauscht werden, ist ein Platz (= Register oder Speicherstelle) nötig, an dem temporär der Inhalt des Zieloperanden zwischengespeichert wird, bevor MOV in Aktion tritt. Mit dem Nachteil, dass dieser temporäre Bereich auch überschrieben wird.

Aus diesem Dilemma hilft der Befehl *exchange*, XCHG. Er tauscht die Inhalte der beiden Operanden aus, indem er einen prozessorinternen temporären Bereich nutzt. Da bei diesem Befehl erster und zweiter Operand sowohl Ziel als auch Quelle sind, spricht man bei XCHG nicht von Ziel- und Quelloperanden, sondern von erstem und zweitem Operanden.

Operanden

Beide Operanden können Register oder Speicherstellen sein. Allerdings ist eine Einschränkung, dass einer der beiden Operanden ein Register sein muss: Der direkte Austausch von Daten zwischen zwei Speicherstellen ist mit XCHG nicht möglich. Falls ein Austausch zwischen einer Speicherstelle und dem Akkumulator erfolgen soll, gibt es eine Ein-Byte-Version, die besonders effektiv ist:

- Austausch der Inhalte zweier Allzweckregister
  `XCHG Reg8, Reg8; XCHG Reg16, Reg16; XCHG Reg32, Reg32`
- Austausch der Inhalte eines Allzweckregisters mit einer Speicherstelle
  `XCHG Reg8, Mem8; XCHG Reg16, Mem16; XCHG Reg32, Mem32`
- Austausch der Inhalte des Akkumulators mit einer Speicherstelle
  `XCHG AX, Mem16; XCHG EAX, Mem32`

Formell möglich, aber in der Wirkung redundant, ist das Vertauschen der Operanden in der Operandenliste, wenn eine Speicherstelle involviert ist:

- Austausch der Inhalte eines Allzweckregisters mit einer Speicherstelle
  `XCHG Mem8, Reg8; XCHG Mem16, Reg16; XCHG Mem32, Reg32`
- Austausch der Inhalte des Akkumulators mit einer Speicherstelle
  `XCHG Mem16, AX; XCHG Mem32, EAX`

Statusflags XCHG verändert keine Statusflags.



XCHG führt einen automatischen LOCK-Befehl aus, falls ein Speicherzugriff im Rahmen von XCHG erfolgt, unabhängig davon, ob der LOCK-Präfix verwendet wird oder nicht! Auf diese Weise ist in jedem Fall sichergestellt, dass während XCHG nur der Prozessor Zugriff auf den Datenbus hat, der den Befehl gerade ausführt.

BSWAP XCHG kann auch dazu benutzt werden, Daten »innerhalb eines Registers« auszutauschen, z.B. `XCHG AH, AL`. Leider ist die Wirksamkeit dieses Befehls auf 16-Bit-Daten beschränkt und dazu noch auf das niedrigerwertige Word der vier Allzweckregister EAX, EBX, ECX und EDX, da nur sie über Alias verfügen, die als Operanden von XCHG akzeptiert werden.

Aus dieser Bedrängnis hilft der Befehl *byte swap*, BSWAP. Er vertauscht byteweise den Inhalt eines 32-Bit-Registers »von vorne nach hinten«:

```
Temp := Reg[07..00];
Reg[07..00] := Reg[31..24]
Reg[31..24] := Reg[07..00]
Temp := Reg[15..08]
Reg[15..08] := Reg[23..16]
Reg[23..16] := Temp
```



BSWAP ist damit der geeignete Befehl, Daten aus dem »Intel-Format« in das »Motorola-Format« zu überführen und umgekehrt (vgl.: »Little-Endian«- und »Big-Endian«-Format« auf Seite 781).

Operanden BSWAP akzeptiert als Operand nur ein 32-Bit-Register:

```
BSWAP Reg32
```

Falls BSWAP ein 16-Bit-Register übergeben wird (Nutzung des operand size override prefix), ist das Ergebnis unbestimmt!



BSWAP verändert keine Statusflags. *Statusflags*

XLAT und XLATB sind zwei Befehle, eine »table look-up translation« durchführen. Hierunter versteht Intel, ein Byte aus einer Tabelle anhand seines Indexes auszulesen. **XLAT** **XLATB**

Table-look-up-Befehle haben keine expliziten Operanden, da die logische Adresse der auszulesenden Tabelle über eine Registerkombination angegeben und der Index über den Akkumulator übergeben wird. Auch das Ziel ist klar: der Akkumulator. Dennoch gibt es neben der »echten«, parameterlosen Form (XLATB) auch eine »parametrische« Form, XLAT. *Operanden*

Die parameterlose Form des XLAT-Befehls erwartet in der Registerkombination DS:(E)BX die Adresse der Byte-Tabelle, aus der das interessierende Byte ausgelesen werden soll:

```
AL := Byte[DS:(E)BX + AL]
```

Der in AL stehende Index wird als vorzeichenloser Offset zur Tabellenbasis interpretiert, somit vorzeichenlos auf 16 bzw. 32 Bit erweitert und zu der in DS:(E)BX stehenden Adresse der Tabelle addiert. Das dadurch im Speicher lokalisierte Byte wird in AL kopiert.

Die »parametrische« Form der table look-up translation erwartet neben einer korrekt belegten Registerkombination DS:(E)BX und dem Index in AL einen explizit angegebenen Operanden. Ihr muss auf Assemblerebene *formal* ein Speicheroperand übergeben werden, *der keinerlei Funktion hat*. Der Assembler übersetzt dann den »parametrischen« Befehl XLAT automatisch in den »parameterfreien« Befehl XLATB. Die parametrische Form wird wie folgt dargestellt:



```
XLAT Mem8
```

Beachten Sie hierbei bitte, dass Mem8 lediglich ein Dummy ist. *Er spielt absolut keine Rolle!* Tatsächlich herangezogen wird XLATB und als Adresse der Tabelle die Adresse, die vorher in die Registerkombination DS:(E)BX eingetragen wurde.



Warum gibt es dann die »parametrische« Form überhaupt? Weiß ich nicht! Und Intel selbst auch nicht: »*This explicit-operand form is provided to allow documentation*«. Aber: »*However, note that the documentation provided by this form can be misleading.*«. Wozu eine Möglichkeit zur Dokumentation, wenn die zu Missverständnissen führen kann? Dann doch lieber gar keine!

Daher mein Tipp: Vergessen Sie einfach die parametrische Form von XLAT, Sie vermeiden dadurch schwer aufzufindende Programmierfehler, die daraus resultieren, dass bei oberflächlicher Betrachtung eine korrekte Nutzung eines übergebenen Operanden vorgegaukelt und vergessen wird, die Registerkombination DS:(E)BX korrekt zu beladen! Und exakt dokumentieren kann man auch anders!

Statusflags Statusflags werden durch XLAT bzw. XLATB nicht verändert.

XADD XADD, *exchange and add*, vertauscht die Inhalte des ersten und zweiten Operanden, addiert sie und schreibt das Ergebnis in den Zieloperanden zurück:

```
Temp := Destination
Destination := Destination + Source
Source := Temp
```

Operanden XADD erwartet als Quelloperanden (zweiter Operand!) ein Register, das Ziel kann entweder ein Register oder eine Speicherstelle sein:

- Austausch und Addition der Inhalte zweier Allzweckregister
  `XADD Reg8, Reg8; XADD Reg16, Reg16; XADD Reg32, Reg32`
- Austausch und Addition der Inhalte eines Allzweckregisters und einer Speicherstelle
  `XADD Mem8, Reg8; XADD Mem16, Reg16; XADD Mem32, Reg32`

Statusflags Die Statusflags werden nach XADD wie nach dem Additionsbefehl ADD gesetzt und spiegeln somit den Zustand der Addition wider.

CMPXCHG
CMPXCHG8B CMPXCHG, *compare and exchange*, und CMPXCHG8B, *compare and exchange 8 bytes*, sind zwei Befehle, die einen »bedingten Austausch« zweier Daten ermöglichen. Allerdings ist dieser bedingte Austausch nicht ganz mit den anderen bedingten Befehlen des Prozessors vergleichbar:

- Es werden keine Statusflags zur Entscheidungsfindung herangezogen, ob die Bedingung erfüllt ist oder nicht.

- Die Prüfung auf Erfüllung der Bedingung (»CMP«-Teil) und die Reaktion auf das Ergebnis (»XCHG«-Teil) erfolgen innerhalb einer Aktion.
- Die Prüfung ist auf Gleichheit der geprüften Daten beschränkt.

CMPXCHG und CMPXCHG8B sind die gleichen Befehle, auch wenn es, an ihren Operanden gemessen, nicht so zu sein scheint! Sie führen folgende Operation durch:

```
if TestValue = DestinationValue
then DestinationValue := SourceValue
else TestValue := DestinationValue
```

Das bedeutet: Sind Testwert und zu testendes Datum gleich, wird in das Ziel der Inhalt der Quelle kopiert. Sind sie es nicht, wird der Testwert mit dem zu prüfenden Datum überschrieben. CMPXCHG verwendet hierzu 8-Bit-, 16-Bit- oder 32-Bit-Daten, also Bytes, Words oder DoubleWords, CMPXCHG8B macht das Gleiche mit 64-Bit-Daten, also QuadWords.

Operanden

Wie das Ablaufschema zeigt, benötigen beide Befehle drei Datenquellen: einen Testwert, einen getesteten Wert und einen Wert, der ggf. zum Verändern des getesteten Wertes benötigt wird (»Korrekturwert«). Der getestete Wert kann dabei entweder in einem Register (bzw., bei CMPXCHG8B, einer Registerkombination) oder an einer Speicherstelle stehen. Da Intel-Prozessor-Befehle nicht mit zwei Speicheroperanden arbeiten können, muss somit sowohl der Testwert als auch der »Korrekturwert« in einem Register stehen. Für den Testwert wurde der Akkumulator reserviert, sodass dieser nicht explizit angegeben werden muss. CMPXCHG hat somit zwei explizite und mit dem Akkumulator einen impliziten Operanden:

- Bedingtes Tauschen mit Allzweckregistern als Operanden, der Testwert befindet sich in AL, AX bzw. EAX:
  `CMPXCHG Reg8, Reg8; CMPXCHG Reg16, Reg16; CMPXCHG Reg32, Reg32`
- Bedingtes Tauschen mit einer Speicherstelle als Operanden, der Testwert befindet sich in AL, AX bzw. EAX:
  `CMPXCHG Mem8, Reg8; CMPXCHG Mem16, Reg16; CMPXCHG Mem32, Reg32`

Am Beispiel von DoubleWords gezeigt führt der Befehl somit folgende Operation durch:

```
if [EAX] = [Mem32/DestReg32]
then [Mem32/DestReg32] := [SourceReg32]; ZF := 1
else [EAX] := [Mem32/DestReg32]; ZF := 0
```

Im Falle von CMPXCHG8B muss der Testwert aufgrund seiner Größe (64 Bit) in einer Registerkombination stehen: EDX:EAX. Damit bleibt für den Ziel- und Quelloperanden nur noch eine Allzweckregister-Kombination (ECX:EBX) übrig. Folglich muss einer der Operanden (der Zieloperand) eine Speicherstelle sein und explizit angegeben werden, während der andere ebenfalls implizit festgelegt ist:

- Bedingtes Tauschen, der Testwert befindet sich in EDX:EAX, der »Korrekturwert« (Quelloperand) in ECX:EBX
  `CMPXCHG8B Mem64`

CMPXCHG8B realisiert somit die folgende Aktion:

```
if [EDX:EAX] = [Mem64]
then [Mem64] := [EXC:EBX]; ZF := 1
else [EDX:EAX] := [Mem64]; ZF := 0
```

Statusflags Falls die Prüfung eine Gleichheit von Testwert und zu testendem Datum zeigt, wird das zero flag gesetzt, andernfalls gelöscht. Bei CMPXCHG werden die anderen Statusflags wie nach CMP gesetzt, bei CMPXCHG8B bleiben sie unverändert.

PUSH POP Der MOV-Befehl als der zentralste und wichtigste Befehl zum Datenaustausch mit dem Speicher adressiert immer das Standard-Datensegment (DS:), sobald ein Speicheroperand involviert ist. Mit Hilfe der segment override prefixes können auch andere Datensegmente benutzt werden, unter anderem auch das Stacksegment (SS:).

Das Stacksegment ist jedoch ein Datensegment, das sich nicht unerheblich von anderen Datensegmenten unterscheidet. Am auffälligsten ist, dass es »von oben nach unten« wächst wie die Stalagtiten in einer Tropfsteinhöhle, während andere Datensegmente »von unten nach oben« wachsen. Aber ein anderer Aspekt hebt es noch in entscheidenderem Maße von »normalen« Datensegmenten ab: Es ist der Notizzettel des Prozessors. Hier legt er wichtige Daten ab, wie z.B. Rücksprungadressen, wenn Unterprogramme aufgerufen werden, oder Parameter, die an Unterprogramme übergeben werden sollen.

Es verwundert daher nicht, dass es »MOV«-Befehle gibt, die dieser Sonderfunktion Rechnung tragen und auf die spezielle Funktion des Stacks eingehen. Zwei dieser Befehle sind PUSH und POP, die ein Datum »auf den Stack PUSHen«, sprich schreiben, oder »vom Stack POPpen«, sprich entfernen.

Der Stack wird dabei genauso behandelt wie das Gebilde, nach dem er benannt ist: wie ein Stapel. Das bedeutet: Man kann mit PUSH und POP nur ein Datum auf den Stapel legen oder das oberste Datum von ihm entfernen!



Bitte beachten Sie, dass das nur für die speziellen »Stack-Befehle« PUSH und POP gilt. Natürlich kann das Stacksegment auch wie jedes andere Datensegment über eine logische Adresse (SS:Offset), z.B. mit dem MOV-Befehl, angesprochen werden.

Da mit PUSH und POP immer das Datum auf der Spitze des Stapels angesprochen wird, braucht die Adresse nicht explizit angegeben zu werden! Das ist auch der entscheidende Vorteil gegenüber der MOV-Version, die ja jeweils die Adresse benötigt. Der Prozessor besitzt zwei Register, die die aktuelle Stackspitze verwalten: Das Segmentregister SS: und das Stackpointer-Register (E)SP.

Die Funktion der Befehle ist einfach: PUSH dekrementiert (!) zunächst den Inhalt von (E)SP, sodass SS:(E)SP nun auf eine freie Stelle auf dem Stack zeigt, der Stack also gewachsen ist. An diese Position wird nun der Inhalt des Operanden von PUSH kopiert. POP geht den umgekehrten Weg: Zunächst wird der Inhalt von SS:(E)SP in den Operanden von POP kopiert und dann (E)SP inkrementiert (!). Der Stack ist geschrumpft.



Bitte denken Sie an die Stalagtiten, wenn Sie im Kopf die Stackspitze verschieben! Der Stack wächst zu niedrigeren Adressen, weshalb die Adresse in (E)SP dekrementiert werden muss und er schrumpft zu höheren Adressen, was ein Inkrementieren bewirkt.

Wer dekrementiert/inkrementiert? Und mit welchem Wert? Der Prozessor! Wie jedes andere Segment auch hat das Stacksegment ein Flag, das die »Standard-Datengröße« bestimmt. Im Codesegment ist es das D-Bit, bei Daten- und Stacksegmenten das B-Bit im jeweiligen Deskriptor. Ist es im Stacksegment gesetzt, so benutzt der Prozessor das 32-Bit-ESP-Register als Stackpointer, der Stack ist dann »32-bittig«. Ist es gelöscht, repräsentiert den Stackpointer das 16-Bit-SP-Register, der stack ist dann 16-bittig.

Die Anzahl zu addierender Bytes liest er aus dem D-Flag des Code- bzw. dem B-Flag des Datensegmentes – je nachdem, woher der Operand kommt (und welches Segment damit betroffen ist) und welches

Datum betroffen ist (Adressen oder »echte« Daten). Damit spielen auch etwaige operand size bzw. address size prefixes eine Rolle. Sind die jeweiligen Flags gesetzt oder zwingen die override prefixes dazu, addiert/subtrahiert er vier Bytes zum Stackpointer, da die Segmente dann 32-bittig ausgelegt sind. Sind sie gelöscht, werden nur zwei Bytes verwendet (16-bittig).



Das kann zu Problemen führen, wenn Daten- oder Codesegment 16-bittig ausgelegt ist, das Stacksegment jedoch 32-bittig. Bei einem PUSH von Daten oder Adressen werden dann nur zwei Bytes auf den Stack geschoben. Die neue Adresse in ESP liegt dann aber nicht an Double-Word-, sondern nur an Word-Grenzen. Diesen Sachverhalt nennt man »*misalignment*« des Stacks (»falsche Ausrichtung«). Falls der sog. alignment check eingeschaltet ist, führt das zu einer alignment check exception (#AC).

Operanden Beide Befehle können Konstanten, Inhalte von Registern oder Speicherstellen als Operanden akzeptieren (XXX steht für PUSH bzw. POP):

- PUSHen/POPpen einer Konstanten
  `XXX Const8; XXX Const16; XXX Const32`
- PUSHen/POPpen eines Registerinhalts
  `XXX Reg16; XXX Reg32`
- PUSHen/POPpen des Inhalts einer Speicherstelle
  `XXX Mem16; XXX Mem32`
- PUSHen/POPpen des Inhalts eines Segmentregisters
  `XXX CS; XXX DS; XXX ES; XXX FS; XXX GS; XXX SS`



Nachdem auch die Inhalte von Registern auf den Stack geschoben werden können, kann man auch das (E)SP-Register verwenden. Damit hat man ein Problem: (E)SP enthält den Stackpointer; dieser wird vor dem Kopieren auf den Stack dekrementiert. Wird nun der »alte« Inhalt – vor dem Dekrementieren – an die neue Stackspitze geschrieben oder der »neue« – nach dem Dekrementieren? Antwort: der alte! Dies gilt übrigens auch für Werte, bei denen zunächst die Adresse berechnet werden muss (»indirekte Adressierung«) und bei denen hierbei das (E)SP-Register involviert ist. Regel: Zuerst wird die Adresse berechnet und dann dekrementiert (und somit (E)SP verändert). Analoges gilt für POP, nur umgekehrt: Zuerst wird inkrementiert und der an der neuen Stackposition stehende Wert für die Adressberechung verwendet.



Nachdem auch Segmentregister Operand für die Befehle sein können, kann mittels POP auch ein Segmentregister geladen werden. Der hierbei verwendete Selektor muss gültig sein und auf ein gültiges Segment zeigen, da jedes Beschreiben eines Segmentregisters den Inhalt des durch den Selektor spezifizierten Deskriptors in den verborgenen Teil des Segmentregisters schreibt. Damit aber ist die Validierung des Segments verbunden inklusive der Prüfung der Privilegien. Es kann zwar, ohne eine exception auszulösen, ein Null-Selektor in ein Segmentregister gePOPpt werden. Jeder folgende Zugriff auf dieses Register führt dann jedoch zu einer general protection exception (#GP).



Das CS-Register kann durch POP nicht neu geladen werden! Um einen neuen Wert in das CS-Register zu schreiben, ist der RET-Befehl erforderlich.

Statusflags

Statusflags werden durch PUSH und POP nicht verändert.

PUSHA
POPA
PUSHAD
POPAD

PUSHA, *push all general-purpose registers*, und POPA, *pop all general-purpose registers*, sind zwei Befehle, die die Register aller Allzweckregister auf den stack schieben oder von dort holen. Sie erfüllen somit folgende Aufgabe »in einem Rutsch«:

```
PUSHA: Temp := (E)SP
       PUSH (E)AX
       PUSH (E)CX
       PUSH (E)DX
       PUSH (E)BX
       PUSH Temp
       PUSH (E)BP
       PUSH (E)SI
       PUSH (E)DI

POPA:  POP (E)DI
       POP (E)SI
       POP (E)BP
       ADD (E)SP, (4)2       ; (E)SP-Inhalt vor PUSHA; daher ADD!
       POP (E)BX
       POP (E)DX
       POP (E)CX
       POP (E)AX
```

Ob durch PUSHA/POPA die 16- oder 32-Bit-Register verwendet werden, entscheidet die Umgebung, in der die Befehle aufgerufen werden.



PUSHAD, *push all general-purpose registers as doublewords*, und POPAD, *pop all general-purpose registers as double words*, sind Alias von PUSHA und POPA. Manche Assembler erzwingen bei Verwendung von PUSHA/POPA Befehlssequenzen für 16-Bit-Register, bei PUSHAD/POPAD für 32-Bit-Register. In der Regel aber werden die Assembler die korrekten Befehlssequenzen anhand der aktuellen Umgebung setzen.

Operanden PUSHA, POPA, PUSHAD und POPAD haben keine Operanden.

Statusflags PUSHA, POPA, PUSHAD und POPAD verändern die Statusflags nicht.

IN
OUT Die bislang betrachteten Befehle ermöglichten einen Datenaustausch mit dem Speicher, sofern nicht prozessorintern Daten in einzelnen Registern ausgetauscht wurden. Doch neben der Kommunikation mit dem Speicher beherrscht der Prozessor natürlich auch die Kommunikation mit externen »Geräten« wie Druckern, Modems etc. Hierzu bedient er sich der Ports. (Zur Beschreibung von Ports vgl. »Ports« auf Seite 827.) Und was beim Speicher der MOV-Befehl ist, ist bei Ports das Befehlspaar IN – OUT. Hierbei übernimmt IN das Lesen eines Datums »aus dem Port«, während OUT ein Datum »über ein Port ausgibt«.



Die Kommunikation mit der Peripherie ist bei den modernen Betriebssystemen Sache des Betriebssystems! Es allein hat und muss die Kontrolle über den Zugriff haben. Daher können Sie in der Regel im protected mode Ports direkt nicht mehr ansprechen, sondern müssen Betriebssystemfunktionen benutzen. Dies ist Teil der Schutzkonzepte im protected mode.

Die Port-Adresse ist bei beiden Befehlen vorgegeben: Sie wird im DX-Register abgelegt. Der Datenaustausch erfolgt über den Akkumulator.



Beachten Sie hierbei, dass unabhängig von der Umgebung (16-Bit- bzw. 32-Bit-Umgebungen) die Port-Adresse immer 16-bittig ist, da die IA-32-Architektur von Intel »nur« 65.536 Ports zulässt. Somit reichen zu einer Adressierung der Ports Words und damit ein Word-Register aus. Allerdings haben die Ports 0 bis 255 eine herausragende Bedeutung, sodass auch eine Byte-Konstante als Portadresse übergeben werden kann.

Dagegen bestimmt der Akkumulator die Datengröße des zu übertragenden Datums: Wird AL benutzt, werden Bytes ausgetauscht, bei AX Words und bei EAX DoubleWords.

IN und OUT können somit folgende Operanden annehmen: *Operanden*

- Adressierung des Ports durch eine Konstante
```
IN  AL, Const8; IN  AX, Const8; IN  EAX, Const8
OUT Const8, AL; OUT Const8, AX; OUT Const8, EAX
```
- Adressierung des Ports durch das DX-Register
```
IN  AL, DX; IN  AX, DX; IN  EAX, DX
OUT DX, AL; OUT DX, AX; OUT DX, EAX
```

Die Statusflags werden durch IN und OUT nicht beeinflusst. *Statusflags*

1.1.6 Operationen zur Datenkonvertierung

Unter Datenkonvertierung versteht die CPU die Überführung einer Integer in eine andere, konkret das Erweitern des Wertebereiches einer Integer. Somit ist der umgekehrte Weg nicht realisiert.



Lassen Sie sich durch die Begriffe »Byte«, »Word«, »DoubleWord« und »QuadWord« in den Mnemonics der folgenden Befehle nicht verwirren! Die Befehle verarbeiten vorzeichenbehaftete Zahlen, berücksichtigen also ein Vorzeichen. Daher können sie ShortInts in SmallInts, SmallInts in LongInts und LongInts in QuadInts konvertieren. Die Konversion führt nur dann mit vorzeichenlosen Integers (Bytes, Words, DoubleWords) zu korrekten Ergebnissen, wenn deren MSB nicht gesetzt ist! Der Grund dafür ist, dass es die einzige und eigentliche Aufgabe aller Konvertierungsbefehle ist, eine Vorzeichenerweiterung (»sign extension«) durchführen – und sonst nichts!

CBW
CWD
CDQ

CBW, *convert byte to word*, CWD, *convert word to double word*, und CDQ, *convert double word to quad word* führen genau diese Vorzeichenerweiterung durch. Das Datum muss dazu im Akkumulator stehen. CBW kopiert nun das MSB (= Vorzeichenbit 7) der ShortInt in AL achtmal in die Bitpositionen 8 bis 15, CWD das MSB (= Vorzeichenbit 15) der SmallInt aus AX 16-mal in DX (!) und CDQ das MSB (Vorzeichenbit 31) der LongInt aus EAX 32-mal in EDX. Das Ergebnis sind eine SmallInt in AX, eine LongInt in DX:AX und eine QuadInt in EDX:EAX mit korrektem Vorzeichen. (Vergleiche hierzu den Abschnitt »Codierung von Integers« auf Seite 801.)

CWDE

CWD wurde noch zu einer Zeit realisiert, als die Prozessoren noch nicht über 32-Bit-Register verfügten und daher als zwei 16-Bit-Teile behandelt werden mussten. Daher legt CDW die LongInt in der Registerkom-

bination DX:AX ab. Mit dem Aufkommen der 32-Bit-Prozessoren wurde daher ein Zwilling für CWD geschaffen, der die LongInt in ein 32-Bit-Register ablegt: CWDE, convert word to DoubleWord in extended register. Dieser Befehl entnimmt dem MSB der SmallInt in AX das Vorzeichen und kopiert es 16-mal in die Bitpositionen 16 bis 31 des EAX-Registers.

Operanden Die Befehle haben keine expliziten Operanden. Sie verwenden implizite Quell-/Ziel-Operanden: das AL-/AX-Register (CBW), das AX-/DX:AX-Register (CWD), das AX-/EAX-Register (CWDE) bzw. das EAX/EDX:EAX-Register (CDQ).

Statusflags Statusflags werden nicht verändert.



CBW und CWDE besitzen den gleichen Opcode ($98), sind also identisch. Das mag zunächst verwundern, ist aber dennoch logisch. Mit CBW soll eine ShortInt in eine SmallInt konvertiert werden, den vorzeichenbehafteten Standardwert von 16-Bit-Prozessoren. CWDE konvertiert eine SmallInt in eine LongInt, den vorzeichenbehafteten Standardwert von 32-Bit-Prozessoren. Das bedeutet, beide Befehle konvertieren jeweils in einen Standardwert in einer bestimmten Umgebung. In 32-Bit-Umgebungen (32-Bit-Prozessoren und 32-Bit-Betriebssystem) ist die Standard-Datengröße 32 Bits, in 16-Bit-Umgebungen (16-/32-Bit-Prozessoren und 16-Bit-Betriebssystem) ist sie 16 Bits. Daher wird in 32-Bit-Umgebungen der Opcode $98 durch das Mnemonic CWDE, in 16-Bit-Umgebungen durch CBW repräsentiert. Nutzt man nun in 16-Bit-Umgebungen das Mnemonic CWDE, so wird dem Opcode der operand size override prefix vorangestellt. Analog wird er verwendet, wenn in 32-Bit-Umgebungen das Mnemonic CBW benutzt wird. In der Regel erfolgt das für den Assemblerprogrammierer transparent durch den Assembler.

Analoges erfolgt übrigens mit CWD und CDQ – beide haben den Opcode $99. In 16-Bit-Umgebungen wird diesem Opcode der operand size override prefix vorangestellt, wenn CDQ verwendet wird, in 32-Bit-Umgebungen, wenn CWD genutzt wird.

1.1.7 Verzweigungen im Programmablauf: Sprungbefehle

Für ein besseres Verständnis der Inhalte dieses Kapitels empfiehlt es sich, zunächst das Kapitel »Speicherverwaltung« auf Seite 394 gelesen zu haben.



JMP

JMP, *jump*, dient dazu, die Programmausführung an einer anderen Stelle des Programms fortzusetzen. Hierzu verändert JMP den Inhalt von (E)IP und ggf. des Codesegment-Registers, was auf anderem Wege »von außen« nicht möglich ist

Man unterscheidet vier Arten von Jumps:

- Short jumps; bei diesen Sprüngen handelt es sich um »ultrakurze« Sprünge mit einer Distanz von –128 bis +127 Bytes von der aktuellen, in (E)IP stehenden Position. Short jumps sind somit *relative* Intrasegment-Sprünge.
- Near jumps; bei diesen Sprüngen handelt es sich um Intrasegment-Sprünge, also Sprünge, bei denen das Sprungziel innerhalb des aktuellen Segments liegt. Near jumps können *relativ* angegeben werden, also als Distanz von der in (E)IP stehenden aktuellen Position aus. Short jumps sind somit eine Untergruppe der relativen near jumps. Eine weitere Möglichkeit ist die Angabe eines Offsets im aktuellen Segment. In diesem Fall spricht man von *absoluten* near jumps. Absolute near jumps sind immer auch *indirekte* Sprünge, da als Operand nicht die Zieladresse selbst, sondern nur ein Register oder eine Speicherstelle angegeben wird, in der das Sprungziel steht. Der Prozessor muss somit erst den Operanden auslesen, bevor er die neue Adresse in (E)IP eintragen kann. Relative Sprünge dagegen sind immer *direkte* Sprünge!
- Far jumps; diese Sprünge nennt man auch Intersegment-Sprünge, da das Sprungziel außerhalb des aktuellen in einem anderen Segment liegt, das aber die gleiche Privilegstufe besitzen muss. Far jumps sind damit immer *absolute* Sprünge, da das Sprungziel über eine qualifizierte logische Adresse angegeben wird. Far jumps kommen als direkte und indirekte Sprünge vor: Im einen Fall wird als Operand eine Speicherstelle übergeben, in der das Sprungziel steht (indirekte Adressierung), im anderen Fall stellt der Operator selbst eine logische Adresse dar (direkte Adressierung).

- Task switches; hierunter versteht man den Sprung an eine Position in einem Codesegment, das in einem anderen Task verwendet wird. Ein task switch ist immer ein indirekter far jump, bei dem zusätzlich neben einer neuen Adresse (CS:EIP) auch andere Prozessorregister verändert werden (Task-Umgebung).



Als Assemblerprogrammierer brauchen Sie sich um die Sprungdistanzen nicht zu kümmern. Sie programmieren den Sprungbefehl, indem Sie jeweils die Adresse eines Labels angeben. Der Assembler berechnet dann selbstständig die Sprungdistanz anhand der Zieladresse und der aktuellen Programmposition und codiert sie in der Befehlssequenz.



Die Behandlung von far jumps im protected mode einerseits und im real mode bzw. virtual 8086 mode andererseits erfolgt etwas unterschiedlich. Während im real mode und virtual 8086 mode das Sprungziel direkt aus der übergebenen Adresse (direkt oder indirekt) berechnet werden kann, spielen im protected mode die Schutzkonzepte eine Rolle. Dies bedeutet, dass far jumps nur unter folgenden Bedingungen ausgeführt werden können. Trifft keine der Bedingungen zu, wird eine general protection exception #GP ausgelöst.

- Sprung in ein non-conforming code segment mit gleicher Privilegstufe (RPL ≤ CPL und DPL = CPL).
- Sprung in ein conforming code segment mit niedrigerer oder gleicher Privilegstufe (DPL ≤ CPL).
- Sprung über ein call gate
- task switch

In allen Fällen benutzt der Prozessor den Selektor-Anteil aus der übergebenen Adresse, um auf den dazugehörigen Deskriptoren in der GDT oder LDT zuzugreifen (vgl. »Segmenttypen, Gates und ihre Deskriptoren« auf Seite 407). Die hier verzeichneten Schutzattribute werden in die Prüfungen der Rechtmäßigkeit des Sprungs einbezogen.

Wird ein call gate benutzt, so benutzt der Prozessor lediglich den Selektor-Anteil der übergebenen Adresse, der Offset-Anteil wird verworfen. Grund: Das eigentliche Sprungziel steht im Deskriptor des call gates, sodass über den Operanden des Sprungbefehls lediglich der Selektor auf diesen call gate descriptor übergeben werden muss. (Nichtsdestoweniger aber muss ein Dummy-Offset übergeben werden, damit eine vollständige logische Adresse als Parameter übergeben wird!)

Ein task switch ist generell nicht wesentlich unterschiedlich zur Nutzung eines call gates: Auch in diesem Fall wird lediglich der Selektor-Anteil benutzt, um in der GDT den Deskriptoren des task state segments zu identifizieren. Hier stehen die Informationen, die notwendig sind, um den task switch – und damit auch den far jump – durchzuführen.

Operanden

Der JMP-Befehl kann somit folgende Operanden besitzen:

- Direkter, relativer short oder near jump
  `JMP Dist8; JMP Dist16; JMP Dist32`
- Indirekter, relativer near jump, Zieladresse in einem Register
  `JMP Reg16; JMP Reg32`
- Indirekter, relativer near jump, *effektive Adresse* des Ziels in einer Speicherstelle
  `JMP Mem16; JMP Mem32`
- Direkter, absoluter far jump
  `JMP Selektor:EA16; JMP Selektor:EA32`
- Indirekter, absoluter far jump; *logische Adresse* des Ziels in einer Speicherstelle
  `JMP Mem16+16; JMP Mem16+32`

Statusflags

Die Statusflags werden lediglich im Rahmen eines task switches verändert. In diesem Fall jedoch alle, weil der Zustand des Flagregisters beim letzten switch restauriert wird. Somit ist sehr wahrscheinlich, dass sich der Status der Statusflags durch den task switch ändert.

Bei allen anderen Jumps bleiben die Statusflags unverändert.

Jcc

Jump on condition cc, Jcc, ist eine Gruppe von Befehlen, die einen relativen short/near jump ausführen, wenn die Bedingung cc erfüllt ist. Hierbei werden zwei verschiedene Prüfungen verwendet:

- Prüfung des CX- bzw. ECX-Registers
- Prüfung der Statusflags

JCXZ
JECXZ

Soll das CX-/ECX-Register geprüft werden, stellen die Befehle JCXZ bzw. JECXZ fest, ob der Inhalt Null ist. Ist das der Fall, ist die Bedingung erfüllt und die als Operand übergebene, vorzeichenbehaftete Sprungdistanz wird zum Inhalt des (E)IP-Registers addiert. Andernfalls wird mit dem unmittelbar folgenden Befehl die Programmausführung fortgeführt.



Mit JCXZ und JECXZ sind nur short jumps möglich, das heißt, die Sprungdistanz muss zwischen –128 und +127 Bytes von der aktuellen Position liegen.

Werden die Statusflags zur Entscheidungsfindung herangezogen, gibt es gemäß der in Tabelle 1.1 auf Seite 43 dargestellten Möglichkeiten der Flagprüfung folgende bedingte Sprungbefehle:

Befehl	Sprung, wenn	Synonyme	Prüfung
JA	größer	JNBE	CF=0 & ZF=0
JAE	größer, gleich	JNB, JNC	CF=0
JB	kleiner	JNAE, JC	CF=1
JBE	kleiner, gleich	JNA	CF=1 | ZF = 1
JC	carry gesetzt		CF=1
JE	gleich	JZ	ZF= 1
JG	größer (±)	JNLE	OF=SF & ZF=0
JGE	größer, gleich (±)	JNL	OF=SF
JL	kleiner (±)	JGE	OF≠SF
JLE	kleiner, gleich (±)	JG	OF≠SF | ZF=1
JNA	nicht größer	JBE	CF=1 | ZF = 1
JNAE	nicht größer, gleich	JB	CF=1
JNB	nicht kleiner	JAE	CF=0
JNBE	nicht kleiner, gleich	JA	CF=0 & ZF=0
JNC	carry gelöscht		CF=0
JNE	nicht gleich	JNZ	ZF=0
JNG	nicht größer (±)	JLE	OF≠SF | ZF=1
JNGE	nicht größer, gleich (±)	JL	OF≠SF
JNL	nicht kleiner (±)	JGE	OF=SF
JNLE	nicht kleiner, gleich (±)	JG	OF=SF & ZF=0
JNO	overflow gelöscht		OF=0
JNP	parity gelöscht	JPO	PF=0
JNS	sign gelöscht		SF=0
JNZ	zero gelöscht	JNE	ZF=0
JO	overflow gesetzt		OF=1
JP	parity gesetzt	JPE	PF=1
JPE	parity gesetzt	JP	PF=1
JPO	parity gelöscht	JNP	PF=0
JS	sign gesetzt		SF=1
JZ	zero gesetzt	JE	ZF=1

Tabelle 1.5: Bedingte Sprungbefehle und die mit ihnen verbundenen Prüfungen der Statusflags

Operanden

Die Jcc-Befehle haben als Operanden immer eine Distanz zum Sprungziel, die sich auf die aktuelle Position im Programm bezieht, die durch den Inhalt von (E)IP bestimmt wird. Das bedeutet, dass der Operand eine vorzeichenbehaftete Zahl ist, die zum (E)IP-Inhalt addiert wird:

```
Jcc Dist8; Jcc Dist16; Jcc Dist32
```

Bitte beachten Sie, dass bei JCXZ und JECXZ nur eine Dist8 als Operand übergeben werden kann!



Auf Assemblerebene wird als Operand immer ein Label angegeben. Für den Programmierer sieht es somit so aus, als ob die Sprungziele mittels Absolutadressen (Offset des Labels zum Segmentbeginn) angegeben werden. Dies ist jedoch nicht der Fall: Der Assembler bestimmt aus der absoluten Adresse des Labels und dem aktuellen Programmzeiger eine Sprungdistanz, die als Operand codiert wird. Falls Sie somit jemals in die Lage kommen sollten (zugegeben: ich wüsste nicht, warum!), »von Hand« Sprungbefehle zu codieren, berücksichtigen Sie bitte diesen Sachverhalt.

In der Regel haben Sie keinen Einfluss darauf, welche Operandengröße (Dist8, Dist16 oder Dist32) verwendet wird! Der Assembler wird versuchen, die 8-Bit-Distanzen zu codieren, wenn dies möglich ist. Andernfalls wird anhand der Umgebung (16-Bit, 32-Bit) entschieden, ob die Sprungdistanzen mit 16 oder 32 Bit codiert werden.



Die Jcc-Befehle mit 8-Bit-Operanden (und somit einem Sprungziel, das zwischen -128 und +127 Bytes von der aktuellen Position entfernt ist), sind besonders effektiv, da sie durch Ein-Byte-Opcodes codiert werden.



Bitte beachten Sie, dass die in Tabelle 1.5 aufgeführten 30 Befehle durch »nur« 16 Opcodes realisiert werden. Der Grund hierfür ist, dass bei acht Befehlen semantische Redundanzen vorliegen (»größer« ist identisch mit »nicht kleiner oder gleich«), vier Befehle mit unterschiedlichen Mnemonics benutzt werden können (»gleich« und »zero« prüfen das zero flag) und zwei weitere Befehle sogar mit zwei weiteren, redundant vorliegenden Befehlen identisch sind (genauer: das gleiche Flag abprüfen; »above or equal« ist identisch mit »not below« und prüft wie »carry« das carry flag). Somit werden mit 12 Opcodes bereits 26 Mnemonics realisiert. Die verbleibenden vier Opcodes haben jeweils ihr »eigenes« Mnemonic. Vergleiche hierzu auch Tabelle 1.1 auf Seite 43.

LOOP
LOOPcc

Der LOOP-Befehl ist quasi ein bedingter Sprungbefehl mit »eingebautem« Zähler. Das bedeutet, man kann mit ihm Schleifen programmieren, die `count`-mal abgearbeitet werden, bevor die Programmausführung mit dem auf den LOOP-Befehl folgenden fortgesetzt wird. Den LOOP-Befehl gibt es in zwei Varianten:

- dem »einfachen« LOOP-Befehl, der lediglich den Zähler berücksichtigt, und
- dem LOOPcc-Befehl, der neben dem Zählerstand auch noch das zero flag auswertet.

Der involvierte Zähler befindet sich in (E)CX. In 16-Bit-Umgebungen wird CX verwendet, in 32-Bit-Umgebungen ECX. LOOP/LOOPcc dekrementiert den Inhalt des Zählers um 1 und prüft, ob das Ergebnis Null ist. Ist dies der Fall, erfolgt keine Programmverzweigung und die Befehlsausführung wird mit dem auf den LOOP-/LOOPcc-Befehl folgenden fortgesetzt. Ist dagegen das Ergebnis von Null verschieden, so verzweigt der »einfache« LOOP-Befehl zum Sprungziel, das über den Operanden angegeben wird.

LOOPcc dagegen prüft nun das zero flag. Je nach Status dieses Flags kann ebenfalls eine Programmverzweigung erfolgen oder nicht. Gemäß Tabelle 1.1 auf Seite 43 gibt es somit folgende Mnemonics:

Befehl	Sprung, wenn	Synonyme	Prüfung
LOOPE	gleich	LOOPZ	ZF=1
LOOPNE	nicht gleich	LOOPNZ	ZF=0

Tabelle 1.6: Bedingte LOOP-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags

Operanden

Wie die bedingten Sprungbefehle (Jcc) auch, haben die LOOP-Befehle immer eine Sprungdistanz von der aktuellen Programmposition als Parameter, die das Sprungziel angibt. Es handelt sich um eine vorzeichenbehaftete Zahl, die vom Prozessor zum Inhalt des (E)IP-Registers addiert wird, wenn die Sprungbedingung erfüllt ist. LOOP-Befehle sind somit bedingte short jumps!

```
LOOP Dist8; LOOPE Dist8; LOOPNE Dist8; LOOPZ Dist8; LOOPNZ Dist8
```



Auf Assemblerebene wird als Operand immer ein Label angegeben. Für den Programmierer sieht es somit so aus, als ob die Sprungziele mittels Absolutadressen (Offset des Labels zum Segmentbeginn) angegeben werden. Dies ist jedoch nicht der Fall: Der Assembler bestimmt

aus der absoluten Adresse des Labels und dem aktuellen Programmzeiger eine Sprungdistanz, die als Operand codiert wird.

Statusflags

Die Statusflags werden von LOOP/LOOPcc nicht verändert. LOOPcc jedoch prüft das zero flag!

CALL
RET

CALL dient dazu, die Programmausführung an einer anderen Stelle des Programms fortzusetzen, wobei im Unterschied zu dem JMP-Befehl wieder an die Stelle zurückgekehrt werden soll, an der die Programmverzweigung erfolgte. Mit CALL werden somit Routinen »aufgerufen«, also Funktionen oder Prozeduren.

Der CALL-Befehl ist dem JMP-Befehl sehr ähnlich. Sie unterscheiden sich im Prinzip nur in einem einzigen Punkt: Bevor der Sprung zum Sprungziel ausgeführt wird, wird vom Prozessor eine »Rücksprungadresse« auf den Stack gelegt. Diese Rücksprungadresse ist diejenige, die auf den CALL-Befehl folgt.

Das bedeutet, dass ein CALL nur dann Sinn macht, wenn im Verlauf der Abarbeitung des Programmcodes, zu dem mittels CALL verzweigt wurde, – also im »gerufenen« Programmcode – auch ein Befehl abgearbeitet wird, der den »Rücksprung« bewirkt. Dies ist der Befehl RET, *return*. CALL und RET bilden somit ein Paar, wobei CALL im *rufenden* und RET im *gerufenen* Programmteil realisiert wird.

RET selbst holt die Rücksprungadresse, die CALL auf den Stack gelegt hat, wieder vom Stack und schreibt sie in CS:(E)IP. Dadurch erfolgt der Rücksprung an die auf den CALL-Befehl folgende Adresse.

Analog den Jumps unterscheidet man drei Arten von Calls:

- Near calls; bei diesen Sprüngen handelt es sich um Intrasegment-CALLs, also Sprünge, bei denen das Sprungziel innerhalb des aktuellen Segments liegt. Near calls können *relativ* angegeben werden, also als Distanz von der in (E)IP stehenden aktuellen Position aus. Eine weitere Möglichkeit ist die Angabe eines Offsets im aktuellen Segment. In diesem Fall spricht man von *absoluten* near calls. Absolute near calls sind immer auch *indirekte* Sprünge, da als Operand nicht die Zieladresse selbst, sondern nur ein Register oder eine Speicherstelle angegeben wird, in der das Sprungziel steht. Der Prozessor muss somit erst den Operanden auslesen, bevor er die neue Adresse in (E)IP eintragen kann. Relative Sprünge dagegen sind immer *direkte* Sprünge!

- Far calls; diese Sprünge nennt man auch Intersegment-CALLs, da das Sprungziel außerhalb des aktuellen in einem anderen Segment liegt, das aber die gleiche Privilegstufe besitzen muss. Far calls sind damit immer *absolute* Sprünge, da das Sprungziel über eine qualifizierte logische Adresse angegeben wird. Far calls kommen als direkte und indirekte Sprünge vor: Im einen Fall wird als Operand eine Speicherstelle übergeben, in der das Sprungziel steht (indirekte Adressierung), im anderen Fall stellt der Operator selbst eine logische Adresse dar (direkte Adressierung).
- Task switches; hierunter versteht man den Sprung an eine Position in einem Codesegment, das in einem anderen Task verwendet wird. Ein task switch ist immer ein indirekter far call, bei dem zusätzlich neben einer neuen Adresse (CS:EIP) auch andere Prozessorregister verändert werden (Task-Umgebung).



Als Assemblerprogrammierer brauchen Sie sich um die Sprungdistanzen nicht zu kümmern. Sie programmieren den CALL-Befehl, indem Sie jeweils die Adresse eines Labels angeben. Der Assembler berechnet dann selbstständig die Sprungdistanz anhand der Zieladresse und der aktuellen Programmposition und codiert sie in der Befehlssequenz.



Die Behandlung von far calls im protected mode einerseits und im real mode bzw. virtual 8086 mode andererseits erfolgt etwas unterschiedlich. Während im real mode und virtual 8086 mode das Sprungziel direkt aus der übergebenen Adresse (direkt oder indirekt) berechnet werden kann, spielen im protected mode die Schutzkonzepte eine Rolle. Dies bedeutet, dass far calls nur unter folgenden Bedingungen ausgeführt werden können. Trifft keine der Bedingungen zu, wird eine general protection exception #GP ausgelöst.

- Sprung in ein non-conforming code segment mit gleicher Privilegstufe (RPL ≤ CPL und DPL = CPL).
- Sprung in ein conforming code segment mit niedrigerer oder gleicher Privilegstufe (DPL ≤ CPL).
- Sprung über ein call gate
- task switch

In allen Fällen benutzt der Prozessor den Selektor-Anteil aus der übergebenen Adresse, um auf den dazugehörigen Deskriptor in der GDT oder LDT zuzugreifen (vgl. »Segmenttypen, Gates und ihre Deskriptoren« auf Seite 407). Die hier verzeichneten Schutzattribute werden in die Prüfungen der Rechtmäßigkeit des Sprungs einbezogen.

Wird ein call gate benutzt, so benutzt der Prozessor lediglich den Selektor-Anteil der übergebenen Adresse, der Offset-Anteil wird verworfen. Grund: Das eigentliche Sprungziel steht im Deskriptoren des call gates, sodass über den Operanden des Sprungbefehls lediglich der Selektor auf diesen call gate descriptor übergeben werden muss. (Nichtsdestoweniger aber muss ein Dummy-Offset übergeben werden, damit eine vollständige logische Adresse als Parameter übergeben wird!)

Ein call gate muss auch benutzt werden, wenn ein Inter-Privileg-CALL erfolgen soll, also ein Codesegment angesprungen werden soll, das von der aktuellen Privilegstufe unterschiedliche Privilegien besitzt. In diesem Fall wird auch ein stack switch durchgeführt.

Ein task switch ist generell nicht wesentlich unterschiedlich zur Nutzung eines call gates: Auch in diesem Fall wird lediglich der Selektor-Anteil benutzt, um in der GDT den Deskriptoren des task state segments zu identifizieren. Hier stehen die Informationen, die notwendig sind, um den task switch – und damit auch den far jump – durchzuführen.

Korrespondierend zu den calls gibt es die entsprechenden returns:

- Near returns; dies ist der Gegenspieler zum near call. Da ein near call lediglich den Inhalt des (E)IP-Registers als Rücksprungadresse auf den Stack schiebt (Intrasegment-CALLs!), holt ein near return auch nur diesen Offset wieder von Stack und transferiert ihn in das (E)IP-Register zurück.
- Far returns; als Gegenspieler zum far call lädt ein far return eine vollständige logische Adresse (CS:(E)IP) vom Stack zurück. Man unterscheidet die far returns in zwei Klassen: Intersegment-Returns, bei denen zwar das Segment geändert wird, nicht aber die Privilegstufe der Schutzkonzepte. Bei den Interprivileg-Returns dagegen wird auch die Privilegstufe geändert. Dies ist bei der Rückkehr aus Codesegmenten der Fall, die über ein call gate abgelaufen sind. Falls im Rahmen eines Interprivileg-Returns invalide Selektoren in DS, ES, FS und GS gefunden werden, werden sie gelöscht. Da ein Interprivileg-Return immer mit einem stack switch verbunden ist, werden auch SS und (E)SP neu belegt.

Falls vor Aufruf der Routine Parameter für die Routine auf den Stack gelegt wurden, liegen sie »unter« der Rücksprungadresse auf dem Stack. Den RET-Befehl gibt es daher in einer Version, die als Operanden



eine Konstante akzeptiert, die die Anzahl zu entfernender Bytes angibt, nachdem die Rücksprungadresse vom Stack entfernt wurde.



Es gibt Hochsprachen, bei denen der rufende Teil nicht nur die an die Routine zu übergebenden Parameter auf den Stack legt, bevor er ein CALL ausführt, sondern auch dafür verantwortlich ist, dass der Stack nach der Rückkehr wieder von diesen Parametern befreit wird. Diese Hochsprachen verwenden grundsätzlich nur den »einfachen« RET-Befehl. Dem gegenüber gibt es jedoch auch Hochsprachen, bei denen die gerufene Routine für die Säuberung des stacks verantwortlich ist. Diese Hochsprachen verwenden den parametrischen RET-Befehl. In der Regel ist bei der Entwicklung von Assembler-Modulen darauf zu achten, dass die gleichen Konventionen eingehalten werden, die die entsprechenden Hochsprachencompiler ebenfalls achten.

Operanden

Der CALL-Befehl kann somit folgende Operanden besitzen:

- Direkter, relativer near call
  `CALL Dist16; CALL Dist32`
- Indirekter, relativer near call, Zieladresse in einem Register
  `CALL Reg16; CALL Reg32`
- Indirekter, relativer near call, *effektive Adresse* des Ziels in einer Speicherstelle
  `CALL Mem16; CALL Mem32`
- Direkter, absoluter far call
  `CALL Selektor:EA16; CALL Selektor:EA32`
- Indirekter, absoluter far call; *logische Adresse* des Ziels in einer Speicherstelle
  `CALL Mem16+16; CALL Mem16+32`

Der RET-Befehl kommt in zwei Versionen vor:

- Return ohne Parameterentfernung vom Stack
  `RET`
- Return mit Entfernung von Const16 Bytes vom Stack
  `RET Const16`

Statusflags

Die Statusflags werden lediglich im Rahmen eines task switches verändert. In diesem Fall jedoch alle, weil der Zustand des Flagregisters beim letzten switch restauriert wird. Somit ist sehr wahrscheinlich, dass sich der Status der Statusflags durch den task switch ändert.

Bei allen anderen Calls sowie den RET-Befehlen bleiben die Statusflags unverändert.

SYSENTER SYSEXIT

SYSENTER und SYSEXIT ist ein Befehlspaar, das auch paarweise eingesetzt werden muss. Beide Befehle nutzen einen Mechanismus, der dazu dient, schnell und mit möglichst geringem Performance-Verlust beim Aufruf von Systemroutinen von der Anwenderebene (Privilegstufe 3) zur Kernel-Stufe (Privilegstufe 0) und zurück zu gelangen, indem auf die zeitaufwändigen Zugriffsprüfungen, die z.B. im Rahmen eines Calls über ein call gate durchgeführt werden, verzichtet wird.



Voraussetzung dafür, dass SYSENTER/SYSEXIT verwendet werden können, ist, dass der Selektor für das Privilegstufe-0-Codesegment auf ein »flaches«, 32-Bit-Codesegment von 4 GByte Größe zeigt, das die Flags execute, read, accessed und non-conforming gesetzt hat (vgl. »Codesegmente und Codesegment-Deskriptoren« auf Seite 408). Ferner muss das Privilegstufe-0-Stacksegment auf ein ebenfalls »flaches«, 32-Bit-Datensegment von 4 GByte Größe zeigen, das die Flags read/write, accessed und expansion-up gesetzt hat (vgl. »Stacksegmente« auf Seite 415 bzw. »Datensegmente und Datensegment-Deskriptoren« auf Seite 411).

Um einen »schnellen« Zugang zur Privilegstufe 0 zu erhalten, müssen vor Aufruf von SYSENTER mit Hilfe des Befehls WRMSR folgende Informationen in die dafür vorgesehenen, modellspezifischen Register (MSRs) eingetragen werden:

- SYSENTER_CS_MSR (MSR-Adresse $174): Selektor auf das anzuspringende Codesegment mit Privilegstufe 0.
- SYSENTER_ESP_MSR (MSR-Adresse $175): SYSENTER_EIP_MSR (MSR-Adresse $176): 32-Bit-Offset in dieses Segment an die Einsprungstelle, an der die Programmausführung begonnen werden soll.
- 32-Bit-Stack-Pointer, der den Stack-Rahmen beschreibt, der beim Eintritt in die Privilegstufe 0 verwendet wird.



Neben diesen Einträgen in die MSRs müssen noch weitere Bedingungen erfüllt sein! Der Prozessor verwendet den Eintrag SYSENTER_CS_MSR zu mehreren Dingen: Er dient als Selektor in die global descriptor table (GDT), an der der Deskriptor für das anzuspringende Codesegment steht. Der darauf folgende Eintrag in der GDT

(= SYSTENTER_CS_MSR + 8) muss der Deskriptor für das Stacksegment sein, das nach Eintritt in die Privilegstufe 0 verwendet wird. Diese beiden GDT-Einträge werden von SYSENTER genutzt. Weitere 8 Bytes höher (SYSENTER_CS_MSR + 16), also im unmittelbar folgenden GDT-Eintrag, muss der Deskriptor des Codesegments mit Privilegstufe 3 stehen, in das wieder zurückgesprungen werden soll, also (in der Regel!) das Segment, aus dem heraus mittels SYSENTER herausgesprungen wird. Und nochmals 8 Bytes höher (SYSENTER_CS_MSR + 24) steht der Deskriptor für das nach der Rückkehr zu benutzende Stacksegment. Diese beiden GDT-Einträge nutzt SYSEXIT. Die vier Deskriptoren müssen unbedingt vor Aufruf des Befehls SYSENTER in der GDT verzeichnet worden und valide sein!



SYSENTER ist *kein* verkappter CALL-Befehl! Daher ist es extrem wichtig, mit diesem Befehl nur Routinen aufzurufen, die nicht mit RET oder IRET abgeschlossen werden, sondern mit SYSEXIT. Andernfalls benutzt der Prozessor fälschlicherweise Daten vom aktuellen Stack als Rücksprungadressen, was mit großer Wahrscheinlichkeit zum Desaster führen wird. Da aber in der Regel nicht öffentlich ist, welche Kernel-Funktion mit SYSEXIT abgeschlossen wird, macht die Verwendung des Paares SYSENTER – SYSEXIT nur dann Sinn, wenn die Anwendungsroutine (Level 3) und die Kernel-Routine (Level 0), die hier kommunizieren, von einem Entwickler(-team) stammen oder detaillierte Informationen vorliegen. SYSENTER und SYSEXIT sind nicht dazu geeignet, die Schutzkonzepte zu umgehen!



Nicht alle Prozessoren verfügen über die Befehle SYSENTER/SYSEXIT, die erst mit dem Pentium Pro eingeführt wurden. Ob der vorliegende Prozessor den schnellen Zugriff auf Privilegstufe-0-Routinen ermöglicht, entscheidet das Bit SEP in den feature flags, die mittels des CPUID-Befehls erhalten werden können (vgl. Seite 143).

SYSENTER SYSENTER macht nun Folgendes:

- Laden des CS-Registers mit dem Selektor aus SYSENTER_CS_MSR
- Laden des EIP-Registers mit der EIP aus SYSENTER_EIP_MSR
- Laden des SS-Registers mit dem Selektor aus SYSENTER_CS_MSR + 8
- Laden des ESP-Registers mit dem ESP aus SYSENTER_ESP_MSR
- Umschalten zu Privilegstufe 0

- Löschen der Flags VM, IR und RF in EFlags
- Beginn der Befehlsausführung an der neuen Adresse CS:EIP.

Analog macht SYSEXIT Folgendes: *SYSEXIT*

- Laden des CS-Registers mit dem Selektor aus SYSENTER_CS_MSR + 16.
- Laden des EIP-Registers mit der EIP aus EDX
- Laden des SS-Registers mit dem Selektor aus SYSENTER_CS_MSR + 24.
- Laden des ESP-Registers mit dem ESP aus ECX
- Umschalten zu Privilegstufe 3
- Beginn der Befehlsausführung an der neuen Adresse CS:EIP

Operanden



ACHTUNG! In gewisser Weise ist der Mechanismus hinter SYSENTER/SYSCALL ein Aushebeln von Schutzmechanismen, die ja im protected mode nicht ohne Grund eingeführt worden sind. Das ist nur dadurch rechtfertigbar, dass die Mechanismen, die SYSENTER/SYSEXIT ermöglichen, selbst geschützt sind (modellspezifische Register!). Somit bleibt die ernüchternde Erkenntnis, dass diese Befehle nur im Rahmen des Betriebssystems eingesetzt werden können und für Otto oder Lieschen Normalprogrammierer daher keine Rolle spielen.

Operanden

SYSENTER besitzt keine expliziten Operanden. Dennoch werden für den Hin- und Rücksprung Angaben zum anzuspringenden Codesegment (CS:EIP) und zum dort zu verwendenden Stacksegment (SS:ESP) und zum Code- und Stacksegment, in das zurückgesprungen werden soll, benötigt. Diese Informationen werden implizit in den MSRs $174 bis $176, in der GDT sowie ECX und EDX übergeben. Ziel des Sprungs und zu benutzendes Stacksegment werden den modellspezifischen Registern (MSRs) entnommen (CS_{neu} = SYSENTER_CS_MSR; SS_{neu} = SYSENTER_CS_MSR + 8; EIP_{neu} = SYSENTER_EIP_MSR; ESP_{neu} = SYSENTER_ESP_MSR), ECX enthält den 32-Bit stack pointer des Stacksegments, das nach der Rückkehr mittels SYSEXIT verwendet werden soll ($ESP_{rück}$). Das dazugehörige Stacksegment wird mit Hilfe der MSR bestimmt ($CS_{rück}$ = SYSENTER_CS_MSR + 24). EDX enthält analog den 32-Bit instruction pointer des Codesegments, der zur Rückkehr mittels SYSEXIT verwendet werden soll ($EIP_{rück}$). Auch hier wird das dazugehörige Codesegment durch die MSR selektiert: $CS_{rück}$ = SYSENTER_CS_MSR + 16.

SYSEXIT hat ebenfalls keine expliziten Operanden. Die zum korrekten Ablauf notwendigen Informationen wurden beim Aufruf von SYSENTER in den MSRs $174 bis $176, in der GDT sowie ECX und EDX übergeben.

Statusflags SYSENTER und SYSEXIT verändern keine Statusflags.

Systemflags SYSENTER löscht die Systemflags VM, IR und RF.

SYSCALL SYSRET SYSCALL und SYSRET sind AMD-Entwicklungen zweier Befehle, die im Prinzip das Gleiche machen (sollen) wie SYSENTER und SYSEXIT: Beschleunigung von Zugriffen auf das Betriebssystem. Dies erfolgt, indem auf die zeitaufwändigen Zugriffsprüfungen, die z.B. beim Aufruf einer Betriebssystemroutine über ein call gate erfolgen (müssen), verzichtet wird. Im Unterschied zu SYSENTER/SYSEXIT werden durch SYSCALL/SYSRET zwar eine neue Befehlsadresse (CS:EIP) und ein neues Stacksegment (SS) für den Kernelmodus und den Rücksprung in den Usermodus gewählt, nicht aber neue Stack-Pointer-Adressen (ESP)!



SYSCALL und SYSRET sind nur auf einigen Prozessoren von AMD implementiert. Ob dies bei einem gegebenen Prozessor der Fall ist, kann mittels des CPUID-Befehls und dessen erweiterter Funktion $8001 ermittelt werden (vgl. Seite 143). Ist das Flag SCE, *system call extension*, gesetzt, stehen die Befehle zur Verfügung.

Bis zum heutigen Tage beherrscht kein Intel-Prozessor (zumindest nach meinen Informationen) diese Befehle. Daher müssen sie als nicht kompatibel eingestuft werden. Ihre Benutzung sollte daher nur dann erfolgen, wenn Kompatibilität zu Intel-Prozessoren nicht erforderlich ist. Da SYSCALL und SYSRET wie SYSENTER und SYSEXIT im Rahmen von Betriebssystemen und -Modulen eingesetzt werden, würde dies bedeuten, dass ein Betriebssystem (-Modul) nur für AMD-Prozessoren und selbst hier nur für solche geschrieben wird, die die Befehle unterstützen. Dies erscheint mir sehr unwahrscheinlich!



Voraussetzung dafür, dass SYSCALL/SYSRET verwendet werden können, ist, dass der Selektor für das Privilegstufe-0-Codesegment auf ein »flaches«, 32-Bit-Codesegment von 4 GByte Größe zeigt, das die Flags execute, read, accessed und non-conforming gesetzt hat (vgl. »Codesegmente und Codesegment-Deskriptoren« auf Seite 408). Ferner muss das Privilegstufe-0-Stacksegment auf ein ebenfalls »flaches«, 32-Bit-Datensegment von 4 GByte Größe zeigen, das die Flags read/write,

accessed und expansion-up gesetzt hat (vgl. »Stacksegmente« auf Seite 415 bzw. »Datensegmente und Datensegment-Deskriptoren« auf Seite 411).

SYSCALL/SYSRET benutzen wie SYSENTER/SYSEXIT ein modellspezifisches Register, das *SYSCALL/SYSRET target address register* (STAR-MSR). Es besitzt die Adresse $C000_0081, umfasst 64 Bit Information und enthält in den Bits 63 bis 48 den für SYSRET erforderlichen Selektor auf das Code- (CS) und Stack- (SS) Segment, das nach Rückkehr aus dem Kernelmodus (Modus 0) eingestellt werden soll, in den Bits 47 bis 32 den für SYSCALL erforderlichen Selektor für das Code- und Stacksegment im Kernelmodus und in den Bits 31 bis 0 die Zieladresse zum Sprung in den Kernelmodus. Summa: Es sind die gleichen Informationen, die auch Intel mit seinen modellspezifischen Registern übergibt. Allerdings fehlen die in ECX, EDX und SYSENTER_ESP_MSR übergebenen Werte für die Rücksprungadresse (EIP) und die Stackzeiger im Kernel- und Usermodus (ESP).



Das bedeutet: SYSCALL/SYSRET sind »näher« an den Befehlen CALL/RET, als es SYSENTER/SYSEXIT sind. CALL legt eine Rücksprungadresse auf den Stack und SYSCALL in ECX ab, die auf den dem CALL/SYSCALL-Befehl folgenden Befehl zeigt, während SYSENTER keinerlei Rücksprungangaben sichert. Hier muss das Rücksprungziel explizit angegeben und in EDX als Operand übergeben werden. Dafür kann SYSEXIT aber auch an eine andere als die dem SYSENTER-Befehl folgende Adresse zurückspringen.



Analog zu SYSENTER/SYSEXIT müssen auch bei SYSCALL/SYSRET neben den Einträgen in die MSRs noch weitere Bedingungen erfüllt sein! Der Prozessor verwendet die Bits 47 bis 32 des STAR-MSR zu mehreren Dingen: Sie dienen als Selektor in die global descriptor table (GDT), an der der Deskriptor für das anzuspringende Codesegment steht. Der darauf folgende Eintrag in der GDT (= STAR-MSR[47..32] + 8) muss der Deskriptor für das Stacksegment sein, das nach Eintritt in die Privilegstufe 0 verwendet wird. Diese beiden GDT-Einträge werden von SYSCALL genutzt. In den Bits 63 bis 48 stehen die gleichen Informationen für SYSRET: STAR-MSR[63..48] ist der Selektor in die GDT, an der der Deskriptor für das Rückkehr-Codesegment steht, der darauf folgende GDT-Eintrag (STAR-MSR[63..32] + 8) enthält den Deskriptor für das Rückkehr-Stacksegment. Diese vier Deskriptoren müssen unbe-

dingt vor Aufruf des Befehls SYSCALL in der GDT verzeichnet worden und valide sein!

SYSCALL SYSCALL macht nun Folgendes:

- Kopieren des EIP-Registerinhaltes in ECX. *ACHTUNG: Dieser Wert stellt die Rücksprungadresse dar, die von SYSRET verwendet wird. Somit ist ggf. der ECX-Inhalt nach SYSCALL zu sichern und unmittelbar vor Aufruf von SYSRET zu restaurieren, wenn ECX im Rahmen der Befehlsverarbeitung in der Kernelroutine benötigt wird.*
- Laden des CS-Registers mit dem Selektor aus STAR-MSR[47..32]
- Laden des EIP-Registers mit der EIP aus STAR-MSR[31..0]
- Laden des SS-Registers mit dem Selektor aus STAR-MSR[47..32] + 8
- Umschalten zu Privilegstufe 0
- Löschen der Flags VM, IR und RF in EFlags
- Beginn der Befehlsausführung an der neuen Adresse CS:EIP.

SYSRET Analog macht SYSRET Folgendes:

- Laden des CS-Registers mit dem Selektor aus STAR-MSR[63..48].
- Laden des EIP-Registers mit der EIP aus ECX
- Laden des SS-Registers mit dem Selektor aus STAR-MSR[63..48] + 8.
- Umschalten zu Privilegstufe 3
- Beginn der Befehlsausführung an der neuen Adresse CS:EIP

Operanden SYSCALL besitzt keine expliziten Operanden. Dennoch werden für den Hin- und Rücksprung Angaben zum anzuspringenden Codesegment (CS:EIP) und zum dort zu verwendenden Stacksegment (SS) und zum Code- und Stacksegment, in das zurückgesprungen werden soll, benötigt. Diese Informationen werden implizit im MSR $C000_0081 und in der GDT übergeben. Ziel des Sprungs und zu benutzendes Stacksegment werden dem modellspezifischen Register (MSR) entnommen (CS_{neu} = STAR-MSR[47..32]; SS_{neu} = STAR-MSR[47..32] + 8; EIP_{neu} = STAR-MSR[31..00]). Das nach der Rückkehr mittels SYSRET zu benutzende Stacksegment wird mit Hilfe des MSR bestimmt ($CS_{rück}$ = STAR-MSR[63..48] + 8). ECX enthält den durch SYSCALL geretteten 32-Bit instruction pointer des Codesegments, der zur Rückkehr mittels SYSEXIT verwendet werden soll ($EIP_{rück}$). Auch hier wird das dazugehörige Codesegment durch das MSR selektiert: $CS_{rück}$ = STAR-MSR[63..48].

SYSRET hat ebenfalls keine expliziten Operanden. Die zum korrekten Ablauf notwendigen Informationen wurden beim Aufruf von SYSCALL im MSR $C000:8001, in der GDT sowie in ECX abgelegt.

SYSCALL und SYSRET verändern keine Statusflags. *Statusflags*

SYSCALL löscht die Systemflags VM, IR und RF. *Systemflags*

1.1.8 Andere bedingte Operationen

Es gibt nicht nur die bedingten Sprungbefehle, mit denen auf eine bestimmte Situation (Bedingung) reagiert werden kann. Zumindest bei den moderneren Prozessoren gibt es auch zwei Befehle, mit denen Flaggen gesetzt oder Daten kopiert werden können, je nachdem, ob eine Bedingung erfüllt ist oder nicht.

CMOVcc

Eine Spielart des MOV-Befehls ist der bedingte MOV-Befehl, CMOVcc oder *conditional move on cc*. Mit diesem Befehl kann der Inhalt aus einem Register oder einer Speicherstelle dann und nur dann in ein Register kopiert werden, wenn die Bedingung *cc* erfüllt ist, die anhand der Stellung der Statusflags geprüft wird. Andernfalls unterbleibt das Kopieren.



Nicht alle Prozessoren verfügen über den CMOVcc-Befehl, der erst mit dem Pentium Pro eingeführt wurde. Ob der Befehl implementiert ist, lässt sich mittels des CPUID-Befehls feststellen. Falls das CMOV-Flag (Bit 15 der feature flags) gesetzt ist, wird CMOVcc unterstützt.

Aufgrund der Auswertung der Statusflags, gibt es gemäß der in Tabelle 1.1 auf Seite 43 dargestellten Möglichkeiten der Flagprüfung folgende bedingte MOV-Befehle:

Befehl	MOV, wenn	Synonyme	Prüfung
CMOVA	größer	CMOVNBE	CF=0 & ZF=0
CMOVAE	größer, gleich	CMOVNB, CMOVNC	CF=0
CMOVB	kleiner	CMOVNAE, CMOVC	CF=1
CMOVBE	kleiner, gleich	CMOVNA	CF=1 | ZF = 1
CMOVC	carry gesetzt		CF=1
CMOVE	gleich	CMOVZ	ZF= 1
CMOVG	größer (±)	CMOVNLE	OF=SF & ZF=0

Tabelle 1.7: CMOVcc-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags

Befehl	MOV, wenn	Synonyme	Prüfung
CMOVGE	größer, gleich (±)	CMOVNL	OF=SF
CMOVL	kleiner (±)	CMOVGE	OF≠SF
CMOVLE	kleiner, gleich (±)	CMOVG	OF≠SF | ZF=1
CMOVNA	nicht größer	CMOVBE	CF=1 | ZF = 1
CMOVNAE	nicht größer, gleich	CMOVB	CF=1
CMOVNB	nicht kleiner	CMOVAE	CF=0
CMOVNBE	nicht kleiner, gleich	CMOVA	CF=0 & ZF=0
CMOVNC	carry gelöscht		CF=0
CMOVNE	nicht gleich	CMOVNZ	ZF=0
CMOVNG	nicht größer (±)	CMOVLE	OF≠SF | ZF=1
CMOVNGE	nicht größer, gleich (±)	CMOVL	OF≠SF
CMOVNL	nicht kleiner (±)	CMOVGE	OF=SF
CMOVNLE	nicht kleiner, gleich (±)	CMOVG	OF=SF & ZF=0
CMOVNO	overflow gelöscht		OF=0
CMOVNP	parity gelöscht		PF=0
CMOVNS	sign gelöscht		SF=0
CMOVNZ	zero gelöscht	CMOVNE	ZF=0
CMOVO	overflow gesetzt		OF=1
CMOVP	parity gesetzt		PF=1
CMOVPE	parity gesetzt		PF=1
CMOVPO	parity gelöscht		PF=0
CMOVS	sign gesetzt		SF=1
CMOVZ	zero gesetzt	CMOVE	ZF=1

Tabelle 1.7: CMOVcc-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags (Forts.)

Operanden Die CMOVcc-Befehle haben als Zieloperanden immer ein Allzweckregister. Der Quelloperand kann entweder ein Allzweckregister oder eine Speicherstelle sein, sodass folgende Operandenkombinationen möglich sind:

- Bedingtes Kopieren des Inhalts eines Registers in ein Register
  `CMOVcc Reg16, Reg16; CMOVcc Reg32, Reg32`
- Bedingtes Kopieren des Inhalts einer Speicherstelle in ein Register
  `CMOVcc Reg16, Mem16; CMOVcc Reg32, Mem32`



Bitte beachten Sie, dass die in Tabelle 1.7 aufgeführten 30 Befehle durch »nur« 16 Opcodes realisiert werden. Der Grund hierfür ist, dass bei acht Befehlen semantische Redundanzen vorliegen (»größer« ist identisch mit »nicht kleiner oder gleich«), vier Befehle mit unterschiedlichen

Mnemonics benutzt werden können (»gleich« und »zero« prüfen das zero flag) und zwei weitere Befehle sogar mit zwei weiteren, redundant vorliegenden Befehlen identisch sind (genauer: das gleiche Flag abprüfen; »above or equal« ist identisch mit »not below« und prüft wie »carry« das carry flag). Somit werden mit 12 Opcodes bereits 26 Mnemonics realisiert. Die verbleibenden vier Opcodes haben jeweils ihr »eigenes« Mnemonic. Vergleiche hierzu auch Tabelle 1.1 auf Seite 43.

Statusflags

Die Statusflags werden durch CMOVcc nicht verändert.

SETcc

Neben den bedingten Programmverzweigungen (Jcc) und dem bedingten Kopieren von Daten (CMOVcc) gibt es auch die Möglichkeit, eine Flagge bedingt zu setzen. Dies ermöglichen die bedingten Befehle SETcc, die die Statusflags zur Entscheidungsfindung heranziehen. Somit gibt es gemäß der in Tabelle 1.1 auf Seite 43 dargestellten Möglichkeiten der Flagprüfung folgende bedingten »Setz-Befehle«:

Befehl	SET, wenn	Synonyme	Prüfung
SETA	größer	SETNBE	CF=0 & ZF=0
SETAE	größer, gleich	SETNB, SETNC	CF=0
SETB	kleiner	SETNAE, SETC	CF=1
SETBE	kleiner, gleich	SETNA	CF=1 | ZF = 1
SETC	carry gesetzt		CF=1
SETE	gleich	SETZ	ZF= 1
SETG	größer (±)	SETNLE	OF=SF & ZF=0
SETGE	größer, gleich (±)	SETNL	OF=SF
SETL	kleiner (±)	SETGE	OF≠SF
SETLE	kleiner, gleich (±)	SETG	OF≠SF | ZF=1
SETNA	nicht größer	SETBE	CF=1 | ZF = 1
SETNAE	nicht größer, gleich	SETB	CF=1
SETNB	nicht kleiner	SETAE	CF=0
SETNBE	nicht kleiner, gleich	SETA	CF=0 & ZF=0
SETNC	carry gelöscht		CF=0
SETNE	nicht gleich	SETNZ	ZF=0
SETNG	nicht größer (±)	SETLE	OF≠SF | ZF=1
SETNGE	nicht größer, gleich (±)	SETL	OF≠SF
SETNL	nicht kleiner (±)	SETGE	OF=SF
SETNLE	nicht kleiner, gleich (±)	SETG	OF=SF & ZF=0

Tabelle 1.8: Bedingte SET-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags

Befehl	SET, wenn	Synonyme	Prüfung
SETNO	overflow gelöscht		OF=0
SETNP	parity gelöscht		PF=0
SETNS	sign gelöscht		SF=0
SETNZ	zero gelöscht	SETNE	ZF=0
SETO	overflow gesetzt		OF=1
SETP	parity gesetzt		PF=1
SETPE	parity gesetzt		PF=1
SETPO	parity gelöscht		PF=0
SETS	sign gesetzt		SF=1
SETZ	zero gesetzt	SETE	ZF=1

Tabelle 1.8: Bedingte SET-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags (Forts.)

Ist die Bedingung erfüllt, so wird das als Operand übergebene Byte auf den Wert »1« gesetzt, andernfalls auf »0«.

Operanden Die SETcc-Befehle erwarten lediglich einen Zieloperanden. Bei diesem Operanden handelt es sich um ein Byte (!), das in einem Byte-Register oder einer Byte-Variablen stehen kann:

- Setzen/Löschen der Byte-Flagge in einem Register
  `SETcc Reg8`
- Setzen/Löschen der Byte-Flagge in einer Speicherstelle
  `SETcc Mem8`

Statusflags Statusflags werden von SETcc nicht verändert.

1.1.9 Programmunterbrechungen durch Interrupts/Exceptions



Einzelheiten zu Interrupts und Exceptions und was sie von einander unterscheidet, finden Sie in Kapitel »Exceptions und Interrupts« auf Seite 486.

Man unterscheidet zwei Arten von Interrupts: Hardware-Interrupts und Software-Interrupts. Letztere können von der Software ausgelöst werden, indem die vom Prozessor hierzu zur Verfügung gestellten Befehle genutzt werden.

Das Auslösen eines Softwareinterrupts ist sehr ähnlich der Programmunterbrechung mit einem Far-CALL-Befehl. Das bedeutet, dass der

Prozessor analog zum CALL-Befehl die auf den INT-Befehl folgende Adresse als Rücksprungadresse auf den Stack legt, bevor er zur Zieladresse verzweigt. Die angesprungene Prozedur, der »Interrupthandler«, muss also durch einen RET-analogen Befehl (»*return from interrupt handler*«, IRET) abgeschlossen werden, der die Rücksprungadresse vom Stack liest und in CS:(E)IP einträgt, was man als »Rücksprung« bezeichnet. Doch es gibt drei gravierende Unterschiede zu einem Far-CALL-Aufruf:

- Es wird keine Zieladresse übergeben, zu der analog zum Far-CALL-Befehl gesprungen werden kann, sondern eine »Interrupt-Nummer«. Diese muss erst in eine Adresse umgerechnet werden, was der Prozessor jedoch selbstständig macht.
- Der Prozessor rettet den Inhalt des EFlags-Registers auf den Stack, bevor die Rücksprungadresse dort abgelegt wird. Der den Interrupthandler abschließende IRET-Befehl muss daher im Rahmen des Rücksprungs auch diesen EFlags-Inhalt vom Stack nehmen und in das Register zurückschreiben.
- Einem Interrupthandler, der über den INT-Befehl angesprungen wird, kann über den Stack kein Parameter übergeben werden. Somit verfügt der IRET-Befehl anders als der RET-Befehl nicht über einen optionalen Parameter.

Interrupts sind somit eine »spezielle Form« des Unterprogrammaufrufs, die sich dadurch auszeichnet, dass sie unabhängig vom aktuellen Programm systemweit und durch genau festgelegte Randbedingungen erfolgt. Interrupts eignen sich daher besonders gut für Systemdienste, die auch Anwendungsprogrammen nutzbar gemacht werden sollen.



Beachten Sie bitte, dass die Interrupts mit den Nummern 00h bis 1Fh durch Intel reserviert und nur die Nummern 20h bis FFh »frei« verfügbar sind. Frei verfügbar heißt in diesem Zusammenhang: durch das Betriebssystem. Denn Sie können als Anwendungsprogrammierer zwar mittels der Interrupt-Befehle des Prozessors diese Interrupts nutzen, also die entsprechenden Handler aufrufen, nicht aber eigene Interrupthandler programmieren und einbinden. Die Tage des guten, alten DOS mit dem eigenmächtigen »Verbiegen« von Interruptvektoren sind endgültig vorbei!



Obwohl Sie mit dem INT-Befehl Systemdienste des Betriebssystems aufrufen oder bestimmte Exceptions auslösen können, sollten Sie das nicht tun (Ausnahme: INTO)! Zum einen müssen Sie genau Bescheid wissen, welcher Dienst sich hinter welcher Interrupt-Nummer verbirgt, was selten der Fall sein dürfte. Zum anderen werden die meisten und wichtigsten Systemdienste durch dokumentierte Funktionen des Betriebssystems zur Verfügung gestellt, sodass sich die Auslösung des Interrupts selbst in der Regel erübrigt. Drittens kann die falsche Nutzung von Interrupts zu großen Problemen führen. Viertens ist es höchst wahrscheinlich, dass Sie nicht über die erforderlichen Privilegien verfügen, die entsprechenden Interrupt-Handler zu nutzen. Und fünftens gibt es Exceptions und Interrupts, die nur im Rahmen von bestimmten Betriebssystemteilen Sinn machen (Debugger-Exceptions, page faults, segment not present etc.) Das heißt: Sie machen sich in der Regel nur Probleme.

INT
IRET
IRETD

INT ist der allgemeine Befehl zur Auslösung von Software-Interrupts. Wie eben beschrieben, legt er zunächst den Inhalt des Flagregisters und anschließend die auf den INT-Befehl folgende Adresse als Rücksprungadresse für den IRET-Befehl auf den Stack. Im real mode oder im virtual 8086 mode wird nun der als Operand übergebene Byte-Wert als Index in die *interrupt vector table* (IVT) interpretiert. Der an der entsprechenden Stelle stehende Wert stellt die Adresse des Interrupthandlers dar, der nun angesprungen wird. Im protected mode stellt das als Operand übergebene Byte den Index in die *interrupt descriptor table* dar. Der dort verzeichnete Deskriptor wird ausgelesen und entsprechend der Art der enthaltenen Information die Zieladresse bestimmt. Einzelheiten finden Sie in »Interrupt-Behandlung« ab Seite 489.



IRET und IRETD haben denselben Opcode. IRET wird benutzt, wenn die Umgebung 16-bittig ist, IRETD im Falle einer 32-Bit-Umgebung. Sie braucht das nicht zu interessieren, da die meisten Assembler IRET in beiden Umgebungen akzeptieren und entsprechend umsetzen. IRETD ist somit obsolet! Allerdings kann es sein, dass einige Disassembler, wie sie z.B. durch Debugger benutzt werden, je nach Umgebung den Opcode $CF als IRET bzw. IRETD darstellen.

Operanden

Der INT-Befehl akzeptiert nur eine 8-Bit-Konstante als Operanden:

```
INT Const8
```

Dieser Wert wird als vorzeichenloser Index in die IDT (protected mode; interrupt descriptor table) oder IVT (real mode; interrupt vector table) interpretiert.

Der IRET-/IRETD-Befehl besitzt keine Operanden.

Statusflags

Die Statusflags werden durch INT nicht verändert. Allerdings können in Abhängigkeit des Betriebsmodus einige Systemflags (IF, TF, NT, AC, RF, VM) gelöscht werden. Da aber das gesamte EFlags-Register auf den Stack kopiert und nach Rückkehr vom Interrupthandler restauriert wird, machen sich diese Veränderungen lediglich im Interrupthandler selbst, nicht aber im unterbrochenen Programm bemerkbar.

Im Falle von IRET/IRETD dagegen werden alle Flags des EFlags-Registers verändert, da IRET die auf dem Stack liegende Kopie des Inhalts des EFlags-Registers in das Register kopiert. Somit werden alle Änderungen, die innerhalb des Handlers an den Flags des EFlags-Registers vorgenommen werden, rückgängig gemacht.

INT0
INT3

INTO und INT3 sind Mnemonics für einen Befehl, der Interrupt #4 bzw. Interrupt #3 auslöst. INT3 ist der »Debugger-Interrupt« #3, der zur Realisierung von »Breakpoints« herangezogen werden kann. INTO ist der »Overflow-Interrupt« #4, der einen Handler aufruft, wenn ein arithmetischer Überlauf stattgefunden hat. Allerdings gibt es Unterschiede bei der Interrupt-Auslösung via INTO bzw. INT3 im Vergleich zu INT 04h bzw. INT 03h:

INT3

INT3 besitzt einen Ein-Byte-Opcode ($CC). Damit unterscheidet er sich von der Zwei-Byte-Version, die mittels INT 03h ($CD03) codiert würde. Wichtig ist dieser Unterschied, da die Ein-Byte-Version auch bei Ein-Byte-Codes als Breakpoint benutzt werden kann, während hier die Zwei-Byte-Form das erste Byte des folgenden Befehls überschreiben würde. Aus diesem Grunde übersetzen auch alle mir bekannten Assembler den Befehl INT 03 in den Opcode für INT3. Der Opcode für INT 03h muss, falls man ihn absolut brauchte, »von Hand« codiert werden.

INTO

INTO, *interrupt on overflow,* prüft zunächst das overflow flag. Ist es gesetzt, wird ein INT 04h ausgelöst, andernfalls nicht. INTO kann somit als *Bedingter Interrupt* aufgefasst werden, dessen Auslösung an die Stellung eines Statusflags gebunden ist. Anders als die anderen bedingten Befehle gibt es jedoch keine INTcc-Version.

Operanden

INTO und INT3 haben keine Operanden.

Statusflags Die Statusflags werden durch INT3 / INTO nicht verändert. Allerdings können in Abhängigkeit des Betriebsmodus einige Systemflags (IF, TF, NT, AC, RF, VM) gelöscht werden. Da aber das gesamte EFlags-Register auf den Stack kopiert und nach Rückkehr vom Interrupthandler restauriert wird, machen sich diese Veränderungen lediglich im Interrupthandler selbst, nicht aber im unterbrochenen Programm bemerkbar.

BOUND Auch der Befehl BOUND hat mit Interrupts zu tun, auch wenn er mit einer Exception verknüpft ist, denn Exceptions sind auch nichts anderes als Interrupts.

Bound prüft, ob ein als Parameter übergebener Wert innerhalb der Grenzen liegt, die ebenfalls als Parameter übergeben werden. Ist das der Fall, erfolgt gar nichts und der Prozessor fährt mit dem nächsten Befehl fort. Liegt der Wert jedoch außerhalb der Grenzen, wird die bound range exceed exception #BR (INT 05h) ausgelöst.



Im Unterschied zu anderen Interrupts wird bei BOUND als Rücksprungadresse die Adresse des BOUND-Befehls selbst auf den Stack gelegt. Das bedeutet, dass nach Rückkehr aus dem Interrupthandler erneut der BOUND-Befehl ausgeführt wird. Wurde die zum Interrupt führende Verletzung der Feld-Grenzen nicht im Handler behoben, wird dadurch erneut eine bound range exceeded exception #BR (INT 05h) ausgelöst.

Operanden BOUND hat zwei Operanden: ein Register, in dem der zu prüfende Wert übergeben wird, und einen Speicheroperanden, in dem die zwei Grenzen übergeben werden:

```
BOUND Reg16, Mem16+16; BOUND Reg32, Mem32+32
```

Der im ersten Operanden übergebene Prüfwert ist eine vorzeichenbehaftete Integer, die als Index in ein Feld interpretiert wird. Der zweite Operand enthält jeweils die vorzeichenbehafteten Grenzen des Feldes. Der erste Wert an der Speicheradresse enthält hierbei die untere, der zweite Wert die obere Grenze.

Statusflags Die Statusflags werden durch BOUND nicht verändert. Allerdings können in Abhängigkeit des Betriebsmodus einige Systemflags (IF, TF, NT, AC, RF, VM) gelöscht werden. Da aber das gesamte EFlags-Register auf den Stack kopiert und nach Rückkehr vom Interrupthandler restauriert wird, machen sich diese Veränderungen lediglich im Interrupthandler selbst, nicht aber im unterbrochenen Programm bemerkbar.

1.1.10 Instruktionen zur gezielten Veränderung des Flagregisters

Da in diesem Abschnitt auch die Kommunikation mit dem Stack angesprochen wird, empfiehlt es sich, das Kapitel »Stack« auf Seite 385 durchgelesen zu haben.



PUSHF
POPF
PUSHFD
POPFD

PUSHF/PUSHFD, *push flags* bzw. *push flags as double word*, sind zwei Befehle, die den Inhalt des EFlags-Registers auf den Stack schieben und somit spezielle Implementationen des PUSH-Befehls darstellen (vgl. Seite 94). Analog sind POPF/POPFD, *pop flags* bzw. *pop flags as double word*, die Spezialimplementationen des POP-Befehls für das EFlags-Register.



Analog PUSHA/PUSHAD und POPA/POPAD (vgl. Seite 97) sind PUSHFD und POPFD Alias von PUSHF und POPF. Manche Assembler erzwingen bei Verwendung von PUSHF/POPF Befehlssequenzen für 16-Bit-Register, bei PUSHFD/POPFD für 32-Bit-Register. In der Regel aber werden die Assembler die korrekten Befehlssequenzen anhand der aktuellen Umgebung setzen.

PUSHF dekrementiert den Stackpointer (E)SP um 2 bzw. 4 (je nach Umgebung) und kopiert den Inhalt des Flags/EFlags-Registers dorthin. Beim Kopieren jedoch werden die Bits der Kopie, die den Flags VM und RF entsprechen, auf Null gesetzt. POPF inkrementiert den Stackpointer um 2 bzw. 4, nachdem es den an (E)SP stehenden geretteten Registerinhalt wieder in das Flags/EFlags-Register kopiert hat.



Bei POPF/POPFD gibt es leichte Unterschiede anhand des aktuellen Betriebsmodus. Im real mode oder im protected mode mit Privilegstufe 0 können alle nicht reservierten Flags außer VIP, VIF und VM verändert werden. VIP und VIF werden nach POPF/POPFD gelöscht, VM wird nicht verändert. Im protected mode mit Privilegstufen größer Null und kleiner oder gleich IOPL kann das IOPL-Feld ebenfalls nicht verändert werden. IF wird dann nur verändert, wenn die Privilegstufe kleiner als IOPL ist. Im virtual 8086 mode muss IOPL den Wert 3 haben, damit POPF/POPFD nicht eine general protection exception #GP auslöst. In diesem Fall bleiben die Flags VM, RF, VIP und VIF sowie das Feld IOPL unverändert.



Die Kombination PUSHF/POPF bzw. deren »D«-Varianten können dazu benutzt werden, Flags gezielt zu verändern, die anders nicht verändert werden können. Hierzu kann z.B. folgende Befehlssequenz verwendet werden:

```
PUSHF       ; EFlags-Registerinhalt auf den Stack und
POP  EAX    ; von dort in das EAX-Register laden
:           ; hier können Befehle stehen, die einzelne
:           ; Bits verändern, z.B. mit Hilfe von Masken
:           ; und den Befehlen AND, XOR bzw. OR
PUSH EAX    ; Rückweg: EAX auf den Stack und
POPF        ; zurück ins EFlags-Register
```

Bitte beachten Sie hierbei zweierlei:

- Das Verändern reservierter Bits im EFlags-Register, das über diese Methode möglich ist, kann zu unerwünschten und unvorhersehbaren Ergebnissen führen und sollte daher unterbleiben.
- Nicht alle Flags sind auf diese Weise veränderbar. So können VIP, VIF und VM auf diese Weise nicht verändert werden und IOPL nur, wenn die erforderlichen Privilegien vorliegen.

Operanden PUSHF/PUSHFD und POPF/POPFD haben keine Operanden. Quelle für PUSHF/PUSHFD ist implizit das EFlags-Register, Ziel der Stack an der Position SS:(E)SP + 4 (2). Umgekehrt ist Quelle bei POPF/POPFD implizit SS:(E)SP, Ziel das EFlags-Register.

Statusflags Alle Statusflags, das Kontrollflag und weitere Flags des EFlags-Registers werden durch POPF/POPFD anhand des zurückgespeicherten Wertes verändert, PUSH/PUSHFD dagegen verändert die Statusflags, das Kontrollflag und alle Systemflags nicht.

LAHF
SAHF Load status flags in AH register, LAHF, benutzen die Bits 7, 6, 4, 2 und 0, um das sign flag (Bit 7 des EFlags-Registers), zero flag (Bit 6), adjust flag (Bit 4), parity flag (Bit 2) und carry flag (Bit 0) in ein Code-Byte im AH-Register zu kopieren. Die Bits 5, 3 und 1 bleiben unberücksichtigt, Bit 1 im Code-Byte wird gesetzt, der Rest gelöscht. Umgekehrt wird durch store AH register into flags, SAHF, aus den Bits 7 (sign flag), 6 (zero flag), 4 (adjust flag), 2 (parity flag) und 0 (carry flag) des Code-Bytes in AH die entsprechenden Flags im EFlags-Register gesetzt. Die Bits 5, 3 und 1 des Codeworts in AH bleiben unberücksichtigt.

LAHF kann in Verbindung mit FPU-Befehlen dazu genutzt werden, den Status nach FPU-Vergleichen in das EFlags-Register zu kopieren und somit eine Voraussetzung für Programmverzweigungen zu schaffen.



Operanden

LAHF und SAHF haben nur implizite Operanden: das EFlags- und das AH-Register.

Statusflags

LAHF verändert die Statusflags nicht, durch SAHF werden sie gemäß dem Code-Byte in AH gesetzt.

CLC
STC
CMC

Clear carry flag, CLC, *set carry flag*, STC, und *complement carry flag*, CMC, sind drei Befehle, die das carry flag explizit löschen, setzen oder »umdrehen«. Mehr ist dazu wirklich nicht zu sagen!

CLD
STD

Analog CLC und STC kann mit CLD, *clear direction flag*, und STD, *set direction flag*, das einzige Kontrollflag des Prozessors, das direction flag, gezielt gelöscht oder gesetzt werden. Zur Bedeutung des direction flags siehe Abschnitt »Operationen mit »Strings«« auf Seite 127.

CLI
STI

Gleiches ist mit dem interrupt enable flag IF möglich: STI, *set interrupt enable flag*, setzt es, CLI, *clear interrupt enable flag*, löscht es explizit. Eine weitere Besprechung der Bedeutung des interrupt enable flags erfolgt im Rahmen dieses Buches nicht, da diese Befehle nur innerhalb von Interrupt-Handlern Sinn machen, das weitere Auftreten von Interrupts zu unterdrücken (CLI) oder wieder zuzulassen (STI). Interrupt-Handler sind aber nicht Gegenstand dieses Buches.

1.1.11 Operationen mit »Strings«



Zum besseren Verständnis der Arbeitsweise der in diesem Abschnitt besprochenen Befehle empfiehlt es sich, das Kapitel »Zugriffe auf den Speicher: Von Adressen und Adressräumen« ab Seite 434 durchgelesen zu haben.

Vergessen Sie alles, was Sie in Hochsprachen einmal über Strings gelernt haben! Strings sind unter Assembler ganz allgemein eine Reihe gleicher Daten. Daher gibt es Byte-Strings, Word-Strings und DoubleWord-Strings. In Hochsprachen würde man sie als »eindimensionale Felder« aus Bytes, Words oder DoubleWords bezeichnen. Strings können daher ASCII- oder ANSI-Zeichen (Bytes) bzw. Unicode (Words) enthalten, müssen aber nicht. Sie können auch Zahlen oder Bitfelder aufnehmen. Im Gegenteil: Wie man an einigen Befehlen sehen kann

(SCAS und CMPS), gelten die in Strings verzeichneten Daten als Zahlen. So benutzen beide Befehlsgruppen den arithmetischen CMP-Vergleich.

Strings sind unter Assembler deshalb etwas Besonderes, da mit ihnen eine besonders einfache Verarbeitung der gleichen Daten eines Strings möglich ist. So gibt es z.B. Repetierbefehle, die ähnlich einer Schleife immer den gleichen Befehl auf die einzelnen Daten der »Felder« anwenden, nur sind sie sehr viel effizienter und damit schneller.

Die einzelnen String-Befehle werden im Anschluss behandelt. Hier folgt eine Darstellung der allgemeinen Eigenschaften der Stringbefehle.

Operanden

Stringbefehle haben keine expliziten Operanden, da die logischen Adressen von Quell- und/oder Ziel-String jeweils über bestimmte Registerkombinationen verwaltet werden. Das bedeutet, dass die zu verwendenden Datengrößen anders festgelegt werden müssen. Daher gibt es für jeden Stringbefehl, den der Prozessor kennt, drei Mnemonics, deren letzter Buchstabe die zu verwendende Datengröße angibt. So werden z.B. beim Laden von Daten aus Strings (LODS, *load from string*) mit LOD*SB* Strings *B*yte-weise geladen, mit LOD*SW W*ord-weise und mit LOD*SD D*oubleWord-weise. Dennoch gibt es auch die »parametrische« Form des Mnemonics, also die, die keine Angaben über die Datengröße macht (z.B. LODS).



Lassen Sie sich durch die Anzahl der verschiedenen Mnemonics für den gleichen Befehl nicht blenden! Es gibt für jeden Stringbefehl eigentlich nur zwei Opcodes: einen für Byte-weise Verarbeitung und einen zweiten für Word- bzw. DoubleWord-weise Verarbeitung. Bei Letzterem entscheidet wiederum die Umgebung, in der der Befehl ausgeführt wird, inwieweit die Befehlssequenz durch einen Präfix ergänzt wird: In 32-Bit-Umgebungen (32-Bit-Prozessor und 32-Bit-Betriebssystem) sind 32-Bit-Daten Standard, sodass der jeweilige Opcode für die DoubleWord-Version zuständig ist. Bei Nutzung der Word-Version kommt dann der operand size override prefix zum Einsatz. In 16-Bit-Umgebungen (16-/32-Bit-Prozessoren mit 16-Bit-Betriebssystem) dagegen sind 16-Bit-Daten Standard, sodass der jeweilige Opcode für die Word-Version verwendet wird. Für die DoubleWord-Version wird in diesem Fall der operand size override prefix verwendet. Einzelheiten hierzu finden Sie im Abschnitt »Adress- und Operandengrößen« auf Seite 765. Zum Stichwort »Adressen« finden Sie zusätzliche Informationen im Abschnitt »Beziehungskisten: Von der effektiven zur logischen Adresse« ab Seite 435.

Alle Befehle, die ein Datum aus dem String auslesen (LODSx, MOVSx, CMPSx, SCASx, OUTSx), benutzen als Quelle die logische Adresse, die durch die Kombination DS:ESI (in 32-Bit-Umgebungen) bzw. DS:SI (in 16-Bit-Umgebungen) referenziert wird. Soweit das ausgelesene Datum in ein Register kopiert wird, ist das grundsätzlich der Akkumulator (AL, AX, EAX). Die Befehle, die ein Datum in einen String speichern (STOSx, MOVSx, INSx), speichern es in die Adresse, die durch die Kombination ES:EDI bzw. ES:DI referenziert wird. (Hier haben Sie ein weiteres Beispiel für die Spezialisierung einiger Allzweckregister: (E)SI ist das source index register, (E)DI das destination index register!). Befehle, die mit Ports kommunizieren, benutzen darüber hinaus das DX-Register zur Adressierung des gewünschten Ports, der Befehl SCASx den Akkumulator für den zu vergleichenden Wert.



Nach jedem Stringbefehl wird/werden die benutzten Adressen aktualisiert. Hierzu wird zu den in den Registerkombinationen DS:(E)SI bzw. ES:(E)DI stehenden Adressen jeweils die Größe des Datums addiert/subtrahiert, sodass ein erneuter Aufruf des Stringbefehls automatisch mit dem korrekten Datum erfolgt: Nach jedem Stringbefehl zeigen die Adressregister auf das jeweils nächste zu verarbeitende Datum. In Verbindung mit Repetierbefehlen (REP bzw. REPcc) lassen sich damit sehr schnelle und effektive Manipulationen von Strings erreichen. Ob die Adressanpassung dabei »vorwärts« (Addition der Datumsgröße zur logischen Adresse) oder »rückwärts« (Subtraktion) erfolgt, entscheidet die Stellung des direction flags im EFlags-Register (DF = 0: vorwärts; DF = 1: rückwärts).



Die »parametrische« Form der String-Befehle erwartet ein oder zwei explizit angegebene Operanden. Ihr muss auf Assemblerebene *formal* ein Quell- und/oder Zieloperand übergeben werden, der nur dazu dient, dem Assembler die Größe der verwendeten Daten mitzuteilen. Dieser übersetzt dann den »parametrischen« Befehl in den jeweils benötigten »parameterfreien« Stringbefehl.

Die parametrische Form kann daher folgende Operanden nutzen:

```
CMPS Mem8, Mem8; CMPS Mem16, Mem16; CMPS Mem32, Mem32
INS  Mem8, DX;   INS  Mem16, DX;    INS  Mem32, DX
LODS Mem8;       LODS Mem16;        LODS Mem32
MOVS Mem8, Mem8; MOVS Mem8, Mem8;   MOVS Mem32, Mem32
OUTS DX, Mem8;   OUTS DX, Mem16;    OUTS DX, Mem32
SCAS Mem8;       SCAS Mem16;        SCAS Mem32
STOS Mem8;       STOS Mem16;        STOS Mem32
```

Beachten Sie hierbei bitte, dass die Operanden Mem8, Mem16 und Mem32 lediglich Dummies sind, mit deren Hilfe angegeben wird, mit welchen Daten es der Befehl zu tun hat (Bytes, Words, DoubleWords). *Das Datum selbst bzw. seine Adresse spielt hierbei keine Rolle!* Das bedeutet, Sie können jedes geeignete Datum als Parameter übergeben, ohne Gefahr zu laufen, dass es verändert wird! Tatsächlich herangezogen werden dann jeweils die Adressen, die vorher sowieso in die Registerkombinationen DS:(E)SI und/oder ES:(E)DI einzutragen sind.



Kann mir mal jemand erklären, warum es die »parametrischen« Formen überhaupt gibt? Intel selbst nicht, da zumindest mich die Begründung der Existenz nicht befriedigt: »*This explicit-operands form is provided to allow documentation*«. Und so gibt denn Intel auch zu: »*However, note that the documentation provided by this form can be misleading.*«

Wenn man die Operanden nur dazu benutzt, dem Assembler mitzuteilen, ob der einen XXX-Befehl in XXXB, XXXW oder XXXD zu übersetzen hat, kann man ja gleich den entsprechenden parameterlosen Befehl nehmen! Daher mein Tipp: Vergessen Sie einfach die parametrischen Formen der Befehle, Sie verhindern dadurch schwer aufzufindende Programmierfehler, die daraus resultieren, dass bei oberflächlicher Betrachtung die korrekte Nutzung der übergebenen Operanden vorgegaukelt und vergessen wird, die Registerkombinationen DS:(E)SI und/oder ES:(E)DI korrekt zu beladen! Und exakt dokumentieren kann man auch anders!



Bei Befehlen, die sich auf DS:ESI bzw. DS:SI beziehen, kann mit Hilfe eines segment override prefix ein anderes als das DS-Register als Bezug für die Adressberechnung herangezogen werden. Bei Befehlen, die ES:EDI bzw. ES:DI benutzen, ist ein segment override nicht möglich! Bei Befehlen, die beide Registerkombinationen verwenden (CMPSx, MOVSx), wird mit einem segment prefix override immer das Standard-Datensegment (DS) umdefiniert.

Statusflags

INSx, LODSx, MOVSx, OUTSx und STOSx verändern keine Statusflags. CMPSx und SCASx dagegen setzen die Statusflags analog einem CMP-Befehl, sodass im Anschluss mit Hilfe von bedingten Befehlen eine Programmverzweigung erfolgen kann.



Beachten Sie hierbei, dass die Repetier-Präfixe, die bei diesen Befehlen erlaubt sind (REPE/REPZ/REPNE/REPNZ), allerdings nur das zero flag prüfen! Das bedeutet z.B., dass der Präfix REPE in Verbindung mit

SCASD verwendet wird, um die Stelle im String zu finden, an der der Testwert *nicht* steht – mit den bedingten Befehlen kann dann ausgewertet werden, ob das Datum kleiner oder größer als das Testdatum ist. Achten Sie darauf, hier keine Ungereimtheiten zu programmieren!

Kontrollflag

Die String-Befehle sind die einzigen Befehle, die Verwendung vom einzigen Kontrollflag des Prozessors machen! So bestimmt das direction flag im EFlags-Register, in welcher Richtung die Strings bearbeitet werden. Ist es gesetzt, so wird »rückwärts« (von hohen zu niedrigen Adressen) gearbeitet: Die Indexregister werden nach der Operation jeweils um die Operandengröße (Byte, Word, DoubleWord) *dekrementiert*. Ist es dagegen gelöscht, so wird »vorwärts« (von niedrigen zu hohen Adressen) gearbeitet und die Indexregister werden um die entsprechenden Beträge *inkrementiert*!



Beachten Sie bitte, dass das Kontrollflag immer für beide Strings gilt, wenn ein Stringbefehl mit zwei Strings arbeitet (CMPS, MOVS). Leider ist es nicht möglich, einen String »von vorne nach hinten« und den anderen »von hinten nach vorne« durchzuarbeiten.

LODS
LODSB
LODSW
LODSD

Diese Gruppe von Befehlen, *load from string by byte/word/double word*, ist dafür zuständig, ein Datum aus einem String in den Akkumulator zu laden. Je nach Datengröße ist dies das AL-Register (LODSB), das AX-Register (LODSW) oder das EAX-Register (LODSD). Quelle ist ein String, dessen auszulesende Adresse in DS:ESI (32-Bit-Umgebungen) bzw. DS:SI (16-Bit-Umgebungen) verzeichnet ist. Nach der Operation wird der Inhalt von ESI/SI um 1 (LODSB), 2 (LODSW) bzw. 4 (LODSD) inkrementiert bzw. dekrementiert, sodass ein erneuter Aufruf des Befehls sofort das nächste Datum auslesen kann.

STOS
STOSB
STOSW
STOSD

Diese Befehlsgruppe, *store to string by byte/word/double word*, speichert ein Datum aus dem Akkumulator in einen String. Je nach Datengröße wird es aus dem AL-Register (STOSB), dem AX-Register (STOSW) oder dem EAX-Register (STOSD) entnommen und in den String kopiert, dessen Adresse in ES: EDI (32-Bit-Umgebungen) bzw. ES:DI (16-Bit-Umgebungen) verzeichnet ist. Nach der Operation wird der Inhalt von EDI/DI um 1 (STOSB), 2 (STOSW) bzw. 4 (STOSD) inkrementiert bzw. dekrementiert, sodass ein erneuter Aufruf des Befehls sofort das nächste Datum abspeichern kann.

MOVS
MOVSB
MOVSW
MOVSD

Diese Gruppe von Befehlen, *move string by byte/word/double word*, kopiert ein Datum aus einem String in einen anderen. Je nach Datengröße erfolgt das Byte-weise (MOVSB), Word-weise (MOVSW) oder DoubleWord-weise (MOVSD). Der Quell-String befindet sich hierbei an der Adresse, die in DS:ESI (32-Bit-Umgebungen) bzw. DS:SI (16-Bit-Umgebungen) verzeichnet ist, die Adresse des Ziel-Strings befindet sich analog in ES:EDI bzw. ES:DI. Nach dem Befehl werden sowohl ESI/SI also auch EDI/DI um 1 (MOVSB), 2 (MOVSW) bzw. 4 (MOVSD) inkrementiert bzw. dekrementiert, sodass mit einem erneuten Aufruf des Befehls unmittelbar ein weiteres Datum kopiert werden kann.



Bitte beachten Sie, dass es einen »Namenskonflikt« gibt! MOVSD ist das Mnemonic für MOVS mit der Operandengröße DoubleWord, jedoch wird dieses Mnemonic seit der Einführung der SSE2-Befehle auch für den Transfer von skalaren DoubleWords in und aus einem XMM-Register verwendet (vgl. Seite 346). Ein echter Konflikt ist das jedoch nicht, da der Assembler anhand der übergebenen Operanden feststellen kann, ob nun die String- oder die XMM-Variante benutzt werden soll. Und auch für den Programmierer dürfte dies anhand des Programmkontextes ziemlich eindeutig sein.

CMPS
CMPSB
CMPSW
CMPSD

Diese Gruppe von Befehlen, *compare strings by byte/word/double word*, vergleicht ein Datum aus einem String mit einem in einem anderen. Je nach Datengröße erfolgt das Byte-weise (CMPSB), Word-weise (CMPSW) oder DoubleWord-weise (CMPSD). Der erste String befindet sich hierbei an der Adresse, die in DS:ESI (32-Bit-Umgebungen) bzw. DS:SI (16-Bit-Umgebungen) verzeichnet ist, der zweite an der in ES:EDI bzw. ES:DI spezifizierten Adresse. Nach dem Befehl werden sowohl ESI/SI also auch EDI/DI um 1 (MOVSB), 2 (MOVSW) bzw. 4 (MOVSD) inkrementiert bzw. dekrementiert, sodass mit einem erneuten Aufruf des Befehls unmittelbar zwei weitere Daten verglichen werden können.



Für den Vergleich werden die gleichen Mechanismen genutzt wie bei CMP. Das bedeutet, dass formal die Differenz aus dem Datum des ersten Strings (DS:EDI) und des zweiten Strings (ES:ESI) gebildet wird und anhand des anschließend verworfenen, temporären Ergebnisses die Statusflags gesetzt werden.



Bitte beachten Sie, dass es einen »Namenskonflikt« gibt! CMPSD ist das Mnemonic für CMPS mit der Operandengröße DoubleWord, jedoch wird dieses Mnemonic seit der Einführung der SSE2-Befehle auch für

den Vergleich zweier skalarer DoubleWords verwendet (vgl. Seite 346). Ein echter Konflikt ist das jedoch nicht, da der Assembler anhand der übergebenen Operanden feststellen kann, ob nun die String- oder die XMM-Variante benutzt werden soll. Und auch für den Programmierer dürfte dies anhand des Programmkontextes ziemlich eindeutig sein.

SCAS
SCASB
SCASW
SCASD

Diese Befehlsgruppe, *scan string by byte/word/double word*, vergleicht ein Datum aus dem Akkumulator mit einem aus einem String. Je nach Datengröße wird dabei das AL-Register (SCASB), das AX-Register (SCASW) oder das EAX-Register (SCASD) als Vorlage benutzt, mit dem das aus dem String stammende Datum verglichen wird. Dessen Adresse ist in ES: EDI (32-Bit-Umgebungen) bzw. ES:DI (16-Bit-Umgebungen) verzeichnet. Nach der Operation wird der Inhalt von EDI/DI um 1 (SCASB), 2 (SCASW) bzw. 4 (SCASD) inkrementiert bzw. dekrementiert, sodass ein erneuter Aufruf des Befehls sofort das nächste Datum prüfen kann.



Analog dem CMP-Befehl erfolgt die Prüfung, indem vom Muster im Akkumulator formal das Datum aus dem String abgezogen wird und anhand des anschließend verworfenen, temporären Ergebnisses die Statusflags gesetzt werden.

INS
INSB
INSW
INSD

Diese Befehlsgruppe, *Input from port to string by byte/word/double word*, liest ein Datum aus dem in DX spezifizierten Port und legt es in einem String ab. Je nach Datengröße wird der Port dabei Byte-weise (INSB), Word-weise (INSW) oder DoubleWord-weise (INSD) ausgelesen. Die Adresse des Ziel-Strings ist in ES: EDI (32-Bit-Umgebungen) bzw. ES:DI (16-Bit-Umgebungen) verzeichnet. Nach der Operation wird der Inhalt von EDI/DI um 1 (INSB), 2 (INSW) bzw. 4 (INS) inkrementiert bzw. dekrementiert, sodass ein erneuter Aufruf des Befehls sofort das nächste Datum auslesen kann.

OUTS
OUTSB
OUTSW
OUTSD

Auch der umgekehrte Weg ist möglich: Diese Befehlsgruppe, *Output from string to port by byte/word/double word*, liest ein Datum aus dem String und legt es im durch den Inhalt des DX-Registers spezifizierten Port ab. Je nach Datengröße wird der Port dabei Byte-weise (OUTSB), Word-weise (OUTSW) oder DoubleWord-weise (OUTSD) beschrieben. Die Adresse des Quell-Strings ist in DS: ESI (32-Bit-Umgebungen) bzw. DS:SI (16-Bit-Umgebungen) verzeichnet. Nach der Operation wird der Inhalt von ESI/SI um 1 (OUTSB), 2 (OUTSW) bzw. 4 (OUTSD) inkrementiert bzw. dekrementiert, sodass ein erneuter Aufruf des Befehls sofort das nächste Datum auf den Port legen kann.

1.1.12 Präfixe

Präfixe sind Code-Bytes, die Teil einer Befehlssequenz sind und keine eigenständige Funktion haben. Vielmehr erlauben sie nur in Verbindung mit dem Opcode eines Befehls, diesen zu modifizieren und bestimmte Standardbedingungen außer Kraft zu setzen (»*override*«). So können mit Hilfe von Präfixen Daten- oder Adressgrößen umdefiniert werden, Befehle repetiert oder Speicherzugriffe verboten werden.



Zur generellen Bedeutung und Benutzung von Präfixen als Teil von Befehlssequenzen vgl. das Kapitel »Mnemonics, Befehlssequenzen, Opcodes und Microcode« auf Seite 768. Hilfreich und für das Verständnis wichtig sind auch die Informationen im Kapitel »Adress- und Operandengrößen« auf Seite 765.



Präfixe sind, wie gesagt, keine eigenständigen Befehle. Es gibt für sie nicht in jedem Fall Mnemonics, da viele von ihnen vom Assembler/Compiler automatisch anhand der herrschenden Situation gesetzt werden. So streut der Assembler z.B. bei der Programmierung des Befehls `MOV AX, 01234h` automatisch ein *operand size override prefix* ein, wenn in und für 32-Bit-Umgebungen programmiert wird, da in diesen Umgebungen 32-Bit-Register Standard sind und somit EAX der »normale« Operand wäre.



Ein Präfix wirkt immer nur auf den nächsten Befehl! Aus diesem Grunde muss er unmittelbar vor dem Befehl oder als Teil der Operandenliste des Befehls angegeben werden, auf den er wirken soll. Soll z.B. ein Datum aus einem Segment geholt werden, das nicht über das Datensegment-Register DS adressiert wird, so hieße der Befehl z.B. `MOV EBX, ES:[Var32]`. Mit dieser Befehlskonstruktion wird als Operand eine logische Adresse übergeben, die von der standardmäßig übergebenen durch die Wahl eines anderen Segmentes als Bezugspunkt abweicht. Soll dagegen z.B. ein Byte in einem String gesucht werden, so hat das mit `REP SCASB` angegeben zu werden. Der Wiederholungspräfix `REP` wirkt hierbei nur auf den unmittelbar folgenden Befehl `SCAS`.



Beachten Sie, dass in der Regel Präfixe nicht bei allen Befehlen Sinn machen, angewendet werden (dürfen) oder wie erwartet reagieren! So machen die segment override prefixes nur in Verbindung mit Speicherzugriffen Sinn, nicht aber bei bedingten Sprüngen oder Unterprogrammaufrufen. Wiederholungspräfixe sind sinnlos, wenn ein

Speicherzugriff erfolgen soll – oder warum sollte man 4711-mal hintereinander den Befehl MOV EAX, EBX ausführen?

Daher bewirken manche Präfixe in Kombination mit der einen Befehlsklasse das eine, in Verbindung mit einer anderen das andere. Beispiel hierfür sind die Präfixe mit den Codes $2E und $3E: In Verbindung mit einer Adresse dienen sie als segment override prefix (CS: bzw. DS:), in Verbindung mit einer Programmverzweigung als branch hint (branch taken bzw. branch not taken).

Eine weitere Verwendung von Präfix-Codes ist die »Erweiterung« der Anzahl von Opcode-codierenden Bytes von zwei auf drei! Üblicherweise werden Opcodes mit einem, maximal zwei Bytes codiert. In einigen Ausnahmefällen reichen diese beiden Opcode-Bytes jedoch nicht aus. Wenn dann Teil der Befehlssequenz ein ModR/M-Byte ist, kann dieses teilweise dazu benutzt werden, den Opcode zu erweitern. Doch es gibt auch eine andere Möglichkeit, die von einigen Befehlen der SIMD-Erweiterungen genutzt wird: Verwendung von Präfixen, die ansonsten für diese Befehle keine Bedeutung hätten. So verwenden einige SSE-Befehle den Präfix $F3, der in Verbindung mit String-Befehlen als REPE interpretiert wird. SSE2-Befehle verwenden darüber hinaus auch die Präfixe $F2 (bei String-Befehlen ist dies der REPNE-Präfix) und $66 (bei Befehlen mit Speicheroperanden ist dies der operand size override prefix).

segment override

Mit den segment override prefixes kann bei Befehlen, die auf Daten zugreifen, ein anderes als das standardmäßig vorgesehene Segmentregister DS als Datensegment-Register gewählt werden. Es gibt für jedes der sechs verfügbaren Segmentregister ein Mnemonic: CS:, DS:, ES:, FS:, GS: und SS: (bitte beachten Sie den obligatorischen »:« als Teil jedes Mnemonics).



Für ein besseres Verständnis der segment override prefixes empfiehlt es sich, das Thema Adressierung genauer zu verstehen. Falls Sie sich auf diesem Gebiet noch nicht so gut auskennen, sollten Sie das Kapitel »Zugriffe auf den Speicher: Von Adressen und Adressräumen« ab Seite 434 konsultieren.



Segment override prefixes beziehen sich immer auf die logische Adresse und sind somit immer Teil einer Adressangabe (weshalb auch der Doppelpunkt Pflicht ist, siehe »Beziehungskisten: Von der effektiven zur logischen Adresse« auf Seite 435). Auch wenn die Angabe des Be-

zugssegments durch viele Befehle in Form des Einbindens des DS-Registers impliziert und lediglich eine effektive Adresse verwendet wird, ist der standardmäßige Zugriff auf das Datensegment immer codiert durch `Segment:Effektive Adresse`. Daher stehen diese Präfixe (auf der Assemblerebene) nicht wie andere vor dem eigentlichen Befehl, sondern sind Teil der zum Befehl gehörenden Operandenliste, wie im Beispiel `MOV ES:[Var32], EAX`. ACHTUNG: Auf der Maschinenebene werden die Präfixe dagegen als das dargestellt, was sie sind: Präfixe *vor* Opcodes! Wundern Sie sich daher nicht, wenn ein Debugger beispielsweise den eben genannten Befehl so darstellt: `ES: MOV [Var32], EAX`. Assembler arbeiten anwenderfreundlich, Debugger hardwarenah!



Es gibt zwei Möglichkeiten, das anzusprechende Datensegment zu ändern:

- Änderung des Inhaltes des Segmentregisters DS
- Angabe eines Segmentregisters mit einem segment override prefix

Methode 1 macht nur Sinn, wenn »längerfristig« die Änderung des Standard-Datensegments Sinn macht (was auch immer längerfristig in diesem Zusammenhang heißt!). Durch den Aufwand, der aufgrund der Schutzkonzepte im protected mode getrieben werden muss, wenn ein Segmentregister beschrieben wird, wirkt es sich negativ auf die Performance aus, wenn häufig der Inhalt der Segmentregister verändert wird, vor allem, wenn es ein »Hin- und Herschalten« ist. In diesem Falle ist es sinnvoller, ein freies Segmentregister mit dem zweiten Datensegment zu belegen und über Methode 2 den jeweiligen Zugriff zu realisieren.



Beachten Sie, dass das Stacksegment vom Prozessor grundsätzlich über das SS-Register angesprochen wird. Ein segment override für das Stacksegment ist somit nicht möglich, falls unterschiedliche Stacksegmente benutzt werden sollen (müssen: Stichwort Wechsel in eine andere Privilegstufe!), muss der Inhalt des SS-Registers verändert werden.

Ebenso ist es nicht möglich, das Codesegment durch ein segment override prefix zu ändern! Ein anderes Codesegment als das aktuelle kann nur über den Umweg CALL oder JUMP mit einer qualifizierten (also vollständigen, aus Segment und Offset bestehenden) Adresse oder über INT erfolgen!

operand size override

Der operand size override prefix ist ein automatisch vom Assembler/Compiler eingestreuter Präfix, der immer dann Verwendung findet, wenn mit Operandengrößen gearbeitet werden muss, die vom aktuellen Standard abweichen. Details hierzu finden Sie im Kapitel »Adress- und Operandengrößen« auf Seite 765.

address size override

Auch der address size override prefix ist ein automatisch vom Assembler/Compiler eingestreuter Präfix, der immer dann Verwendung findet, wenn mit Adressen gearbeitet werden muss, die vom aktuellen Standard abweichen. Details hierzu finden Sie ebenfalls im Kapitel »Adress- und Operandengrößen« auf Seite 765.

REP
REPcc

Es gibt zwei Präfixe, mit denen der folgende Befehl wiederholt werden kann, ohne dass eine Schleife programmiert werden müsste: *repeat* (REP) und *repeat while condition cc* (REPcc).



Und hier folgt schon die erste, erhebliche Einschränkung! Nicht jeder Befehl kann mit dem Präfix REP bzw. REPcc erweitert werden! Es handelt sich ausschließlich um die String-Befehle, die im Abschnitt »Operationen mit »Strings«« in diesem Kapitel besprochen wurden. Was passiert, wenn Sie REP oder REPcc in Verbindung mit anderen Befehlen benutzen, ist hardwareabhängig und unvorhersehbar! Es können zweitens auch keine Schleifen programmiert werden! Wenn die Stringbefehle im Rahmen einer Schleife eingesetzt werden und nicht isoliert repetiert werden können, muss mit LOOP gearbeitet werden.



Eine dritte Einschränkung: Nicht alle Stringbefehle können mit den beiden Präfixen verwendet werden. So kann REP nur in Verbindung mit MOVS, STOS, LODS, INS und OUTS genutzt werden, während REPcc nur in Verbindung mit SCAS und CMPS funktioniert. (Bitte beachten Sie, dass hier jeweils die Oberbegriffe benutzt werden: MOVS steht für MOVS, MOVSB, MOVSW und MOVSD!)

REP ist ein einfacher Repetierbefehl. Er wiederholt den folgenden Stringbefehl solange, bis das Abbruchkriterium erfüllt ist. Und dieses Abbruchkriterium ist, dass ein in ECX (32-Bit-Umgebungen) bzw. CX (16-Bit-Umgebungen) stehender Zähler auf Null heruntergezählt wurde. Das bedeutet, dass REP immer dann zum Einsatz kommt, wenn eine vorher bestimmbare Anzahl von Wiederholungen erfolgen soll.

Soll dagegen auch im Rahmen der Abarbeitung der Stringbefehle ein vorzeitiger »Ausstieg« möglich sein, weil z.B. das Byte gefunden wurde, was mit SCASB gesucht wird, so hat ein zusätzliches Abbruchkriterium geprüft zu werden. Bei SCAS und CMPS kommt es darauf an, zwei Daten in zwei Strings (CMPS) bzw. ein Datum aus einem String mit einem Testdatum (SCAS) zu vergleichen. Abbruchkriterium bei solchen Vergleichen ist z.B. die Gleichheit. REPcc prüft daher zusätzlich zum Zählerstand in ECX/CX auch, ob das zero flag gesetzt ist. Und gesetzt wird das zero flag dann, wenn CMPS oder SCAS die Identität der Werte festgestellt haben.

REPcc gibt es somit in zweimal zwei Versionen: REPE/REPZ und RENE/REPNZ. REPE und REPZ sind Synonyme, ebenso REPNE und REPNZ. REPE und REPNE sind die logischen Negationen von einander, ebenso REPZ und REPNZ. Es wundert daher nicht, dass es für REPcc nur zwei Codes gibt: $F3 (REPE/REPZ) und $F2 (REPNE/REPNZ). Vgl. hierzu auch Seite 43. Der Code für REP ist $F3.



Da der Code für REP und REPE/REPZ gleich ist ($F3), bewirkt er als Präfix Unterschiedliches, je nachdem, mit welchem Stringbefehl er benutzt wird. In Verbindung mit den »einfachen« Stringbefehlen wie MOVS oder INS wird lediglich der Zählerstand als Abbruchkriterium verwendet, in Verbindung mit SCAS und CMPS neben dem Zählerstand auch das zero flag. Bitte behalten Sie dies immer im Hinterkopf!



Ob bei der Verwendung von REPcc die Wiederholung aufgrund eines abgelaufenen Zählers (ECX = 0) oder aufgrund der anderen Abbruchbedingung (ZF = 0 bzw. ZF = 1) erfolgte, lässt sich mit Hilfe bedingter Verzweigungen recht elegant feststellen. Das folgende Codefragment prüft hierzu den Zählerstand mittels JCXZ:

```
        :
        :
        REPNE SCASD ; scanne string, bis Datum gefunden
        JCXZ  Down  ; Sprung, wenn Zähler abgelaufen
        :           ; Zähler nicht abgelaufen (ECX > 0)
        :           ; dann wurde Datum gefunden
        JMP   Ende
Down:   :           ; Zähler abgelaufen (ECX = 0)
        :           ; dann wurde Datum nicht gefunden
Ende:   :
```

Im folgenden Codefragment wird die Prüfung des zero flag verwendet:

```
        :
        :
        REPNE SCASD ; scanne string, bis Datum gefunden
        JE    Found ; Datum gefunden (ZF = 1)
        :           ; Datum nicht gefunden (ZF = 0)
        :           ; dann ist auch ECX = 0!
        JMP   Ende
Found:  :           ; Datum gefunden, ECX > 0!
        :           ;
Ende:   :
```

Sie sehen, beide Versionen führen zum gleichen Ziel!



Die schnellste und effizienteste Möglichkeit, einen großen Speicherbereich zu initialisieren, ist die Verwendung von `REP STOS`! Sie macht jedoch wirklich nur dann Sinn, wenn der Speicherblock so groß ist, dass der Overhead bei der Verwendung der Stringbefehle (Laden des Segmentregisters ES samt Prüfung im Rahmen der Schutzkonzepte, Initialisieren des DI-Registers mit der Startadresse des Strings, Initialisieren des Counters in ECX) nicht ins Gewicht fällt.



Falls Sie REP in Verbindung mit INS oder OUTS verwenden wollen, bedenken Sie bitte, dass nicht jeder I/O-Port in der Lage ist, mit den Transfergeschwindigkeiten des Prozessors im Rahmen von REP mitzuhalten! Dies kann eventuell zu erheblichen Problemen führen, die nicht so leicht aufzufinden sind!

LOCK

In Multi-Prozessor-Umgebungen kann es vorkommen, dass ein Prozessor auf eine Speicherstelle zugreifen will, die ein anderer gerade verändert. Um dies zu verhindern, muss daher der Datenbus kurzfristig für Zugriffe »von außen« gesperrt werden, solange ein Prozessor ihn benötigt. Genau dieses Sperren ermöglicht der Präfix LOCK. Er garantiert, dass für den folgenden Befehl der Prozessor den alleinigen Zugriff auf den Datenbus hat. Nach dem »verschlossenen« Befehl ist der Datenbus wieder frei.

Das bedeutet, dass LOCK nur dann Sinn macht, wenn auf den Speicher zugegriffen wird. Daher kommen als Befehle, vor denen LOCK stehen kann, nur folgende in Frage: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD und XCHG.



Die genannten Befehle akzeptieren als Operanden auch Nicht-Speicherstellen. Wird der LOCK-Präfix mit einem der genannten Befehle verwendet und keiner der Operanden ist eine Speicherstelle oder wird er in Verbindung mit einem oben nicht aufgeführten Befehl eingesetzt, wird eine invalid opcode exception #UD generiert!

branch hints

Teil der SSE2-Erweiterungen der Intel-Prozessoren ist die Unterstützung der branch prediction unit des Prozessors. Diese Einheit versucht, möglichst gute Vorhersagen zu machen, wohin bei Programmverzweigungen im konkreten Fall verzweigt wird. Dazu gibt es recht klug ausgedachte und raffinierte Methoden, auf die ich hier nicht weiter eingehen kann.

Manchmal ist es jedoch sinnvoll, der prediction unit ein wenig unter die Arme zu greifen. Schließlich weiß niemand besser als der Programmierer, mit welchen Daten es ein Programm an welcher Stelle zu tun bekommt. Daher wurden mit den branch hints Möglichkeiten geschaffen, der Unit zu signalisieren, was wohl als Nächstes am wahrscheinlichsten passieren wird. Hierzu werden zwei Codes benutzt, die bislang nur mit speicherzugreifenden Befehlen erlaubt waren: $2E und $3E, die weiter oben als segment override prefixes CS: und DS: beschrieben worden sind.

In Verbindung mit einem bedingten Sprungbefehl (Jcc) erlauben es diese Präfixe nun, der prediction logic einen Hinweis zu geben, was der Programmierer als wahrscheinliches Ergebnis der Programmverzweigung hält: branch taken ($2E), also eine zu erwartende Programmverzweigung an die Adresse des Sprungbefehls, oder branch not taken ($3E), also ein ungehinderter Fluss des Programmablaufs mit der dem Jcc-Befehle folgenden Instruktion.



Bitte beachten Sie, dass es kein Mnemonic für diese Präfixe gibt, obschon sie vom Programmierer verwendet werden müssen, da kein Assembler/Compiler der Welt diesen hint abgeben kann. Daher ist im Falle der Nutzung dieser Präfixe auf die »Assemblierung von Hand« mittels DB-Anweisungen zurückzugreifen.

1.1.13 Adressierungs-Befehle



Zum Verständnis des Inhaltes dieses Abschnitts empfiehlt es sich, Details zu logischen und effektiven Adressen und zur direkten und indirekten Adressierung zu kennen. Falls hierzu Bedarf besteht, können Sie die erforderlichen Informationen in den Abschnitten »Beziehungskisten: Von der effektiven zur logischen Adresse« auf Seite 435 bzw. »Speicheradressierung« auf Seite 816 erhalten.

Bislang wurden Adressen von Speicherstellen nur in Form der Angabe als Operanden für Befehle besprochen. So kann man einem MOV-Befehl die Adresse einer Speicherstelle angeben, in die oder aus der ein Datum zu lesen ist. Die Angabe erfolgt dabei sowohl bei Assemblern als auch bei Compilern über die Verwendung von Konstanten- und/oder Variablennamen, bei Sprungbefehlen über Labels.

Diese Art der Adressierung nennt man *direkte* Adressierung, da die Adresse direkt (wenn auch durch einen Namen »verschlüsselt«) dem Befehl übergeben wird. Häufig aber möchte oder muss man auch den *indirekten* Weg gehen. Dann befindet sich die Adresse in einem Register (Registerkombination), das dem Befehl als Operand übergeben wird und aus dem er sich die Adresse vor dem Speicherzugriff holt. In diesem Register kann sie nach Bedarf manipuliert werden (z.B. Addition eines Offsets zur einer Basisadresse bei Feldern oder anderen Datenstrukturen).

Doch wie bekommt man eine Adresse in ein Register? Im Quelltext wurden ja lediglich Variable, Konstanten und Labels deklariert, die ein Alias für die zu verwendenden Adressen sind. Die Adressen selbst berechnet der Assembler/Compiler anhand der Deklarationen – der Programmierer bekommt sie in der Regel niemals zu Gesicht (was ja das Anwenderfreundliche an den Assemblern/Compilern ist!). Mehr noch: Zur Zeit der Erstellung des Quelltextes existieren diese Adressen noch gar nicht, sie werden erst während der Assemblierung/Compilierung erzeugt. Und dennoch muss und kann über die Alias mit ihnen gearbeitet werden.

Um also Adressen, die es noch gar nicht gibt, in Register zu laden, gibt es zwei Typen von Befehlen: diejenigen, die die effektive Adresse (also den Offset einer logischen Adresse) in ein Register schreiben und diejenigen, die eine vollständige logische Adresse in eine Registerkombination schreiben.

LEA Mit *load effektive address*, LEA, wird die effektive Adresse, also der Offset-Anteil einer logischen Adresse, berechnet und in ein Register geladen.

Operanden LEA erwartet zwei Operanden: als Zieloperanden ein Register, als Quelloperanden den Namen einer Speicherstelle:

```
LEA Reg16, Mem; LEA Reg32, Mem
```

Da hier nur die Adressen eines Datums eine Rolle spielen, ist es unerheblich, welches Datum *Mem* repräsentiert.



Da der Zieloperand 16 oder 32 Bit breit sein kann, auf der anderen Seite jedoch `Mem` je nach Umgebung ebenfalls 16 oder 32 Bit breit sein kann, gibt es vier Fälle zu unterscheiden:

- Zielregister 16 Bit, 16-Bit-Umgebung (und damit 16-Bit-Offsets): Die effektive 16-Bit-Adresse wird im Register abgelegt.
- Zielregister 16 Bit, 32-Bit-Umgebung (und damit 32-Bit-Offset): Von der effektiven 32-Bit-Adresse wird das niedrigerwertige Word in das Register abgelegt, das höherwertige Word wird verworfen
- Zielregister 32 Bit, 16-Bit-Umgebung: Die effektive 16-Bit-Adresse wird in das niedrigerwertige Word des Registers geladen, das höherwertige wird mit Nullen aufgefüllt.
- Zielregister 32 Bit, 32-Bit-Umgebung: Die effektive 32-Bit-Adresse wird im Register abgelegt.



Assembler »übersetzen« in der Regel den Befehl `LEA Reg, Mem` in den effektiveren Befehl `MOV Reg, OFFSET Mem`, wenn *Mem* eine direkte Adresse repräsentiert. Im Falle indirekter Adressierung dagegen muss LEA verwendet werden.

Statusflags Die Statusflags werden durch LEA nicht verändert.

LDS
LES
LFS
LGS
LSS

Will man dagegen eine vollständige logische Adresse bestehend aus Segment-Selektor und effektiver Adresse eruieren, so kommt der Befehl *load far pointer* ins Spiel. Ihn gibt es in fünf Versionen, je nachdem, welches Segmentregister involviert werden und den Selektor des Segments aufnehmen soll: LDS (DS-Register), LES (ES-Register), LFS (FS-Register), LGS (GS-Register) und LSS (SS-Register). Das Codesegment-Register CS kann nicht benutzt werden!

Operanden

Alle Befehle erwarten als Zieloperanden ein Register, das zusammen mit dem im Opcode codierten Segmentregister den 48-Bit-Pointer einer 32-Bit-Umgebung oder den 32-Bit-Pointer einer 16-Bit-Umgebung aufnimmt. Quelloperand (zweiter Operand) ist der Name der gewünschten Speicherstelle (XXX steht für LDS, LES, LFS, LGS oder LSS):

```
XXX Reg16, Mem; XXX Reg32, Mem
```



Bitte beachten Sie, dass alle fünf Befehle in ein Segmentregister schreiben. Dies bedeutet im protected mode, dass eine Überprüfung der Privilegien im Rahmen der Schutzkonzepte erfolgt. Ferner werden die nicht sichtbaren Teile der Segmentregister mit den Daten aus den Deskriptoren geladen (vgl. »Hardwareunterstützung für Deskriptoren und Deskriptortabellen« auf Seite 431.

Das bedeutet auch, dass »Null-Selektoren« ohne Auslösung einer exception geladen werden können; allerdings führt dann jeder nachfolgende Zugriff auf das Segment zu einer general protection exception #GP.

Statusflags

Die Statusflags werden durch LDS, LES, LFS, LGS und LSS nicht verändert.

1.1.14 Spezielle Befehle

CPUID

Viele Fähigkeiten des Prozessors sind prozessorspezifisch. Aufgrund der Evolution der Prozessoren verfügt jeweils die neueste Generation über Befehle, die der Vorgänger noch nicht hatte. Daher ist es wichtig, vor der Benutzung von verschiedenen Befehlen prüfen zu können, ob der Prozessor dies oder jenes kann oder nicht.

Bis zum Pentium war hierzu nötig, festzustellen, welcher Prozessor vorliegt. Dies erfolgte über mehr oder weniger »gute«, teilweise »trickreiche«, manchmal »unsaubere« (self-modifying code) Methoden und mehr schlecht als recht. Seit dem Pentium dagegen gibt es hierfür einen Befehl, der die CPU-Identität preisgibt: CPUID, *CPU identification*.



Ob der aktuelle Prozessor über den Befehl CPUID verfügt, können Sie dem EFlags-Register entnehmen. Lässt sich das ID-Flag (Bit 21) gezielt umschalten, so ist CPUID implementiert. Das sollte bei allen Prozessoren ab dem Pentium und seinen Klonen und selbst für spätere Versionen des 80486 der Fall sein!

Dieser Befehl gibt eine Masse an Informationen zurück, die fast alles über die Fähigkeiten des Prozessors aussagen. Die Informationsfülle ist so groß, dass man dem Befehl einen Parameter mitgeben muss, der ihm signalisiert, zu welchem »Thema« man Informationen wünscht: zum Prozessortyp, zu seinen Features, zu caches und TLBs etc. Man teilt sie in »Funktionen« des Befehls CPUID auf.

Es gibt zwei grundsätzliche Typen von Informationen: die »Basisinformationen« mit Funktionsnummern, bei denen Bit 31 gelöscht ist, und »erweiterte« Informationen, bei denen Bit 31 der Funktionsnummer gesetzt ist.



Bitte beachten Sie, dass Intel und AMD hier zwar kompatible Informationen zur Verfügung stellen, dazu aber nicht immer die analogen Funktionen benutzen. Bis zum heutigen Tage verwenden z.B. aktuelle AMD-Prozessoren wie Athlon MP Model 6 und Duron Model 3 nur die Basisfunktionen 0 und 1 des CPUID-Befehls, um Kompatibilität zu Intel-Prozessoren zu gewährleisten (vendor ID string, processor signature und feature flags). Alle darüber hinaus gehenden, teilweise AMD-spezifischen Informationen (3DNow!-Verfügbarkeit, Informationen zu Cache-Größe etc.) werden in »erweiterten« Funktionen verwaltet. Umgekehrt »entdeckt« Intel erst jetzt die erweiterten Funktionen, indem es erstmalig mit dem Pentium 4 deren Existenz dokumentiert, wenn auch zurzeit noch keine Informationen damit zu erhalten sind.

Funktion 0 (Basisfunktion)

Funktion 0 gibt in EAX die höchste Funktionsnummer (Basisfunktionen!), die der Prozessor kennt. So liefert CPUID über diese Funktion bei einem Pentium 4 den Wert »2« zurück, was besagt, dass die Funktionen 0 bis 2 verfügbar sind. Der P III gibt den Wert 3 zurück, Pentium Pro und Pentium II den Wert 2. Der Pentium und späte Versionen des 80486 geben »1« zurück. In den Registern EBX, ECX und EDX wird üblicherweise ein Identifikationsstring (*vendor identification string*) zurückgegeben. Die Register sollten in der Reihenfolge ECX:EDX:EBX ausgelesen werden, jedoch in Intel-Schreibweise. Bei einem originalen Intel-Prozessor wird dann $6C65746E : $49656E69 : $756E6547 übergeben, was den ASCII-Zeichen "letn", "Ieni" und "uneG" entspricht. Nach Intel-Art von hinten nach vorne gelesen steht dann da: »GenuineIntel«. AMD verwöhnt uns hier mit einem »AuthenticAMD«.



Da jeder Prozessor über eine unterschiedliche Anzahl an Funktionen verfügt, sollte auf jeden Fall Funktion 0 aufgerufen und der Inhalt von EAX ausgewertet werden, bevor man eine höhere Funktion aufruft. Andernfalls riskiert man, vermeintliche Informationen zu erhalten, die jedoch nicht richtig sind. Denn jeder Funktionswert oberhalb des höchsten gültigen Funktionswertes gibt die Information des höchsten gültigen Wertes zurück, es sei denn, es handelt sich um einen gültigen Wert einer erweiterten Funktion. Beispiel: Die Übergabe von $0005 oder $8005 bei einem Pentium 4 gibt die Information der Funktion $0002 zurück, da der höchste gültige Funktionswert abgesehen von den erweiterten Werten $8000 bis $8004 der Wert »2« ist.

Funktion 1 (Basisfunktion)

Funktion 1 gibt Informationen zur Prozessor-Version zurück, zu seinen Features und einigen Besonderheiten:

EAX enthält bei Intel nach Rückkehr die *processor signature*, also Informationen über die Prozessor-Version gemäß Abbildung 1.14.

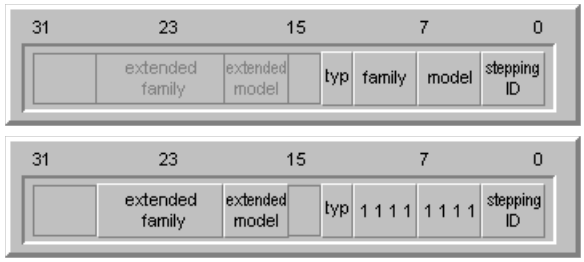


Abbildung 1.14: Speicherabbild des EAX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls bei Intel-Prozessoren

Die in den Feldern »type«, »family« bzw. »instruction family«, »model« und »stepping ID« eingetragenen Bits sind als binär codierte Zahlen zu interpretieren. Das bedeutet: type hat einen Wertebereich von 0 bis 3 (11_b), die anderen Felder von 0 bis 15 (1111_b). Liegt der Wertebereich der Felder »family« und »model« zwischen 0 und 14 (1110_b), so sind die Bits 16 bis 31 des EAX-Registers nicht definiert (Abbildung 1.14, oben). Andernfalls existieren entsprechende »Extended«-Felder: family wird in »extended family« fortgeschrieben, model in »extended model« (Abbildung 1.14, unten).

Das Feld »type« enthält den Typ des Prozessors: *original OEM* (00_b), *Intel overdrive* (01_b), *dual* (10_b) und *reserved* (11_b). Prozessoren vom Typ dual sind in der Lage, in Zwei-Prozessor-Systemen eingesetzt zu werden.



AMD gibt aus Kompatibilitätsgründen vergleichbare Informationen zur processor signature in EAX zurück, wie Abbildung 1.15 zeigt. Allerdings kennt AMD das Feld *type* nicht und benutzt bis heute auch nicht die Felder *extended family* und *extended model*. Stattdessen spendiert AMD seinen neueren Prozessoren eine »AMD processor signature« in Funktion 8001.

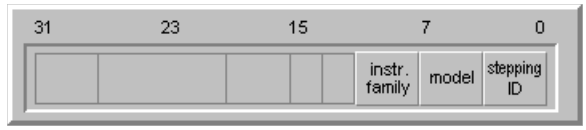


Abbildung 1.15: Speicherabbild des EAX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls bei AMD-Prozessoren

Das EBX-Register enthält bei Intel zurzeit drei Felder, die in Abbildung 1.16 dargestellt sind. AMD bezeichnet den Inhalt des EBX-Registers als reserviert.



Abbildung 1.16: Speicherabbild des EBX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls bei Intel-Prozessoren

- brand index; dieses Feld kann dazu benutzt werden, den Prozessor zu identifizieren. Es besteht seit dem Pentium III Xeon.
- CLFLUSH; dieses Feld gibt die Größe der cache line in 8-Byte-Blöcken an, die mit dem Befehl CLFLUSH geleert wird. Das Feld existiert seit dem P 4.
- APIC ID; dieses Feld gibt die physische 8-Bit-Nummer an, die dem lokalen APIC (*advanced programmable interrupt controller*) während des Einschaltens des Prozessors zugeordnet wird. Das Feld existiert seit dem Pentium 4.

Der brand index ist ein Zeiger in eine Tabelle, die zurzeit den in Tabelle 1.9 dargestellten Inhalt hat.

Index	Brand String
0	This processor does not support the brand identification feature
1	Celeron processor
2	Pentium III processor
3	Intel Pentium III Xeon processor
4 – 7	reserved for future processor
8	Intel Pentium 4 processor
9 – 255	reserved for future processor

Tabelle 1.9: Dem brand index aus Funktion 1 des CPUID-Befehls zugeordnete brand strings

ECX enthält Daten, die laut Intel und AMD reserviert sind.

In EDX wird ein Bit-Feld zurückgegeben, das üblicherweise als *feature flags* bezeichnet wird. Die hier definierten Flags signalisieren, ob der Prozessor über die entsprechende Fähigkeit verfügt. Die feature flags sind in Abbildung 1.17 dargestellt, im oberen Teil der Abbildung für Intel-Prozessoren und im unteren Teil für AMD-Prozessoren. Man erkennt, dass AMD einige der von Intel implementierten Funktionen nicht realisiert (processor serial number, PSN; CLFLUSH; debug store, DS; thermal monitor, ACPI und TM; und self snoop, SS).

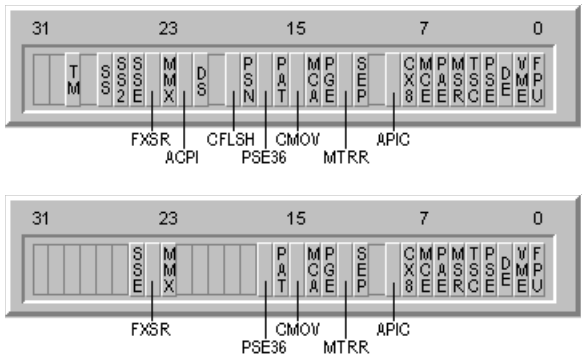


Abbildung 1.17: Speicherabbild des EDX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls

Falls das entsprechende Flag gesetzt ist, bedeuten:

FPU *floating point unit* on chip; auf dem CPU-Chip ist eine Fließkommaeinheit vorhanden.

VME *virtual 8086 mode enhancements* verfügbar; das bedeutet, dass in control register #4 das Flag VME existiert, mit dem man die

enhancements nutzen kann, ebenso das Flag PVI für die Verwaltung von virtuellen Interrupts, dass eine Interrupt-Umleitung per Software möglich ist und das TSS hierfür um eine Umleitungsliste erweitert wurde sowie dass im EFlags-Register die Flags VIF und VIP verfügbar sind.

DE *debugging extensions* verfügbar; dieses Flag signalisiert, dass I/O-Breakpoints unterstützt werden und hierfür das Flag DE in control register #4 existiert.

PSE *page size extensions* verfügbar; hierdurch werden 4-MByte-Pages ermöglicht sowie ihre Kontrolle über das Flag PSE in control register #4. Ferner ist in den PDE (page directory entries) das Flag *dirty* definiert. Einzelheiten siehe Kapitel »PSE-Modus« auf Seite 447.

TSC *time stamp counter* verfügbar; der Prozessor unterstützt den Befehl RDTSC inklusive des Flags TSD in control register #4.

MSR *machine specific registers* instructions vorhanden; der Prozessor unterstützt die Befehle RDMSR und WRMSR.

PAE *physical address extension* möglich; der Prozessor unterstützt physikalische Adressen jenseits von 4 GByte (32-Bit-Adressen). Hierzu werden erweiterte Page-Table-Formate unterstützt und eine zusätzliche Ebene in der Adressberechnung eingeführt. Wie groß der adressierbare Bereich jenseits der 4-GByte-Marke ist, wird nicht definiert und ist implementationsabhängig. Siehe auch »PAE-Modus« auf Seite 449.

MCE *machine check exceptions verfügbar*; die Exception #18 (machine check exception #MC) ist definiert. Damit enthält control register #4 auch das Flag MCE, das das Verhalten bei einer #MC steuert.

CX8 *compare and exchange 8* bytes implementiert; der Prozessor unterstützt den Befehl CMPXCHG8B.

APIC *APIC* on chip; auf dem Prozessorchip ist ein *advanced programmable interrupt controller* (APIC) vorhanden.

SEP SYSENTER und SYSEXIT implementiert; der Prozessor unterstützt das Befehlspaar SYSENTER und SYSEXIT, was bedeutet, dass auch die hierzu notwendigen modellspezifischen Register (MSRs) vorhanden sind und angesprochen werden können.

MTRR *memory type range registers* vorhanden; der Prozessor unterstützt MTRRs als spezielle MSRs. Das MSR MTRRCAP (Adresse 254) enthält feature flags, die genauere Informationen zu den Memory-Typen beherbergen, wie viele MTRRs es gibt und ob statische MTRRs unterstützt werden.

PGE *page global entry* bit verfügbar; das control register #4 enthält das Flag PGE, das in Verbindung mit dem global bit in page directory entries (PDEs) und page table entries (PTEs) steuert, ob die betreffende page als »global verfügbar« markiert werden kann und somit vom »Flushen« ausgeschlossen wird.

MCA *machine check architecture* wird unterstützt; der Prozessor unterstützt die machine check architecture, die einen Mechanismus für Fehlersuche und -bericht darstellt. Das bedeutet auch, dass es das MSR MCG_CAP (Adresse 377) gibt, in dem feature flags zur machine architecture verzeichnet sind.

CMOV CMOVcc implementiert; der Prozessor unterstützt den Befehl CMOVcc und, sofern eine floating point unit verfügbar ist (Flag FPU!), auch die Befehle FCOMI und FCMOV.

PAT *page attribute tables* werden unterstützt. Vgl. »Page Attribute Table« auf Seite 457.

PSE36 *36-Bit page size extension* verfügbar; der Prozessor unterstützt eine weitere Möglichkeit, Adressen jenseits der 4-GByte-Grenze anzusprechen. Einzelheiten siehe Kapitel »PSE-36-Modus« auf Seite 449.

PSN *processor serial number* verfügbar; der Prozessor besitzt eine Seriennummer, die mit Hilfe des CPUID-Befehls festgestellt werden kann.

CFLSH CLFLUSH implementiert; der Befehl CLFLUSH wird unterstützt.

DS *debug store* möglich; der Prozessor unterstützt die Möglichkeit, Debug-Informationen in einen speicherresidenten Puffer zu schreiben.

ACPI Thermal Monitor and Software Controlled Clock Facilities; der Prozessor besitzt spezielle MSRs, mit denen er seine Temperatur verfolgen und in Abhängigkeit davon, softwaregesteuert, die Prozessor-Performance variieren kann.

MMX MMX-Erweiterungen werden unterstützt; der Prozessor unterstützt die MMX-Befehle und hat die dazu erforderlichen FPU-Register.

FXSR FXSAVE und FXRSTOR; der Prozessor implementiert die Befehle FXSAVE und FXRSTOR und stellt dem Betriebssystem in control register #4 das Flag OSFXSR zur Verfügung, mit dem dieses Anwendungsprogrammen signalisieren kann, ob es die Befehle FXSAVE und FXRSTOR unterstützt.

SSE SSE-Erweiterungen werden unterstützt; der Prozessor implementiert die XMM-Register und unterstützt die SSE-Befehle.

SSE2 SSE2-Erweiterungen werden unterstützt; der Prozessor unterstützt die SSE2-Erweiterungen.

SS *self snoop*; der Prozessor unterstützt die Verwaltung von kollidierenden Speichertypen, indem es seinen eigenen cache überprüft.

TM *thermal monitor*; der Prozessor unterstützt die TCC (thermal control circuitry), die die Temperatur des Prozessors überwacht und regelt.

Funktion 2 (Basisfunktion)

Funktion 2 des CPUID-Befehls, die bislang nur in Intel-Prozessoren eine Rolle spielt, gibt Informationen zu Cache und TLB des Prozessors zurück. (AMD verwendet für vergleichbare Informationen die erweiterten Funktionen $8005 und $8006.) Hierbei ist der Aufbau der Registerinhalte EAX, EBX, ECX und EDX mit einer Ausnahme identisch und in Abbildung 1.18 dargestellt.

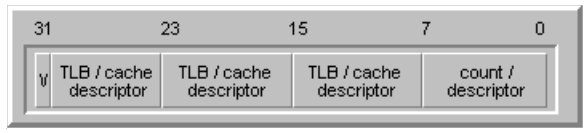


Abbildung 1.18: Speicherabbilder der Inhalte der Register EAX, EBX, ECX und EDX nach Aufruf der Funktion 2 des CPUID-Befehls bei Intel-Prozessoren

Das niedrigstwertige Byte (Bits 0 bis 7) des EAX-Registers enthält einen Wert, der angibt, wie oft die Funktion 2 aufgerufen werden muss, um alle Informationen zu erhalten. In den Registern EBX, ECX und EDX steht an dieser Stelle, wie an den anderen Positionen auch, ein Code für die Eigenschaften des cache und/oder TLB, der anhand Tabelle 1.10 de-

kodiert werden kann. Das Flag V zeigt an, ob der Registerinhalt valide (V = 0) oder als reserviert aufzufassen ist (V = 1).

Code	Beschreibung
00h	Nulldeskriptor
01h	Instruction TLB: 4 kB Pages, 4-way set associative, 32 entries
02h	Instruction TLB: 4 MB Pages, 4-way set associative, 2 entries
03h	Data TLB: 4 kB Pages, 4-way set associative, 64 entries
04h	Data TLB: 4 MB Pages, 4-way set associative, 8 entries
06h	1st-level instruction cache: 8 kB, 4-way set associative, 32 byte line size
08h	1st-level instruction cache: 16 kB, 4-way set associative, 32 byte line size
0Ah	1st-level data cache: 8 kB, 2-way set associative, 32 byte line size
0Ch	1st-level data cache: 16 kB, 4-way set associative, 32 byte line size
40h	No 2nd level cache (P6 family) or no 3rd-level cache (Pentium 4)
41h	2nd-level cache: 128 kB, 4-way set associative, 32 byte line size
42h	2nd-level cache: 256 kB, 4-way set associative, 32 byte line size
43h	2nd-level cache: 512 kB, 4-way set associative, 32 byte line size
44h	2nd-level cache: 1 MB, 4-way set associative, 32 byte line size
45h	2nd-level cache: 2 MB, 4-way set associative, 32 byte line size
50h	Instruction TLB: 4 kB, 2 MB or 4 MB pages, 64 entries
51h	Instruction TLB: 4 kB, 2 MB or 4 MB pages, 128 entries
52h	Instruction TLB: 4 kB, 2 MB or 4 MB pages, 256 entries
5Bh	Data TLB: 4 kB and 4 MB pages, 64 entries
5Ch	Data TLB: 4 kB and 4 MB pages, 128 entries
5Dh	Data TLB: 4 kB and 4 MB pages, 256 entries
66h	1st-level data cache: 8 KB, 4-way set associative, 64 byte line size
67h	1st-level data cache: 16 KB, 4-way set associative, 64 byte line size
68h	1st-level data cache: 32 KB, 4-way set associative, 64 byte line size
70h	Instruction Trace cache, 8 way set associative, 12K µOps
71h	Instruction Trace cache, 8 way set associative, 16K µOps
72h	Instruction Trace cache, 8 way set associative, 32K µOps
79h	unified 2nd-level cache, 128 KB, 8 way set associative, 64 byte cache line, sectored
7Ah	unified 2nd-level cache, 256 KB, 8 way set associative, 64 byte cache line, sectored
7Bh	unified 2nd-level cache, 512 B, 8 way set associative, 64 byte cache line, sectored
7Ch	unified 2nd-level cache, 1 MB, 8 way set associative, 64 byte cache line, sectored

Tabelle 1.10: Codes für Eigenschaften von caches und translation look-aside buffers (TLBs) des Prozessors

Code	Beschreibung
82h	unified 2nd-level cache, 256K Bytes, 8 way set associative, 32 byte cache line
83h	unified 2nd-level cache, 512K, Bytes 8 way set associative, 32 byte cache line
84h	unified 2nd-level cache, 1M Bytes, 8 way set associative, 32 byte cache line
85h	unified 2nd-level cache, 2M Bytes, 8 way set associative, 32 byte cache line

Tabelle 1.10: Codes für Eigenschaften von caches und translation look-aside buffers (TLBs) des Prozessors (Forts.)



Bitte beachten Sie, dass die Codes in EAX, EBX, ECX und EDX nicht notwendigerweise in der Reihenfolge ihrer Werte verzeichnet sein müssen. Das bedeutet, die Register können wahllos Codes aus dieser Tabelle aufnehmen.

Ein Beispiel: Der erste Prozessor der Pentium-4-Familie liefert folgende Daten zurück, wenn man Funktion 2 des CPUID-Befehls aufruft:

EAX: $665B5001; EBX: $00000000; ECX: $00000000; EDX: $007A7000.

Das bedeutet: Alle Registerinhalte sind valide (gelöschte Bits 31!) und die Funktion muss nur einmal aufgerufen werden, um alle Informationen zu erhalten (AL = $01). Ansonsten werden die Codes $50, $5B, $66, $70 und $7A zurückgegeben, was bedeutet, dass der Prozessor einen Instruction-TLB mit 64 Einträgen zum Mappen von 4-k-, 2-M- oder 4-MByte-Pages hat, einen Daten-TLB mit 64 Einträgen für 4-k- bzw. 4-MByte-Pages, einen 8-kByte-1st-Level-Daten-Cache mit einer cache line size von 64 kByte, einen 12k-µop-Trace-Cache und einen 256-kByte-2nd-Level-Cache mit einer in Sektoren aufgeteilten cache line size von 64 Byte.

Funktion 3 (Basisfunktion)

Auch Funktion 3 ist nur auf Intel-Prozessoren realisiert und hier auch nur bei wenigen. Sie gibt einen Teil der Seriennummer des Prozessors zurück. Diese Seriennummer besteht aus 96 Bits und wird aus der mittels Funktion 1 feststellbaren processor signatur (EAX) und den in der Registerkombination EDX:ECX zurückgegebenen Werten der Funktion 3 gebildet, wobei Bit 63 der Seriennummer durch Bit 31 in EDX repräsentiert wird und Bit 0 der Seriennummer durch Bit 0 in ECX. Die Bits 95 bis 64 stammen aus der processor signature der Funktion 1. Die Inhalte von EAX und EBX sind durch Intel reserviert. Die Seriennummer besteht aus Hexadezimalzeichen, was bedeutet, dass alle Nibbles Werte zwischen 0_h und F_h annehmen können.



Funktion 3 wird nur durch den Pentium III (und hier auch nicht von allen!) unterstützt, da die Implementation auf massiven Widerstand bei den Kunden und Bedenken bei Datenschützern stieß. Intel hat daraufhin Seriennummern in den folgenden Prozessoren nicht mehr realisiert und bei neueren Pentium-III-Prozessoren die Funktion deaktiviert. Daher sollte in jedem Fall geprüft werden, ob das feature verfügbar ist. Zuständig hierfür ist ein gesetztes Flag PSN in den feature flags.

Funktion 8000 (erweitert)

Analog der Basisfunktion 0 gibt auch die »erweiterte« Funktion 0 ($8000) in EAX die höchste Nummer der verfügbaren erweiterten Funktionen zurück. Die Inhalte von EBX, ECX und EDX sind reserviert. Da die Erweiterung der Funktionalität des CPUID-Befehls evolutionär erfolgt, besitzen nicht alle Prozessoren solche erweiterten Funktionen. Ob sie verfügbar sind, kann jedoch einfach festgestellt werden: Wird der CPUID-Befehl mit dem Wert $8000 in EAX aufgerufen (also praktisch die erweiterte Funktion 0), so muss der in EAX zurückgegebene Wert größer als $8000 sein. Ist dies nicht der Fall, sind auf dem Prozessor keine erweiterten Funktionen aufrufbar.

Funktion 8001 (erweitert)

Die erweiterte Funktion 1 ($8001) gibt analog der Basisfunktion 1 weitere Informationen zum Prozessor zurück: In EAX liegt analog zur Standard-Funktion eine »AMD processor signature«, in EDX »extended feature flags«.



Bitte beachten Sie, dass Intel selbst im Pentium 4 die erweiterte Funktion 1 ($8001) nicht realisiert (»currently reserved«), ihre Existenz jedoch bereits dokumentiert hat (»extended feature bits«). AMD dagegen nutzt bereits seit einigen Modellen des K6 (K6-2, Modell 8 und K6-III, Modell 9) diese Funktion für eine von der Basisfunktion 1 abweichende Angabe zum Prozessor (Abbildung 1.19; vgl. Abbildung 1.15) ...

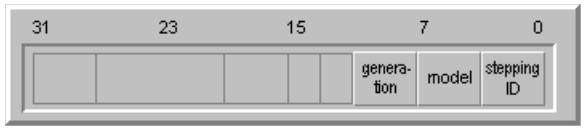


Abbildung 1.19: Speicherabbild des Registers EAX nach Aufruf der Funktion 8001 des CPUID-Befehls bei AMD-Prozessoren

... und der »extended feature flags«. Die Flags 0 bis 9, 12 bis 17 und 23 und 24 sind hierbei identisch mit denen aus den feature flags der Funktion 1 (vgl. Abbildung 1.17), wie Abbildung 1.20 zeigt.

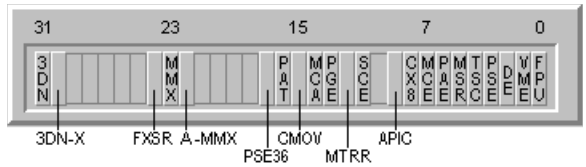


Abbildung 1.20: Speicherabbild des Registers EDX nach Aufruf der Funktion 8001 des CPUID-Befehls bei AMD-Prozessoren

Hinzugekommen ist bzw. verändert gegenüber den »Standard-Feature-Flags« wurde:

3DN — Bit 31: 3DNow!-Erweiterungen werden unterstützt.

3DN-X — Bit 30; erweiterte 3DNow!-Erweiterungen werden unterstützt.

A-MMX — Bit 22; erweiterte MMX-Erweiterungen werden unterstützt, die mit Intels SSE-Befehlssatz eingeführt wurden.

SCE — Bit 11; anstelle des Flag SEP (Funktion $0001), das die Verfügbarkeit der Befehle SYSENTER/SYSEXIT signalisiert, steht hier das Flag SCE, *system call extension*, das die Verfügbarkeit der AMD-spezifischen Befehle SYSCALL und SYSRET signalisiert.

Funktion 8002
Funktion 8003
Funktion 8004
(erweitert)

Mit den erweiterten Funktionen 2 bis 4 ($8002 bis $8004) ist es möglich, einen brand string auszulesen. Dieser String enthält den vom Prozessorhersteller definierten offiziellen Namen des Prozessors und evtl. seine maximale Taktfrequenz. Diese Frequenz muss nicht notwendigerweise die Frequenz sein, mit der der Prozessor auf dem Board betrieben wird!

Der brand string ist ggf. rechtsbündig ausgerichtet, weshalb führende Leerstellen möglich sind. Die Information ist ASCII-codiert, umfasst 47 Byte und ein abschließendes Null-Byte (wie bei Strings üblich). Die 48 Byte verteilen sich auf drei Funktionen mit je 16 Byte, die in den vier Allzweckregistern in Form von DoubleWords zurückgegeben werden:

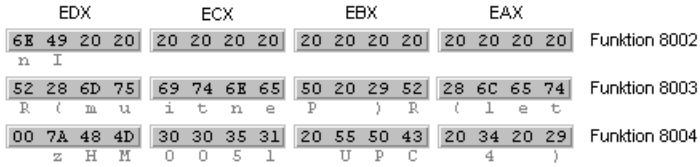


Abbildung 1.21: Beispiel eines Speicherabbildes der Register EAX, EBX, ECX und EDX nach Aufruf des CPUID-Befehls mit den Funktionen 8002, 8003 und 8004

Liest man somit die Register von rechts nach links und in der Reihenfolge EAX:EBX:ECX:EDX sowie beginnend mit Funktion 8002 aus (»Intel-Notation«), so erhält man den brand string. Dies ist in Abbildung 1.21 für den ersten Pentium 4 dargestellt. Er liefert den Null-terminierten String »`Intel(R) Pentium(R) 4 CPU 1500 MHz`« rechtsbündig (d. h. mit 14 führenden Leerstellen) zurück. Ein AMD Athlon MP, Model 6, meldet sich mit »`AMD Athlon(tm)` MP processor« linksbündig (d. h. ohne führende Leerstellen) und mit aufgefüllten Null-Bytes.

Funktion 8005 Funktion 8006 (erweitert)

Funktion $8005 und $8006 sind zurzeit nur bei AMD-Prozessoren realisiert. Funktion $8005 liefert Informationen zum 1st-level cache und TLB (translation lookaside buffer) zurück, Funktion $8006 die gleichen Informationen für den 2nd-level cache und TLB zurück.

Bei beiden Funktionen stehen in EAX Informationen bei Verwendung von 4-MByte bzw. 2-MByte-Pages, in EBX die Informationen bei Verwendung von 4-kByte-Pages. In Abbildung 1.22 sind die jeweiligen Informationen dargestellt: Die Bits 15 bis 0 betreffen den instruction translation lookaside buffer und geben die Anzahl der Einträge (Bits 7 bis 0) und die Assoziativität (Bits 15 bis 8) an, in den Bits 31 bis 16 stehen die gleichen Informationen zum data TLB.

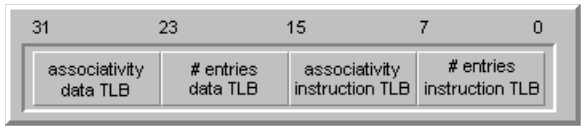


Abbildung 1.22: Speicherabbild der Register EAX und EBX nach Aufruf des CPUID-Befehls mit den Funktionen 8005 und 8006 bei AMD-Prozessoren

In ECX liefert Funktion $8005 Informationen zum 1st-level data cache und in EDX zum 1st-level instruction cache zurück. Hierbei handelt es sich, wie Abbildung 1.23 zeigt, um die Größe der cache line in Bytes (Bits 7 bis 0), die Anzahl der lines per tag (Bits 15 bis 8), die Größe des Cache in kBytes (Bits 31 bis 24) sowie die Assoziativität (Bits 23 bis 16).

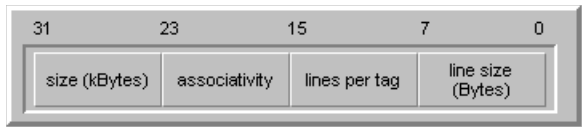


Abbildung 1.23: Speicherabbild der Register ECX und EDX nach Aufruf des CPUID-Befehls mit den Funktionen 8005 und 8006 bei AMD-Prozessoren

Funktion $8006 benutzt hier nur das ECX-Register für die entsprechenden Informationen für den unified 2nd-level Cache. Der Inhalt von EDX gilt hier als reserviert.

In allen Registern besteht das Feld associativity bei Funktion $8005 aus 8 Bits. Der dadurch definierte Wert stellt mit zwei Ausnahmen den tatsächlichen Grad der Assoziativität dar. Die Ausnahmen sind 00_h: *reserviert*, und FF_h: *full associativity* Ein Wert von 02_h steht somit für *2-way associativity*, ein Wert von 08_h für *8-way associativity*.

Bei Funktion $8006 werden in allen Registern nur die vier »unteren« der acht Bits für die Darstellung der Assoziativität benutzt. Die Bedeutungen der Werte werden in Tabelle 1.11 genannt.

Wert	Assoziativität	Wert	Assoziativität	Wert	Assoziativität	Wert	Assoziativität
0_h	2nd level off	4_h	4-way	8_h	16-way	C_h	reserved
1_h	direct mapped	5_h	reserved	9_h	reserved	D_h	reserved
2_h	2-way	6_h	8-way	A_h	reserved	E_h	reserved
3_h	reserved	7_h	reserved	B_h	reserved	F_h	full

Tabelle 1.11: Werte des Feldes associativity nach Aufruf der Funktion $8006 und ihre Bedeutung



Die in EAX zurückgegebenen Werte für die Anzahl der Einträge (»# entries«) beziehen sich immer auf 2-MByte-Pages. Da bei Verwendung von 4-MByte-Pages zwei 2-MByte-Entries pro 4-MByte-Page benötigt werden, muss der zurückgegebene Wert in diesem Fall mit 1.5 (!) multipliziert werden, um die korrekte Anzahl verfügbarer Einträge zu erhalten.

Ein unified 2nd-level TLB wird bei Funktion $8006 dargestellt, indem die »oberen« 16 Bits des EBX-Registers (»data TLB«) auf Null gesetzt sind. Die Informationen zum unified TLB sind dann in den Bits 15 bis 0 (»instruction TLB«) enthalten.

AMDs K5- und K6-Prozessoren unterstützen keine 2-MByte- bzw. 4-MByte-Pages. Daher können sie auch keine Informationen hierzu zurückgeben, weshalb der Inhalt des EAX-Registers in diesem Fall als reserviert gilt. Die Informationen zum TLB sind dann EBX zu entnehmen.



Aus den gleichen Gründen gibt es nur für K6-III-, Athlon- und Duron-Prozessoren Informationen zum 2nd-level TLB (Athlon und Duron) und zum 2nd-level cache (Athlon, Duron, F6-III). Im Falle des K6-III sind die Register EAX und EBX reserviert, in ECX steht die Information zum 2nd-level cache.

Funktion 8007 (erweitert)

Funktion $8007 liefert bei AMD-Prozessoren, die in »mobilen« Systemen eingesetzt werden, Informationen zum »advanced power management«.

Derzeit gelten die Inhalte der Register EAX, EBX und ECX als reserviert, sodass nur in EDX verwertbare Daten stehen. Hierbei handelt es sich um die advanced power management feature bits.

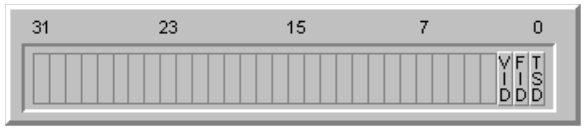


Abbildung 1.24: Speicherabbild des Registers EDX nach Aufruf der Funktion 8007 des CPUID-Befehls bei AMD-Prozessoren

Es bedeuten:

TSD	temperature sensing diode; der Prozessor besitzt eine temperaturempfindliche Diode zur Temperaturüberwachung der CPU.
FID	frequency ID control
VID	voltage ID control

Funktion 8008 (erweitert)

Funktion $8008 ermöglicht Informationen zur physikalischen Adresse und zur Größe der linearen Adresse bei AMD-Prozessoren.

Die Inhalte der Register EBX, ECX und EDX gelten als reserviert, in EAX steht an den Bitpositionen 15 bis 8 die maximale virtuelle Adresse und an den Bitpositionen 7 bis 0 die maximale physikalische Adresse, jeweils angegeben als Anzahl zur Codierung verwendeter Bits. Die Angabe $0000_2022, die hier z.B. ein AMD Athlon MP, Model 6 zurückgibt,

heißt: Es werden 32 (=$20) Bits für die maximale virtuelle Adresse und 34 (=$22) Bits zur Berechnung physikalischer Adressen verwendet.

Operanden Der CPUID-Befehl verwendet keine expliziten Operanden. Allerdings erwartet er in EAX eine Funktionsnummer, mit der die verfügbaren Funktionen des Befehls aufgerufen werden. Die Ergebnisse der Funktionen werden in den Registern EAX, EBX, ECX und EDX zurückgegeben.

Statusflags CPUID verändert keine Statusflags.

ENTER
LEAVE ENTER und LEAVE sind zwei Befehle, die ein Befehlspaar bilden. Sie dienen dazu, einen Stack-Rahmen einzurichten und zu entfernen. ENTER und LEAVE realisieren dabei Befehlssequenzen, die auch mit anderen CPU-Befehlen realisiert werden können.



Zum Verständnis der Arbeitsweise dieser beiden Befehle ist es wichtig, zu wissen, was »der Stack« ist und um was es sich bei »stack frames« handelt. Informationen hierzu finden Sie im Kapitel »Stack« auf Seite 385.

ENTER ENTER ist der Befehl des Befehlspaares, der für die Einrichtung des Stack-Rahmens zuständig ist. Als Stack-Rahmen wird ein Bereich des Stacks bezeichnet, den der Prozessor (und der Programmierer) im Kontext des aktuellen (Unter-) Programms nutzen kann, der sozusagen der »private« Bereich des (Unter-) Programms auf dem Stack darstellt. Hierbei ist privat durchaus als *privat* zu verstehen: In der Regel kann bei verschachtelten (Unter-)Programmen ein Unterprogramm nicht auf die lokalen Variablen (das sind die, die auf dem Stack liegen!) des ihm »übergeordneten« (Unter-)Programms zugreifen, da ihm die Informationen fehlen, wo dessen Stack-Rahmen zu finden ist (vgl. auch »Stack« auf Seite 385). ENTER bietet hier einen Ausweg. Es ist ein sehr vielseitiger Befehl:

Möglichkeit 1 Zum einen kann ENTER »klassisch« eingesetzt werden, was bedeutet, dass keine »Verschachtelungen« von Unterprogrammen in verschiedenen »Ebenen« erfolgen. In diesem Fall simuliert der Befehl ENTER, der immer der erste Befehl der Routine sein sollte, die mittels CALL aufgerufen wird (»Unterprogrammaufruf«), was man auch bei der Einrichtung eines Stack-Rahmens »von Hand« programmieren würde (vgl. Seite 389):

```
PUSH    (E)BP
MOV     (E)BP, (E)SP
SUB     (E)SP, Size
```

ENTER rettet somit zunächst den aktuellen Inhalt des Stack-Basisregisters (E)BP auf den Stack und deklariert die bisherige Stackspitze (Inhalt des (E)SP-Registers) als neue Stackbasis. Anschließend wird die neue Stackspitze anhand der als Operand übergebenen Größe des Stacks gesetzt: Ein neuer Stack-Rahmen wurde definiert. Und wie man sieht, stehen in diesem Stack-Rahmen keinerlei Informationen über etwaige »übergeordnete« Programmstrukturen.

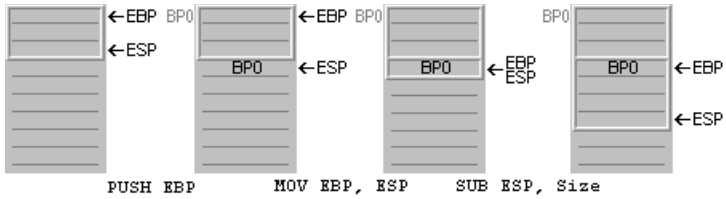


Abbildung 1.25: Aufbau eines Stack-Rahmens durch ENTER mit einem nesting level von 0

Abbildung 1.25 zeigt, wie man »von Hand« einen Stack-Rahmen aufbauen würde. Die gleichen Aktionen laufen ab, wenn der Befehl ENTER in der Möglichkeit 1 mit einem nesting level von »0« aufgerufen wird.



Achtung! ENTER ist ein Befehl, der mit zwei unterschiedlichen Segmenten zurechtkommen muss: dem Stacksegment und dem Datensegment. Beide Segmente haben ein Größenattribut (»Umgebung«). So sind in 32-Bit-Umgebungen sowohl Stack- als auch Datensegment in der Regel 32-bittig, in 16-Bit-Umgebungen 16-bittig ausgelegt. Doch es können auch unterschiedliche Attribute vorherrschen (z.B. aufgrund alter 16-Bit-Windows-3.x-Programme in 32-Bit-Windows-Umgebungen). Sobald das Größenattribut des Stacksegments (das B-Flag im Deskriptor) eine 32-Bit-Größe signalisiert, arbeitet ENTER (und natürlich auch LEAVE!) mit den Registern EBP und ESP. Ist es dagegen gelöscht, wird mit den 16-Bit-Registern BP und SP gearbeitet. Der Wert, um den die Pointer dann jedoch inkrementiert/dekrementiert werden, wird von der Größe der standardmäßig bearbeiteten Daten vorgegeben. Und diese Größe ist das Größenattribut des Datensegments. Das führt z.B. beim SUB-Befehl in der Schleife zu folgenden vier Fällen:

- 32-Bit-Stacksegment, 32-Bit-Datensegment: `SUB EBP, 4;`
- 32-Bit-Stacksegment, 16-Bit-Datensegment: `SUB EBP, 2;`
- 16-Bit-Stacksegment, 32-Bit-Datensegment: `SUB BP, 4;`
- 16-Bit-Stacksegment, 16-Bit-Datensegment: `SUB BP, 2;`

Analoges gilt für die anderen Befehle, die von ENTER/LEAVE simuliert werden!

Möglichkeit 2 Allerdings kann ENTER auch eingesetzt werden, um »verschachtelte« Unterprogrammebenen zu definieren. Sie zeichnen sich dadurch aus, dass in jeder Ebene und für jede Ebene Zeiger auf dem Stack liegen, die die lokalen Parameter der jeweils in der Hierarchie »übergeordneten« Ebenen für die aktuelle Ebene »sichtbar« machen. Hierzu wird dem Befehl ENTER neben der Größe des einzurichtenden Stack-Rahmens auch die Verschachtelungstiefe (nesting level) übergeben.

Hierbei hat die »oberste« Ebene (das »Hauptprogramm«) die Stufe »1«. Übergibt man ENTER diesen Wert als »Tiefe«, führt es folgende Aktionen durch:

```
PUSH    (E)BP
MOV     Temp, (E)SP
PUSH    Temp
MOV     (E)BP, Temp
SUB     (E)SP, Size
```

Das bedeutet, dass nun ein neuer Stack-Rahmen eingerichtet wurde, in dem an der Basis hinter der Adresse der Stackbasis des »übergeordneten« Programmcodes ein Zeiger auf die Adresse der eigenen Stackbasis als zusätzliche lokale Variable eingetragen wurde. Welche Bedeutung das hat, werden wir sehen, wenn man ENTER mit dem Wert für eine »niedrigere« Verschachtelungsebene (»`nesting level`«) aufruft. ENTER führt dann folgende Aktionen aus:

```
PUSH    (E)BP
MOV     Temp, (E)SP
for I := 1 to (Level-1) do
SUB     (E)BP, 4 (2)
PUSH    [(E)BP]
PUSH    Temp
MOV     (E)BP, Temp
SUB     (E)SP, Size
```

Hinzugekommen im Vergleich zum Vorgehen bei Stufe 1 sind die kursiv dargestellten Zeilen. Mit ihnen werden nach dem obligatorischen Zeiger auf die Stackbasis des »übergeordneten« Programmcodes (vgl. Abbildung 1.25) weitere Zeiger als lokale Variablen auf den Stack gebracht. Die Anzahl entspricht hierbei der Verschachtelungsebene – 1. Erst dann wird die Basisadresse des eigenen Stack-Rahmens auf den Stack geschoben.

Wozu nun soll dieses Vorgehen gut sein? Betrachten wir dazu einmal ein Beispiel. Gegeben sei ein »Hauptprogramm« (Ebene 1), dessen Stack-Rahmen mittels ENTER eingerichtet worden ist. Dieses Hauptprogramm besitzt zwei lokale Variablen, sodass ENTER neben dem Level der Wert 8 (= zwei lokale DoubleWord-Variable à vier Byte) als Parameter übergeben wurde. Das Hauptprogramm ruft irgendwann einmal ein Unterprogramm auf (somit Ebene 2), das selbst drei lokale Variablen hat, weshalb dem Befehl ENTER hier die Parameter 2 (= level) und 12 (= drei DoubleWord-Variablen) übergeben wurden. Das Unterprogramm, nennen wir es A, besitzt selbst wieder ein Unterprogramm, B, weshalb wir bei der Einrichtung des Stack-Rahmens für B von einer »Verschachtelungstiefe« von 3 ausgehen und dem Befehl ENTER als Level somit 3 übergeben müssen. B habe vier lokale DoubleWord-Variablen.

Verfolgen wir nun anhand von Abbildung 1.26 einmal, wie ENTER funktioniert.

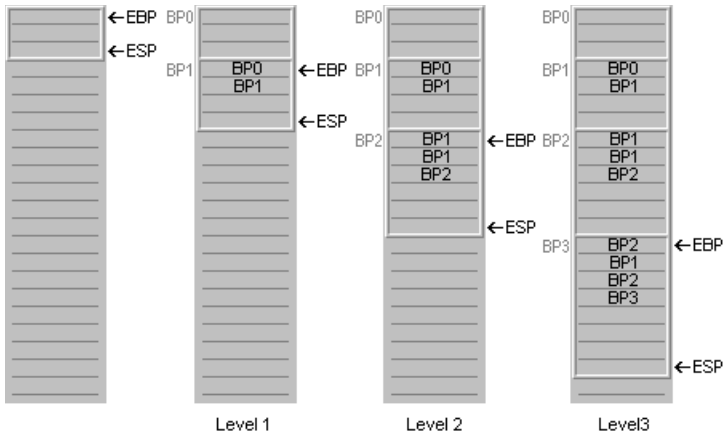


Abbildung 1.26: Darstellung der durch den Befehl ENTER hergestellten stack frames

Wir gehen von dem Zustand aus, dass ein Stack-Rahmen existiert, der zu dem Programmteil gehört, das das Hauptprogramm aufrufen wird (Abbildung 1.26, links). Sobald der Prozessor nach dem Sprung in das Hauptprogramm auf den Befehl ENTER stößt, richtet er den Stack-Rahmen für das Hauptprogramm ein. ENTER war für *nesting level* der Wert 0 und als Platzbedarf für lokale Variable der Wert 8 übergeben worden.

Gemäß der Beschreibung der Aktionen von ENTER für diesen Fall weiter oben wird nun als Erstes der Inhalt von EBP mittels `PUSH EBP` auf den Stack geschoben. EBP zeigt aber auf die Stackbasis der rufenden Routine, in der Abbildung mit *BP0* markiert. Die Spitze des Stacks (ESP) zeigt nun auf diesen Wert (der PUSH-Befehl hat ja dekrementiert!). Diese Adresse wird jetzt zum einen temporär gespeichert, zum anderen auf den Stack gePUSHt. Sodann wird die temporär gespeicherte Adresse in das EBP-Register kopiert, was sie zur Stackbasis des neuen, dem Hauptprogramm zugeordneten Stack-Rahmen macht. Das bedeutet: ganz »unten« (bitte beachten Sie, dass der Stack von oben nach unten wächst!) im neuen Stack-Rahmen liegt nun die Basisadresse der »übergeordneten« Routine, die das Hauptprogramm aufgerufen hat, gefolgt von der Adresse der eigenen Stackbasis. ESP zeigt zu diesem Zeitpunkt auf diese lokale Kopie. Abschließend schafft ENTER noch Platz für die lokalen Variablen, indem es den Stackpointer ESP um die als Parameter übergebene Anzahl Bytes dekrementiert: `SUB ESP, Size`. Das Ergebnis sehen Sie in Abbildung 1.26, Mitte links.

Geht man nun eine Ebene »tiefer«, heißt: rufen wir ein in das Hauptprogramm eingebettetes (»verschachteltes«) Unterprogramm auf, wird als nesting level der Wert 2 (= Ebene 2) übergeben. Nun rettet ENTER wiederum zunächst den Inhalt des EBP-Registers (die Stackbasis der »übergeordneten« Routine) auf den Stack und trägt die resultierende Adresse aus ESP in einen temporären Speicher ein. Dann aber wird eine Schleife aufgerufen, die (nesting level – 1)-mal, hier also einmal durchlaufen wird. Sie subtrahiert mittels `SUB EBP, 4` von der in EBP stehenden Adresse, der Stackbasis der übergeordneten Routine, jeweils die Anzahl Bytes für ein Doppelwort. Damit zeigt aber EBP auf die dort stehende(n) Adresse(n) der Stackbasen/-basis der übergeordneten Routine(n). Diese Adresse(n) werden/wird auf den Stack gePUSHt. Bitte beachten Sie: Nicht die in EBP stehende Adresse wird gePUSHt, sondern der Wert, der auf dem Stack an der in EBP stehenden Adresse steht (indirekte Adressierung!).

Abschließend wird der temporär gespeicherte Wert aus ESP (also die Adresse der neuen Stackbasis!) in EBP kopiert und Platz für lokale Variablen geschaffen. Das kennen wir bereits. Das Ergebnis ist in Abbildung 1.26, Mitte rechts, für eine Routine mit Verschachtelungstiefe 2 und in Abbildung 1.26, rechts, für eine mit Verschachtelungstiefe 3 (Unterprogramm eines Unterprogramms des Hauptprogramms) dargestellt.



Fasst man zusammen, was ENTER in Abhängigkeit der Tiefe der Verschachtelung so tut, lässt sich festhalten:

- An [EBP] steht die Adresse der Stackbasis der übergeordneten Routine; diese wird vom Befehl LEAVE benutzt, den Stack-Rahmen wieder zu entfernen.
- Für jede Verschachtelungstiefe steht an [EBP + (*nesting level* * 4)] die Adresse der Stackbasis der Routine mit der Hierarchiestufe *nesting level*.
- Ab EBP + ((*nesting level* +1) * 4) bis ESP stehen dann die lokalen Variablen der Routine, sofern vorhanden.

Auf diese Weise kann jede Routine nicht nur auf die eigenen lokalen Variablen zurückgreifen, sondern im Rahmen indirekter Adressierung auch auf die aller in der Hierarchie höher stehenden Routinen.

LEAVE

Verglichen mit ENTER ist LEAVE ein sehr einfacher Befehl. LEAVE dient dazu, alle Veränderungen rückgängig zu machen, die ENTER vorgenommen hat, und wird somit zur Entfernung des Stack-Rahmens unmittelbar vor dem Rücksprung aus dem Unterprogramm aufgerufen.

LEAVE führt hierbei die Vorgänge durch, die man auch »von Hand« programmieren würde, um einen Stack-Rahmen zu entfernen:

```
MOV     (E)SP, (E)BP
POP     (E)BP
```

Abbildung 1.27 zeigt, was diese Befehle auf dem Stack bewirken. Es wird von der Situation ausgegangen, die in Abbildung 1.26 nach Einrichten eines Stack-Rahmens mit der Verschachtelungstiefe 3 durch ENTER eingerichtet wurde. Die aktuelle Stackbasis wird zur Stackspitze deklariert und der Wert, der an der neuen Stackspitze steht, als neue Stackbasis in das (E)BP-Register gePOPpt. Da dadurch auch das (E)SP-Register inkrementiert wird, hält (E)SP nach Abschluss von LEAVE tatsächlich die Stackspitze des »übergeordneten« Stack-Rahmens.



Bitte beachten Sie, dass auf diese Weise zwar der Stack-Rahmen entfernt wird, nicht aber die Werte an den entsprechenden Adressen gelöscht werden. Sie sind somit noch physikalisch vorhanden. Nichtsdestotrotz gelten sie als entfernt und Sie sollten tunlichst vermeiden, auf sie zurückgreifen zu wollen. Da nämlich der aktuelle Stack-Rahmen neu gesetzt wurde, kann es sein, dass z.B. im Rahmen von Interrupts,

die nach der Entfernung des Stack-Rahmens auftreten, die in Abbildung 1.27 grau dargestellten Werte überschrieben werden, ohne dass Sie das merken!

Beherzigen Sie daher: *Werte »oberhalb« der aktuellen Stackspitze (also bei niedrigeren Adressen als im (E)SP-Register verzeichnet) sind nicht existent!*

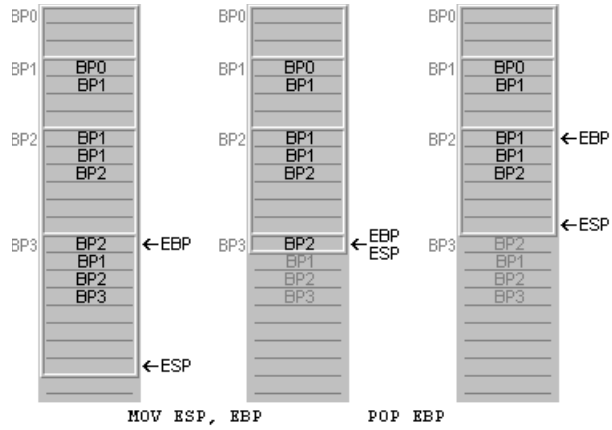


Abbildung 1.27: Entfernen eines Stack-Rahmens mittels LEAVE

Das Entfernen aller anderen stack frames in Abbildung 1.27 erfolgt absolut identisch.



Hier wird nun auch spätestens klar, warum ENTER an die Basis jedes neuen Stack-Rahmens in jedem Fall die Adresse der Basis des übergeordneten Rahmens schreibt: Durch alleiniges Deklarieren der Stackbasis zur Stackspitze ist mit einem einfachen POP die neue (= alte) Stackbasis rückladbar. Das bedeutet, LEAVE benötigt keinerlei Informationen über einen nesting level, um den Stack-Rahmen entfernen zu können. Würde diese zunächst überflüssig erscheinende redundante Ablage der Basisadresse der jeweils direkt übergeordneten Routine nicht erfolgen, müsste LEAVE ein Parameter mit dem aktuellen nesting level übergeben werden, um gemäß der Formel

*EBP := [EBP + (nesting level * 4)]*

die entsprechende Adresse finden zu können.

LEAVE verwendet keine Operanden, ENTER besitzt zwei Operanden, die die Verschachtelungstiefe und den Platzbedarf für lokale Variablen in Bytes übergeben: *Operanden*

```
ENTER Const16, Const8
```

Der erste Operand enthält hierbei die Größe des Stackbereichs in Byte, der für lokale Variablen reserviert werden soll. Da hier nur eine 16-Bit-Konstante übergeben werden kann, können maximal 65.536 Bytes lokal reserviert werden. Der zweite Operand gibt die Tiefe der Verschachtelung der Routine an. Es sind Werte zwischen 0 und 31 erlaubt, was bedeutet, dass die maximale Verschachtelung 31 Ebenen umfassen kann (weshalb auch ENTER den in `Const8` übergebenen Wert modulo 32 nimmt!). Ist dieser Wert Null, wird keine Verschachtelung angenommen und es wird ein »normaler« Stack-Rahmen erzeugt. Bei allen Werten über Null wird der Stack-Rahmen pro Verschachtelungsebene um einen Eintrag vergrößert. Diese zusätzlichen lokalen Parameter nehmen Zeiger auf die Stack-Rahmen der übergeordneten Routinen (= Routinen mit niedrigerem nesting level) auf, sodass über indirekte Adressierung auch auf die lokalen Parameter »höherer« Routinen zugegriffen werden kann.

ENTER und LEAVE verändern die Statusflags nicht. *Statusflags*

NOP, *no operation*, ist der geeignete Befehl für einen *lazy Sunday afternoon*. Er macht: nichts! Im wahrsten Sinne des Worte. NOPs werden in der Regel als Platzhalter oder für bestimmte spezielle Zwecke eingesetzt. **NOP**

NOP ist ein Alias für den Befehl *XCHG (E)AX, (E)AX*. Der macht auch nichts!



NOP besitzt keine Operanden. Daher wird es wie folgt aufgerufen: *Operanden*

```
NOP
```

NOP verändert keine Statusflags. *Statusflags*

WAIT ist ein Alias für FWAIT und wird im Rahmen der FPU-Befehle besprochen. **WAIT**

WAIT besitzt keine Operanden. Daher wird es wie folgt aufgerufen: *Operanden*

```
WAIT
```

WAIT verändert keine Statusflags. *Statusflags*

UD2 UD2, *undefined instruction*, generiert eine invalid opcode exception (#UD). Ansonsten macht es ebenso wie NOP nichts! Der Zweck von UD2 liegt in der gezielten Auslösung von exceptions zum Testen von Software.

Operanden UD2 hat keine Operanden. Daher wird es wie folgt aufgerufen:

```
UD2
```

Statusflags US2 verändert keine Statusflags.

1.1.15 Verwaltungs-(System-)Befehle

Systembefehle sind Befehle, die in irgendeiner Weise mit der Verwaltung des Betriebsmodus des Prozessors und damit auch irgendwie mit dem Betriebssystem zu tun haben. Als Lieschen oder Otto Normalassemblierer werden Sie wohl kaum in die Verlegenheit kommen, diese Befehle zu benutzen – ganz abgesehen davon, dass viele von ihnen bestimmte Privilegien erfordern, die das Betriebssystem Ihnen in der Regel nicht gewährt. Die »privilegierten« Befehle aus dem Systembefehlssatz werden Ihnen weiter unten genannt.



Es ist hilfreich, wenn Sie vor dem Lesen dieses Abschnitts die Kapitel »Speicherverwaltung« auf Seite 394 und »Schutzmechanismen« auf Seite 467 durchgelesen haben. Die folgenden Befehle sind im Kontext mit Informationen aus diesen Kapitel erst verständlich und nachvollziehbar.

System-Tabellen und Tasks Zur Unterstützung des Betriebsystems gibt es mit der global descriptor table, der local descriptor table und der interrupt descriptor table drei Systemtabellen, deren Adressen in speziellen Registern gehalten werden. Hinzu kommt noch ein Register, in dem die Adresse einer Datenstruktur gehalten wird, die den aktuellen Task beschreibt. Um diese Register verwalten zu können, gibt es die folgenden Befehle.

LGDT SGDT LIDT SIDT *Load global descriptor table* (LGDT) und *store global descriptor table* (SGDT) sowie die analogen Befehle *load interrupt descriptor table* (LIDT) und *store interrupt descriptor table* (SIDT) sind vier wesentliche Befehle, ohne die das Betriebssystem nicht auskäme. Mit ihnen kann die Basisadresse sowie die Größe der »zentralen« *global descriptor table* (GDT) bzw. der nicht weniger wichtigen interrupt descriptor table (IDT) in das jeweils dafür vorgesehene Register (GDTR, *global descriptor table register* bzw.

IDTR, *interrupt descriptor table register*) geschrieben oder aus ihm ausgelesen werden.

Diese Befehle sind die einzigen Systembefehle, die im protected mode reale, lineare 32-Bit-Adressen verwenden. Alle anderen Befehle, die mit Adressen umgehen, bedienen sich logischer Adressen, also der Kombination Segment-Selektor: effektive Adresse bzw. der effektiven Adresse selbst.



LGDT und LIDT, nicht aber SGDT und SIDT, sind privilegierte Befehle, was bedeutet, dass sie nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden können. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt unweigerlich zu einer general protection exception #GP.

Operanden

Als Operand erwarten alle Befehle einen Zeiger auf eine Sechs-Byte-Struktur, in der die Basisadresse und das Limit verzeichnet sind bzw. in die die Daten eingetragen werden können:

```
LGDT Mem48; SGDT Mem48; LIDT Mem48; SIDT Mem48
```

Die Belegung des angegebenen Speicherbereiches ist abhängig von der aktuellen Operandengröße, die z.B. im Deskriptor des verwendeten Datensegments in Form des big flags definiert ist und mittels des operand size override prefix verändert werden kann. So signalisiert ein gesetztes big flag eine Standard-Operandengröße von 32 Bit. In diesem Fall oder wenn das big flag gelöscht ist und das operand size override prefix verwendet wird (16-Bit-Standard-Operandengröße, jedoch mittels des Präfix auf 32 Bits gesetzt!), findet sich in den Bytes 0 und 1 des Operanden das 16-Bit-Tabellenlimit. Die Bytes 2 bis 5 des Operanden beherbergen dann die (virtuelle) 32-Bit-Basisadresse der Tabelle.

Ist dagegen das big flag im Datensegment-Deskriptor gelöscht (16-Bit-Standard-Operandengröße) oder ist es gesetzt und es wird der operand size override prefix verwendet (32-Bit-Standard-Operandengröße, durch den Präfix reduziert auf 16 Bits), besitzt die Datenstruktur immer noch 6 Bytes Umfang und das 16-Bit-Tabellenlimit befindet sich in Byte 0 und 1 des Operanden. In diesem Fall jedoch werden nur die Bytes 2 bis 4 des Operanden verwendet, die die 24-Bit-Basisadresse der Tabelle beinhalten. Byte 5 des Operanden ist dann auf Null gesetzt.

Statusflags

Keiner der vier Befehle verändert die Statusflags.



Die GDT und die IDT sind Tabellen, ohne die der protected mode nicht ausgeführt werden kann. Daher ist es wichtig, sie bereits vor dem Eintritt in den protected mode zu generieren. LDTR und GDTR sind daher Befehle, die im Rahmen des protected mode eine Rolle spielen, aber ihre Hauptfunktion im real mode haben: Dort werden sie verwendet, um dem Prozessor die Lage der beiden Tabellen zu übermitteln. Unmittelbar daran anschließend wird dann in den protected mode geschaltet. Dort angekommen, werden GDT und IDT in punkto Tabellenadresse und/oder Größe in der Regel nicht mehr verändert, weshalb LGDT und LIDT hier keine herausragende Funktion mehr haben.

LLDT
SLDT

Load local descriptor table register, LLDT, und store local descriptor table register, SLDT, erfüllen den gleichen Zweck wie die eben besprochenen Befehle für die globalen Tabellen: Sie laden bzw. speichern die »Adresse« der jeweils aktuellen lokalen Deskriptoren-Tabelle in das oder aus dem dafür vorgesehenen Register, dem local descriptor table register (LDTR). Damit sind sie nur im Rahmen der Verwaltungsaufgaben des Betriebssystems interessant und unterliegen entsprechenden Restriktionen.

Im Unterschied zu den vier anderen Befehlen wird hier jedoch nicht eine »echte« Adresse verwendet. Vielmehr müssen alle lokalen Deskriptoren-Tabellen in der GDT mit ihrem LDT-Segment-Deskriptor verzeichnet sein, aus dem die Basisadresse und das Limit der Tabelle entnommen werden können. Daher benötigen LLDT und SLDT als Operanden lediglich einen 16-Bit-Selektor, der den Eintrag in der GDT spezifiziert.



LLDT, nicht aber SLDT, ist ein privilegierter Befehl, was bedeutet, dass er nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden kann. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt unweigerlich zu einer general protection exception #GP.

Operanden

Als Argument erwarten die beiden Befehle einen 16-Bit-Operanden mit einem Selektor (Zeiger) in die GDT:

- Übergabe des Selektors via Register
  `LLDT Reg16; SLDT Reg16`
- Übergabe des Selektors via Speicheroperand
  `LLDT Mem16; SLDT Mem16`

LLDT trägt diesen Selektor in das LDTR ein. Da er auf einen Deskriptor in der GDT zeigt, kann der Prozessor diesem die 32-Bit-Basisadresse sowie das 16-Bit-Limit der LDT entnehmen und zusammen mit den ebenfalls entnehmbaren Attributen des Segmentes im nicht zugänglichen 64-Bit-Cache des LDTR puffern.



Die Übergabe eines Null-Selektors ist nicht verboten und führt zunächst zu keinem Fehler. Vielmehr wird in diesem Fall sowie dann, wenn der Selektor nicht auf einen LDT-Segment-Deskriptor zeigt, der Inhalt des LDTR als »ungültig« markiert. Im Anschluss erzeugt dann jeder nachfolgende Zugriff auf Deskriptoren der durch den ungültigen LDTR-Eintrag referenzierten LDT zu einer general protection exception (#GP). Grund: Der »Null-Selektor« im LDTR zeigt auf den ersten Deskriptor der GDT. Dieser aber gilt als reserviert und enthält keinen Verweis auf ein real existierendes Segment, sodass ihm auch keine Daten entnommen werden können.



LLDT überschreibt das LDT-Feld im task state segments des aktuellen Tasks nicht und hat auch keinen Einfluss auf die Inhalte der Segment-Register. Das bedeutet, dass jeder Zugriff auf Daten und/oder Code (bei dem die Segmentregister involviert sind) unabhängig von Veränderungen im LDTR abläuft. Und jeder nach einem task switch auftretende Switch zurück um aktuellen Task restauriert wieder anhand des nicht veränderten LDT-Feldes die ursprüngliche, task-spezifische LDT.

Statusflags

Statusflags werden durch die Befehle nicht verändert.

LTR
STR

Analog LLDT und SLDT stehen mit LTR und STR zwei Befehle zur Verfügung, die das Task-Register (TR) beschicken oder auslesen. Dieses hat einen zum LDTR analogen Aufbau, weshalb die Aktionen auch analog ablaufen: LTR und STR erwarten als Operanden einen 16-Bit-Selektor, der auf einen task state segment descriptor (TSS-Deskriptor) in der GDT zeigt. LTR trägt diesen Selektor in das TR ein, woraufhin der Prozessor aus dem Deskriptor in der GDT die Adresse, die Größe und die Attribute des task state segments ausliest und im unzugänglichen 64-Bit-Cache puffert.



LTR, nicht aber STR, ist ein privilegierter Befehl, was bedeutet, dass er nur unter höchster Privilegstufe (CPL = 0) ausgeführt werden kann. Der Aufruf aus Anwendungsprogrammen (CPL > 0) oder aus dem real

oder virtual 8086 mode führt unweigerlich zu einer general protection exception #GP.

Operanden Als Argument erwarten LTR und STR einen 16-Bit-Operanden mit einem Selektor (Zeiger) in die GDT:

- Übergabe des Selektors via Register
  `LTR Reg16; STR Reg16`
- Übergabe des Selektors via Speicheroperand:
  `LTR Mem16; STR Mem16`

Statusflags Statusflags werden durch die Befehle nicht verändert.



LTR lädt zwar das task register und markiert auch den korrepondierenden task als busy, indem es das busy flag im TSS-Deskriptor setzt. Dennoch führt LTR keinen task switch durch! Dies hat noch im Rahmen des task switching zu erfolgen. Einzelheiten hierzu entnehmen Sie bitte weiterführender Literatur.



LTR erfordert Privilegstufe 0! Daher ist es äußerst unwahrscheinlich, dass Sie LTR in irgendeiner Weise werden einsetzen können. LTR wird üblicherweise vom Betriebssystem dazu verwendet, den ersten, den »Initialtask«, zu erzeugen. Alle anderen Tasks werden dann im Rahmen des task switching aufgerufen.

System-Ressourcen Neben den bereits mehrfach besprochenen Basisregistern des Prozessors (Allzweckregister, Segmentregister, Flagregister) gibt es noch verschiedene Systemregister: die Kontroll-, Debug- und modellspezifischen Register. Sie zu verwalten ist die Aufgabe der folgenden Befehle.

MOV Im Rahmen der Erweiterung der Prozessoren ab dem 80386 wurden neue Systemregister implementiert, deren Aufgabe einerseits die Unterstützung und Verwaltung von Systemressourcen sind, andererseits dem Anwender bei der Fehlersuche (»Debuggen«) helfen sollen. Dementsprechend gibt es eine Reihe von »Kontrollregistern« und »Debugregistern«, mit denen Datenaustausch möglich sein muss.

Da sich für den Programmierer *formal* kein Unterschied ausmachen lässt, wenn ein Datum zwischen zwei Allzweck- oder je einem Allzweck- und Systemregister ausgetauscht werden, wurden die Opcodes der Instruktionen, die diese Systemregister ansprechen, in die Mnemonik-Familie der MOV-Befehle aufgenommen. Der »MOV-Befehl« wur-

de somit um die Fähigkeit der Kommunikation mit den neuen Systemregistern »erweitert«.



Die Versionen des MOV-Befehls, die mit den Kontroll- oder Debug-Registern der CPU kommunizieren, sind privilegierte Befehle, was bedeutet, dass sie nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden können. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt unweigerlich zu einer general protection exception #GP.

Operanden

Diese Spezialfälle der MOV-Familie können Daten nur zwischen einem Allzweckregister und einem Debug- oder Kontrollregister austauschen:

- Auslesen einen Kontroll- bzw. Debug-Registers
  `MOV Reg32, CReg; MOV Reg32, DReg`
- Beschreiben eines Kontroll- bzw. Debug-Registers
  `MOV CReg, Reg32; MOV DReg, Reg32`

Statusflags

Alle Statusflags sind undefiniert.

RDMSR
WRMSR

Read model specific registers, RDMSR, und *write model specific registers*, WRMSR, sind ein Befehlspaar, mit dem auf die modellspezifischen Register zugegriffen werden kann. Modellspezifische Register sind, wie der Name bereits sagt, spezifisch für jedes Prozessormodell und können von Prozessortyp zu Prozessortyp erheblich variieren, wenn sie überhaupt implementiert sind. Ihre Aufgabe ist, Hardwareunterstützung bei der Entwicklung von Software zu geben (Austesten der Funktionalität, Verfolgung der Befehlsausführung, Performance-Untersuchungen und Bestimmung von Machine-Check-Fehlern).



RDMSR, nicht aber WRMSR, ist ein privilegierter Befehl, was bedeutet, dass er nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden kann. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode oder bei nicht vorhandenen MSRs führt unweigerlich zu einer general protection exception #GP.



Nicht alle Prozessoren verfügen über modellspezifische Register (MSRs). Ob diese überhaupt unterstützt werden, kann mit Hilfe des CPUID-Befehls festgestellt werden. In den feature flags, die dieser Befehl zurückgibt, wenn ihm in EAX der Wert $0000_0001 übergeben wird, signalisiert Bit 5, das Flag MSR, ob MSRs und damit auch die Befehle RDMSR und WRMSR überhaupt implementiert sind.

Operanden RDMSR und WRMSR haben, wie der Befehl CPUID auch, Operanden, die nicht im Opcode codiert werden. Vielmehr muss in ECX die Nummer des gewünschten MSR übergeben werden. RDMSR liefert dann in EDX:EAX die 64 Bits Inhalt des spezifizierten MSR zurück, während WRMSR die in EDX:EAX enthaltenen 64 Bits Information in das spezifizierte MSR zurückschreibt. In beiden Fällen befinden sich die »oberen« 32 Bits des QuadWords in EDX, die »unteren« in EAX.

Statusflags Beide Befehle verändern keine Statusflags.

RDPMC RDPMC, read performance counter, liest die performance monitoring counter aus. Diese Counter wurden mit dem Pentium Pro eingeführt und sind heute bei allen Prozessoren mit MMX-Technologie (also auch MMX-Pentiums) verfügbar. Der Pentium 4 besitzt 18 solcher Counter, die P6-Familie 2. MMX-Pentiums haben auch performance counter, jedoch muss dort ein Zugriff über RDMSR erfolgen. Für Details zu den performance countern verweise ich auf weiterführende Literatur.



RDPMC ist ein bedingt privilegierter Befehl, was bedeutet, dass er eventuell nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden kann. Bedingung für diese Einschränkung ist, dass das Flag PE im CR4 gelöscht ist. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt dann unweigerlich zu einer general protection exception #GP. Ist PE dagegen gesetzt, kann RDPMC auch von Anwendungsprogrammen verwendet werden. Leider gibt es keine einfache Methode, festzustellen, ob RDPMC nun privilegiert ist oder nicht.

Operanden RDPMC hat, wie der Befehl CPUID auch, Operanden, die nicht im Opcode codiert werden. Vielmehr muss in ECX die Nummer des gewünschten Counters übergeben werden. RDPMC liefert dann in EDX:EAX den Inhalt des spezifizierten Counters zurück. Auf die Besprechung von Einzelheiten wird hier verzichtet.

Statusflags Statusflags werden nicht verändert.

RDTSC RDTSC, read time stamp counter, liest den Inhalt des time stamp counter registers aus, einem der modellspezifischen Register des Prozessors. Der Befehl gibt in EDX:EAX den Inhalt dieses 64-Bit-Registers zurück, wobei in EDX das »obere« DoubleWord des QuadWords steht, in EAX das »untere«.

Der time stamp counter wird nach jedem Reset des Prozessors auf Null gesetzt und mit jedem Taktzyklus inkrementiert. Das bedeutet, dass z.B. bei einer Taktfrequenz von 1.6 GHz pro Sekunde 1.6 Milliarden Zyklen gezählt werden. Da der Counter 64 Bits umfasst, können nach einem Reset $2^{64} = 1.845 \cdot 10^{19}$ Zyklen gezählt werden. Dies entspricht bei der genannten Taktfrequenz einer Laufzeit von $1.2 \cdot 10^{10}$ Sekunden oder ca. 365 Jahren – auch wenn die Prozessoren noch schneller werden, sollte man annehmen, dass der Prozessor auch einmal zwischendurch ausgeschaltet wird.



RDTSC ist ein bedingt privilegierter Befehl, was bedeutet, dass er eventuell nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden kann. Bedingung für diese Einschränkung ist, dass das Flag TSD im CR4 gesetzt ist. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt dann unweigerlich zu einer general protection exception #GP. Ist TSD dagegen gelöscht, kann RDTSC auch von Anwendungsprogrammen verwendet werden. Leider gibt es keine einfache Methode, festzustellen, ob RDPMC nun privilegiert ist oder nicht.

Operanden

RDTSC benötigt keine Operanden.

Statusflags

Statusflags werden nicht verändert.

Zugriffsrechte

Es gibt fünf Befehle, die Sie einsetzen können, um zu prüfen, ob Sie ausreichende Zugriffsrechte besitzen, wenn Sie auf ein bestimmtes Segment zugreifen wollen. Auf diese Weise können Sie die Auslösung von Exceptions verhindern, da alle diese Befehle die Zugriffsrechte prüfen, jedoch bei nicht ausreichenden Privilegien ein Statusflag bzw. das RPL-Feld verändern, ohne eine Exception auszulösen. Bis auf ARPL führen alle vier restlichen Befehle folgende grundsätzliche Prüfungen durch:

- Prüfung, ob der übergebene Selektor ein Null-Selektor ist.
- Prüfung, dass der übergebene Selektor auf einen Deskriptor innerhalb der Grenzen der entsprechenden Tabelle (GDT oder LDT) zeigt.
- Wenn es sich nicht um ein »conforming segment« handelt, wird ferner geprüft, ob CPL und RPL kleiner oder gleich dem DPL des Segment-Deskriptors sind, das Segment also im aktuellen Prozess überhaupt sichtbar ist. (Einzelheiten hierzu entnehmen Sie bitte dem Kapitel »Schutzmechanismen« auf Seite 467).

Bei Befehlen, die lediglich die Rechtmäßigkeit des Zugriffs verifizieren sollen und bestimmte Informationen zurückgeben (ARPL, LAR, LSL), erfolgt noch folgende Prüfung:

- Referenziert der übergebene Selektor ein Code-, Daten- oder Systemsegment (TSS oder LDT)? Gates sind nicht erlaubt!

Soll auf Segmente schreibend oder lesend zugegriffen werden, kann vorher der Erfolg geprüft werden, indem die hierzu implementierten Befehle VERR und VERW verwendet werden. Diese prüfen zusätzlich, ob:

- der durch den übergebenen Selektor referenzierte Deskriptor ein Code- oder Datensegment beschreibt – Systemsegmente (TSS und LDT) oder Gates sind nicht erlaubt.
- das Segment entweder als lesbar markiert (VERR) oder ein beschreibbares Datensegment (VERW) ist.



Zum besseren Verständnis der folgenden Befehle sollten Sie etwas genauer über die Schutzkonzepte und Zugriffsrechte samt ihrer Prüfung informiert sein. Falls dies nicht so ist, sollten Sie vorher das Kapitel »Schutzmechanismen« auf Seite 467 lesen.

Die fünf Befehle im Einzelnen:

ARPL ARPL, *adjust requestor privileg level*, reduziert das RPL-Feld eines übergebenen Selektors auf das Niveau eines in einem zweiten Selektor übergebenen RPL.

Der eigentliche Einsatzort von ARPL sind Routinen des Betriebssystems, die Anfragen von Anwenderroutinen auf ihre Rechtmäßigkeit überprüfen müssen. Somit macht die Verwendung von ARPL in Anwenderroutinen wenig Sinn, wenngleich sie auch nicht verboten ist, ARPL also nicht zu den privilegierten Befehlen gehört! Um die Funktionsweise und den Sinn von ARPL besser zu verstehen, ein kleines Gedankenexperiment. Stellen Sie sich vor, Sie wollten unbefugterweise auf Daten im Datensegment des Betriebssystems zugreifen. So brauchten Sie nur einen Selektor, der auf den entsprechenden Deskriptor zeigt. Dessen DPL-Feld enthält die Privilegstufe, die Sie benötigen, um das zu erreichen. Beim Zugriff auf das Datensegment müssen Sie diesen Selektor in ein Segmentregister schreiben. Und weil Sie wissen, dass bei Betriebssystemdaten das DPL den Wert 0 hat, setzen Sie vor dem Schreiben in das Segmentregister dessen RPL auch auf Null – was Ihnen

niemand verbietet. Leider können Sie dennoch nicht auf das Datensegment zugreifen, da die Zugriffsprüfungen nicht nur den RPL mit dem DPL vergleichen, sondern auch CPL. Und der ist im Selektor im CS-Register verzeichnet und kann nicht einfach auf Null gesetzt werden. Daher wird Ihr Zugriffsversuch aufgrund des CPL = 3 von Anwendungsprogrammen abgewiesen.

Nun sind Sie ja nicht dumm und denken sich: ›Dann greifen wir eben über eine Kernel-Routine zu. Die hat einen CPL = 0. Dann müsste es eigentlich funktionieren.‹ Und in der Tat: Dieser Weg hätte Aussicht auf Erfolg ...

... gäbe es nicht ARPL. Denn die angesprungene Betriebssystemroutine setzt mittels ARPL die aktuellen Privilegien auf die Privilegstufe des rufenden Programms zurück! Hierzu prüft es das RPL-Feld im als ersten Operanden übergebenen Selektor mit dem RPL-Feld des im zweiten Operanden übergebenen. Ist dieses kleiner, wird es auf den Wert des RPL-Feldes im zweiten Operanden gesetzt. Andernfalls unterbleibt eine Anpassung. Der erste Selektor wurde somit hinsichtlich der Privilegien an den zweiten »angepasst« und dadurch ggf. um zu weit reichende Privilegien beschnitten.

Wie wirkt sich das in unserem Gedankenexperiment aus? Zum besseren Verständnis denken wir uns in besagte Kernel-Routine hinein, nachdem sie von einem Anwenderprogramm angesprungen wurde. Der CPL ist 0, da wir in einer Kernel-Routine sitzen. Auf dem Stack liegt die Rücksprungadresse, also die auf den CALL-Befehl folgende Adresse, die uns hierher geführt hat. Da es ein Far-Call gewesen sein muss (Intersegment-Call!), der uns hierher gebracht hat, ist Teil dieser Adresse ein Selektor: eine Kopie des CS-Inhaltes der rufenden Anwenderroutine. Dieser Selektor enthält aber als RPL den CPL der rufenden Routine. Und der ist, als Anwendungsprogramm, 3.

Ohne ARPL könnte die Kernel-Routine selbstverständlich auf Betriebssystemdaten zugreifen, da sie selbst ein CPL von 0 hat und damit auf alle Datensegmente zugreifen kann – schließlich war das ja das Konzept, das uns hierher führte. Der Aufruf von ARPL aber bewirkt, dass der Zugriff auf das Datensegment verweigert wird, indem die Betriebssystemroutine ARPL als ersten Operanden den Selektor auf das Datensegment übergibt und als zweiten den Selektor der Rücksprungadresse. Da der RPL auf das Datensegment 0 ist und der RPL des Rücksprung-Selektors 3, hat auch das RPL-Feld des ersten Operanden nach ARPL den Wert 3. Ein Zugriff auf das geschützte Datensegment ist somit auch

über diesen Trick nicht möglich! Wäre die gleiche Routine aus dem Kernel angesprungen worden, hätte die Rücksprungadresse einen RPL von 0 und der Zugriff wäre erlaubt, da ARPL nichts anzupassen hätte.

Dieses Beispiel macht aber auch deutlich, dass ARPL wirklich nur in Betriebssystemroutinen Sinn macht. Innerhalb von Anwendungsprogrammen mit CPL = 3 kann praktisch lediglich vorab geprüft werden, inwieweit der Versuch des Zugriffs auf ein Segment über einen konkreten Selektor Probleme machen könnte.

Operanden Als Argument erwartet ARPL zwei 16-Bit-Operanden. Der Ziel- und erste Quelloperand kann dabei entweder ein 16-Bit-Register oder eine 16-Bit-Speicherstelle sein, der zweite Operand ist immer ein 16-Bit-Register:

- Übergabe beider Selektoren via Register
  `ARPL Reg16, Reg16`
- Übergabe des Ziel- und ersten Quell-Selektoren via Speicheroperand
  `ARPL Mem16, Reg16`

Statusflags Wurde der RPL des ersten Operanden an den zweiten angepasst (d.h. RPL #1 < RPL #2!), wird das zero flag gesetzt, andernfalls gelöscht.



Sowohl den Betriebssystemroutinen, die ARPL verwenden, als auch Ihnen steht frei, das zero flag auszuwerten. Ist es gesetzt, heißt es, dass eine Anpassung stattgefunden hat, der Rufer also geringere Privilegien hat als gefordert. Durch Testen des zero flags können Sie (und die Routine) somit verhindern, dass beim eigentlichen Zugriff eine Exception erzeugt wird.

LAR
LSL Mit LAR, load access rights, können Sie die Zugriffsrechte aus dem Deskriptor auslesen, auf den ein von Ihnen übergebener Selektor zeigt. Dies umfasst das DPL-Feld, das Typ-Feld sowie die Flags S, P, AVL, D/B und G. Beträgt die aktuelle Operandengröße nur 16 Bit (festgelegt durch das big flag des aktuellen Datensegments und ggf. eines vor LAR stehenden operand size override prefix), so werden aus dem Deskriptor lediglich die Felder DPL und Type extrahiert.

Die in die Operanden geschriebenen Werte werden nach dem Auslesen des Deskriptors vor dem Eintrag in den Operanden mit $00FxFF00 (32-Bit) bzw. $FF00 (16 Bit) maskiert, sodass tatsächlich nur Bits gesetzt sein können, die die entsprechenden Informationen beherbergen.

LSL, load segment limit, entnimmt ebenfalls Daten aus dem Deskriptor, auf den der übergebene Selektor zeigt. Doch hier handelt es sich um das Segmentlimit. LSL eignet sich daher hervorragend dazu, Offsets in ein Segment vor ihrer Nutzung gegen die Segmentgröße zu testen, ohne eine Exception auszulösen, wenn die Limits überschritten werden.

LSL hat gegenüber dem Auslesen »von Hand« zwei gewichtige Vorteile: Es setzt im Falle von 32-Bit-Operanden erstens das über das Double-Word zerstückelte Limit zu einem echten 32-Bit-Wert zusammen und berücksichtigt zweitens sogar das granularity bit G. Ist es gesetzt, so wird der im Deskriptor verzeichnete Wert für das Limit mit 2^{12} skaliert und die »unteren« 12 Bits auf 1 gesetzt.



Auch im Falle der Verwendung von 16-Bit-Operanden wird ein korrekter 32-Bit-Wert für das Limit berechnet. In diesem Fall wird jedoch nur das »untere« Word, also die niedrigerwertigen 16 Bits dieses Limits, in den Operanden kopiert.

Operanden

Beide Befehle erwarten einen Zieloperanden als ersten Operanden, in den die extrahierten Daten eingetragen werden, sowie einen Quelloperanden (zweiter Operand), über den der Selektor auf den auszulesenden Deskriptor übergeben wird. Zieloperand ist immer ein Register, während als Quelle ein Register oder eine Speicherstelle möglich sind (XXX steht für LAR oder LSL):

- Übergabe des Selektors via Register
  `XXX Reg16, Reg16; XXX Reg32, Reg32`
- Übergabe des Selektors via Speicheroperand
  `XXX Reg16, Mem16; XXX Reg32, Mem32`

Statusflags

Führt die zu Beginn des Abschnittes angesprochene Prüfung zu einer Zugriffsverletzung, so können keine Daten aus dem Deskriptor extrahiert werden. In diesem Fall wird das zero flag gelöscht und der Inhalt des Zielregisters gilt als undefiniert. Andernfalls wird das zero flag gesetzt und das Zielregister enthält die gewünschten Daten.

VERR
VERW

VERR, verify a segment for reading, und VERW, verify a segment for writing, prüfen, ob das über den übergebenen Selektor referenzierte Segment ausgelesen bzw. beschrieben werden kann. Hierzu verwenden beide Befehle Informationen, genauer das type field und das system flag, aus dem Deskriptor, auf den der Selektor zeigt.

Operanden Beide Befehle erwarten nur einen Operanden, einen Quelloperanden, über den der Selektor übergeben wird. In Frage kommen ein 16-Bit-Register oder eine 16-Bit-Speicherstelle (XXX steht für VERR bzw. VERW):

- Übergabe des Selektors via Register
  `XXX Reg16`
- Übergabe des Selektors via Speicheroperand
  `XXX Mem16`

Statusflags Führt die zu Beginn des Abschnittes angesprochene Prüfung zu einer Zugriffsverletzung, so können keine Daten aus dem Deskriptor extrahiert werden. In diesem Fall wird das zero flag gelöscht und der Inhalt des Zielregisters gilt als undefiniert. Andernfalls wird das zero flag gesetzt, falls die Typ-Überprüfung für VERR ein Codesegment (Bit 11 des zweiten DoubleWords = 1 und system flag = 1) ergeben, dessen read enable flag gesetzt ist, oder ein Datensegment (Bitt 11 = 0 und S = 1), das immer lesbar ist. Das zero flag wird im Falle von VERW nur dann gesetzt, wenn ein Datensegment vorliegt (S = 1 und Bit 11 = 0), dessen write enable flag gesetzt ist. In allen anderen Fällen ist das zero flag gelöscht.

System-Befehle In diesem Abschnitt werden Befehle beschrieben, die im Rahmen sehr spezifischer Aufgaben eine Rolle spielen: Cache-Verwaltung, Task-Verwaltung und Prozessorzustand.

CLTS Bei jedem task switch setzt der Prozessor das TS flag in Kontrollregister 0 (CR0), um dem Betriebssystem den erfolgten task switch zu signalisieren. Dieses kann dann die FPU- und/oder MMX-Umgebung sichern und die für den neuen Task gültige laden. Anschließend sollte das TS flag gelöscht werden. CLTS, *clear task switched flag*, übernimmt diese Aufgabe.



CLTS ist ein privilegierter Befehl, was bedeutet, dass er nur unter höchster Privilegstufe (CPL = 0) oder im real mode zur Initialisierung des protected mode ausgeführt werden kann. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt unweigerlich zu einer general protection exception #GP.

Operanden Der Befehl besitzt keine Operanden.

Statusflags Statusflags werden nicht verändert, CLTS ändert lediglich den Zustand des TS flags in CR0.

HLT

HLT, *halt*, hält den Prozessor an, indem die weitere Ausführung von Instruktionen eingestellt und der Prozessor in den »Halt-Zustand« versetzt wird. Aus diesem Zustand kann er nur noch durch nicht maskierbare Interrupts (NMI und SMI), durch nicht maskierte, maskierbare Interrupts (INTR), eine Debug-Exception oder Signale an den Interrupt- oder Reset-Pins geholt werden.



HLT ist ein privilegierter Befehl, was bedeutet, dass er nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden kann. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt unweigerlich zu einer general protection exception #GP.

Operanden

Der Befehl besitzt keine Operanden.

Statusflags

HLT verändert keine Statusflags.

INVD
WBINVD

INVD, *invalidate internal caches*, leert die internen Caches der CPU, ohne deren Inhalt in den Speicher zurückzuschreiben. WBINVD, *write-back and invalidate internal caches*, ist eine modifizierte Form von INVD, bei der vor dem Leeren der internen Caches deren Inhalt in den Speicher zurückgeschrieben wird, falls sie modifizierte Daten enthalten. Beide Instruktionen legen dann ein Signal auf den Datenbus, den externe Caches nutzen können, ebenfalls ihre Inhalte zurückzuschreiben und/oder sich zu entleeren.



Die Benutzung von INVD birgt eine gewisse Gefahr. Da dieser Befehl den Cache-Inhalt nicht auf Modifikationen überprüft und im Falle geänderter Daten diese in den Hauptspeicher zurückschreibt, gehen die Daten unweigerlich verloren! Daher sollte im Zweifel WBINVD verwendet werden.



INVD und WBINVD sind privilegierte Befehle, was bedeutet, dass sie nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden können. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt unweigerlich zu einer general protection exception #GP.

Operanden

INVD und WBINVD besitzen keine Operanden.

Statusflags

Statusflags werden nicht verändert.

INVLPG *Invalidate page*, INVLPG, löscht einen Eintrag im TLB (»translation lookaside buffer«). Dem Befehl wird als Operand die Adresse einer Speicherstelle übergeben. Der Prozessor bestimmt daraufhin die page, in der diese Adresse physikalisch gehalten wird, und löscht den dazugehörigen Eintrag im TLB.



INVLPG ist ein privilegierter Befehl, was bedeutet, dass er nur unter höchster Privilegstufe (CPL = 0) oder im real mode ausgeführt werden kann. Der Aufruf aus Anwendungsprogrammen (CPL > 0) im protected mode führt unweigerlich zu einer general protection exception #GP.

Operanden Dem Befehl kann nur eine Speicheradresse übergeben werden:

```
INVLPG Mem
```

Statusflags Statusflags werden nicht verändert.

RSM Resume from system management mode, RSM, gibt die Kontrolle wieder an das Programm zurück, das durch einen SMI (system management interrupt) unterbrochen wurde. Die gesamte Prozessorumgebung, die durch den SMI gesichert wurde, wird mittels RSM wieder hergestellt.

Operanden Der Befehl besitzt keine Operanden.

Statusflags Alle Statusflags werden verändert.

1.1.16 Obsolete Befehle

Die Evolution der Intel-Prozessoren führte dazu, dass neue Befehle geschaffen wurden (werden mussten), die die neu geschaffenen Möglichkeiten und/oder Fähigkeiten des »neuen« Prozessors unterstützten. Manchmal sind diese neuen Fähigkeiten/Möglichkeiten im Verlauf der weiteren Evolution in anderen oder nochmals erweiterten Fähigkeiten/Möglichkeiten aufgegangen, die ihrerseits neue Befehle erforderten. Oft wurden die Befehle dazu lediglich »erweitert« (vgl. MOV). In manchem Fällen haben jedoch andere Befehle die Funktionalität übernommen, womit die ehemals neu geschaffenen überflüssig wurden. Aus Gründen der konsequenten Abwärtskompatibilität der Intel-Prozessoren wurden solche »obsoleten« Befehle aber niemals wieder abgeschafft. Es gibt sie noch heute und wird sie vermutlich auch in 10 Prozessor-Generationen noch geben. Ihre Verwendung sollte jedoch nur auf die Fälle beschränkt werden, in denen Abwärtskompatibilität wirklich notwendig ist. Dies ist sehr selten der Fall: So geht heute kein Betriebssystem

mehr davon aus, z.B. auf einem 80286 laufen zu müssen. Daher macht es keinen Sinn, in diesem Fall obsolete 80286-Befehle zu nutzen.

LMSW
SMSW

LMSW/SMSW sind zwei solcher obsoleten Befehle. Eine wesentliche Neuerung des 80286 war die Einführung des *protected mode*. Dieser neue Betriebsmodus machte es erforderlich, neue Register zu kreieren, die eine Verwaltung dieses Modus ermöglichten. Eines dieser neuen Register war das *machine status register*, ein 16-Bit-Register, das das *machine status word* aufnahm. Dieses Register zu beschreiben und auszulesen war Aufgabe der Befehle *load machine status word* (LMSW) und *store machine status word* (SMSW).

Mit Einführung des 80386 wurden weitere tief greifende Erweiterungen vorgenommen, wie z.B. die Erweiterung des Adressbus auf 32 Bit. Das *machine status word* mit seinen 16 Bit reichte nun nicht mehr aus und wurde ersetzt durch das 32 Bit breite *control register* CR0. Das *machine status word* bildete fortan das »untere« Word des DoubleWords in CR0. Da sich diesem Kontrollregister noch weitere Kontroll- und Debug-Register hinzugesellten, wurde zwecks Datenaustausch der bewährte MOV-Befehl dahingehend erweitert, dass er nun auch diese Systemregister ansprechen konnte. LMSW und SMSW wurden damit überflüssig und sind heute nur noch aus Gründen der Abwärtskompatibilität zum 80286 implementiert.



Bitte beachten Sie, dass die Formulierung »der MOV-Befehl wurde erweitert« missverständlich sein könnte! MOV ist, wie alle anderen »Befehle« in diesem Kapitel, ein Mnemonic, also eine Art menschenfreundliches Etikett für die eigentliche, maschinenverständliche Instruktion. Diese besteht u.a. aus dem Opcode als wichtigstem Element der Instruktion. Und diese Opcodes unterscheiden sich im Falle der MOV-Befehle erheblich. So haben selbstverständlich die Opcodes, die mit den Kontroll- oder Debug-Registern kommunizieren, andere Werte als die, die mit den anderen Registern zusammenarbeiten. Insofern hätte man durchaus andere, neue Mnemonics für die Erweiterungen wählen und in die Assembler integrieren können. Da aber Mnemonics Hilfen für den Assemblerprogrammierer sein sollen, nicht Hunderte von kryptischen Zahlenfolgen auswendig lernen zu müssen, sondern vielmehr »Sinn« und Ordnung in die Opcodes zu bekommen, wurde ein »globaler« Befehl geschaffen, der sich mit Datenaustausch zwischen Registern und Speicher beschäftigt: MOV.

1.1.17 Privilegierte Befehle

Die modernen Betriebssysteme von heute realisieren einen mehr oder weniger ausgeprägten Schutz davor, dass sich Programme im Rahmen des Multitasking-Konzeptes gegenseitig beeinflussen können. Die hierzu eingesetzten Schutzkonzepte bedienen sich Mechanismen, die sicherstellen, dass bestimmte Aktionen nur auf der Ebene des Betriebssystems, nicht aber auf Anwendungsebene erfolgen können. Daher sind einige der eben vorgestellten Systembefehle nur auf der Ebene des Betriebssystems ausführbar. Der Aufruf in anderen Programmen führt zur Auslösung von Exceptions.

Diese Befehle werden als privilegiert bezeichnet, da zu ihrer Ausführung eine bestimmte Privilegstufe erforderlich ist. Einzelheiten hierzu entnehmen Sie bitte den »Schutzmechanismen« auf Seite 467. Man unterscheidet zwei Arten privilegierter Befehle: die absolut privilegierten Befehle, die keinerlei Ausnahmen von diesem Konzept zulassen, und die bedingt privilegierten Befehle, bei denen unter bestimmten Umständen eine Ausführung auch mit niedrigerer Privilegstufe möglich ist.

absolut privilegiert

Die absolut privilegierten Befehle sind alle Systembefehle, die einen direkten Einfluss auf die gewählten Schutzkonzepte und/oder Abläufe im Betriebssystem haben. Hierzu zählen die Befehle, mit denen die Deskriptor-Tabellen geladen (LGDT, LLDT, LIDT) oder Tasks verwaltet werden können (LTR, CLTS). Auch die Erweiterungen des MOV-Befehls, die als Operanden die Debug- oder Kontrollregister des Prozessors akzeptieren, gehören zu den privilegierten Instruktionen, wie deren »Schmalspurversion« LMSW oder die Befehle, die auf die modellspezifischen Register zugreifen können (RDMSR, WRMSR). Natürlich sind auch Befehle, die mit dem Paging-Mechanismus zusammenhängen, für alle Programme außer den Betriebssystemroutinen tabu (INVD, WBINVD, INVDPG). Schließlich setzt auch das Anhalten des Prozessors (HLT) eine entsprechende Privilegstufe voraus.

bedingt privilegiert

Bedingt privilegiert dagegen sind Befehle, die nicht direkt in Betriebssystemangelegenheiten eingreifen, jedoch entsprechende Ressourcen nutzen. Dies sind Befehle, die im Rahmen des *performance monitoring* eine Rolle spielen (RDPMC, RDTSC). Bedingt privilegiert heißt in diesem Fall, dass zwei Flags darüber entscheiden, ob sie privilegiert sind oder nicht. Diese beiden Flags (PCE, *performance-monitoring counter en-*

able, und TSD, *time stamp disable*, bzw. Bit 8 und Bit 2 des *control registers #4*) erlauben einen Zugriff auf den *perfomance counter* (PCE = 1) mittels RDPMC bzw. den *time stamp counter* (TSD = 0) mittels RDTSC bzw. unterbinden ihn. Haken an der Angelegenheit: Das CR4 ist nur über einen privilegierten MOV-Befehl zugänglich. Das heißt: Entweder das Betriebssystem erlaubt Ihnen von vornherein die Nutzung dieser Ressourcen oder eben nicht. *Sie* können das nicht ändern!

1.1.18 CPU-Exceptions

Bei der Bearbeitung der eben vorgestellten CPU-Befehle geht es nicht immer reibungslos zu! Sei es, dass bei der Adressierung einer Speicherstelle eine Adresse gewählt wurde, die außerhalb des »erlaubten« Bereiches liegt oder in einem Segment, das zurzeit nicht verfügbar ist, sei es, dass eine Zahl durch »0« dividiert wurde, die FPU meckert oder ein »Gerät« einen Fehler signalisiert. Alles das sind Situationen, in denen die CPU zunächst nicht weiß, wie sie reagieren soll. Denn immerhin muss sie nun etwas tun, was nicht im regulären, derzeit bearbeiteten Programmcode vorgesehen ist.

Solche Situationen nennt man Ausnahmezustände oder »*exceptions*«. Exceptions sind somit »Unterbrechungen« des regulären Programmablaufs. Um sie zu behandeln, ist es notwendig, zunächst den aktuellen Prozessorzustand zu sichern. Dann gilt es, festzustellen, welchen Grund die Ausnahmesituation hat, und entsprechend zu reagieren. Zuständig für die Behandlung solcher Unterbrechungen oder »*interrupts*« sind »Interrupthandler«. Dies sind Programmteile, die mehr oder weniger gut auf die Behandlung von Interrupts oder Exceptions spezialisiert sind. Der Prozessor versucht daher, solche *Handler* aufzurufen. Nicht immer ist das möglich. Aber wenn, dann versucht der Handler, den Fehler so gut wie möglich zu beheben, sodass der Prozessor nach Abschluss der Aktivitäten des Handlers mit der Programmausführung an der Stelle weitermachen kann, an der er unterbrochen wurde.

Einzelheiten zum Exception-Mechanismus und zu Interrupts entnehmen Sie dem Kapitel »Exceptions und Interrupts« auf Seite 486.



Exceptiontypen Gemäß Ursache und Schweregrad unterscheidet man drei Exceptiontypen: fault, trap und abort.

fault Der Typ fault ist der häufigste Typ. Salopp formuliert kann man ihn beschreiben mit den Worten: »Uuuups – da *wäre* ich doch fast in etwas hineingetreten!« Das bedeutet, dass der Fehler (*fault*) von der CPU festgestellt wurde, *bevor* die Operation ausgeführt wurde. Der Exception-Handler kann somit sehr einfach und wirkungsvoll die Ursache der Exception aus dem Weg räumen, um einen weiteren reibungslosen Programmablauf zu gewährleisten. Beispiel: Sollte bei einem DIV/IDIV-Befehl durch »0« dividiert werden, so kann der Prozessor dies feststellen, bevor die Operation erfolgt und tatsächlich zu einem Fehler führt. In diesem Fall kann der Prozessor den aktuellen Zustand einfrieren und den Exception-Handler aufrufen, um den Fehler korrigieren zu lassen. Wenn der Handler dann fertig ist, kann die CPU die Arbeit an der Stelle wieder aufnehmen, an der der Fehler auftrat, und den Exception-auslösenden Befehl nochmals ausführen – in der Hoffnung, der Handler hat den Fehler korrigiert. Wenn nicht, gibt's eine Endlosschleife ...

trap Genauso salopp formuliert bedeutet Typ trap: »Hoppla – da *bin* ich doch in etwas hereingetreten!« Soll heißen, der Fehler konnte erst entdeckt werden, *nachdem* die Operation stattgefunden hat. Erst dann kann der Exception-Handler aufgerufen werden. Nach seiner Aktivität wird somit die Programmausführung *nach* dem die Exception auslösenden Befehl fortgesetzt – wenn überhaupt. Solch eine »Falle« (*trap*) stellt beispielsweise die Situation nach einer Operation dar, die das overflow flag gesetzt hat, z.B. ADD. Zu dem Zeitpunkt, an dem das OF geprüft und im Rahmen von INTO eine Exception ausgelöst werden kann, sind bereits Fakten in Form eines Ergebnisses der Addition geschaffen worden, die nicht mehr rückgängig zu machen sind. Der Handler muss in diesem Falle prüfen, was mit dem Ergebnis zu tun ist.

abort Schließlich der Typ abort: »So meine Lieben, das war es! Und tschüss.« Exceptions dieses Typs treten nach schwerwiegenden, nicht reparablen Fehlern auf, z.B. wenn bei einer Hardwareprüfung (z.B. *machine check*) Hardwarefehler festgestellt werden, Systemtabellen fehlerhaft sind oder ein »Fehler im Fehler« auftritt, also eine Fehlerbehandlung durch einen Exception-Handler zu einer Exception führt. Was könnte die CPU, was ein Exception-Handler tun? Gar nichts – und deshalb kehrt der Handler gar nicht erst in das unterbrochene Programm zurück. Resultat: Das Programm wird abgebrochen.

Exceptions

Die CPU kennt (derzeit) 17 Exceptions. Zwar gibt es noch ein paar weitere Exceptions, diese sind jedoch entweder obsolet (*coprocessor-segment overrun abort* beim Gespann 80386/80387) oder werden intern verwendet und gelten daher als reserviert. Die definierten Exceptions sind:

#DE

Divide Error; diese Exception vom Typ *fault* löst die CPU selbst aus, wenn bei einem DIV oder IDIV durch einen Divisor mit dem Wert »0« dividiert werden soll.

#DB

Debug; hierbei handelt es sich um eine Exception vom Typ *fault* oder *trap*, die entweder als Software-Exception gezielt durch den Befehl INT 01 oder als Hardware-Exception durch die CPU selbst nach jedem Befehl oder bei jedem Datenzugriff ausgelöst wird. Mit dieser Exception wird einem Debugger ermöglicht, nach INT 01 oder jedem Befehl den CPU-Zustand (Registerinhalte, Flag-Stellungen etc.) festzustellen und zwecks Debuggen darzustellen.

#BP

Break Point; diese Software-Exception vom Typ *trap* wird durch den Befehl INT 03 ausgelöst und ermöglicht das gezielte Setzen von »Haltepunkten«, also Stellen, an denen das Programm unterbrochen und die Kontrolle einem Debugger übergeben wird. Während #DB dies nach jedem Befehl (oder INT 01) tut, ermöglicht #BP dem Anwender, den Programmablauf an von ihm bestimmbaren Punkten zu unterbrechen.

#OF

Overflow; diese Software-Exception vom Typ *trap* wird durch den Befehl INTO und daher gezielt durch den Anwender ausgelöst, falls das overflow flag gesetzt ist. Ist es nicht gesetzt, unterbleibt die Exception. Auf diese Weise ist es möglich, z.B. nach Vergleichen eine Programmverzweigung nicht mittels bedingter Sprungbefehle, sondern durch Interrupt zu erreichen. Dies ist vor allem dann sinnvoll, wenn nicht anhand des Vergleiches (oder anderer Flag-setzender Befehle) eine »echte« Programmverzweigung erfolgen soll, sondern nach eventueller Korrektur des Fehlers der Programmablauf in jedem Fall gleich fortgesetzt werden soll.

#BR

Bound Range Exceeded; auch hierbei handelt es sich um eine Software-Exception, die vom Befehl BOUND ausgelöst wird, wenn die im ersten Operanden übergebenen Werte die im zweiten Operanden übergebenen Grenzen überschreiten. Ist dies nicht der Fall, unterbleibt die Exception. #BR ist vom Typ *fault*.

#UD *Undefined (Invalid) Opcode*; diese Exception vom Typ *fault* löst die CPU selbst aus, wenn Sie den Befehl UD2 (gezieltes Auslösen der Exception durch den Anwender zum Zwecke des Debuggens) oder einen nicht existierenden (»undefined«) oder reservierten (»invalid«) Opcode ausführen soll.

#NM *Device Not Available (No Math Coprocessor);* diese Exception vom Typ *fault* wird durch die CPU ausgelöst, sobald ein WAIT/FWAIT oder ein FPU-Befehl ausgeführt werden soll und keine FPU bzw. kein NPX (= mathematischer Co-Prozessor) vorgefunden wird.

#DF *Double Fault*; wie beim Tennis kann es auch bei Exceptions zum »Doppelfehler« kommen. Hier wie dort wird dadurch angezeigt, dass ein Fehler innerhalb der Behandlung eines anderen Fehlers aufgetreten ist: Wenn innerhalb eines *exception* oder *interrupt handlers* ein Befehl, der zur Auslösung einer Exception, eines NMIs (»non-maskable external interrupt«) oder eines INTRs (»interrupt request«) befähigt ist, genau diese(n) auslöst. Dies hat in der Regel zur Folge, dass das Programm abgebrochen wird. #DF ist daher eine Exception vom Typ *abort*.

#TS *Invalid TSS*; die CPU löst diese Exception aus, falls bei einem *task switch* oder dem Zugriff auf ein *task state segment* (TSS) festgestellt wird, dass das zu verwendende TSS fehlerhaft ist. #TS ist vom Typ *fault*.

#NP *Segment Not Present*; diese Exception löst die CPU aus, wenn im Rahmen der Speicherverwaltung auf ein Segment zugegriffen werden soll, das sich nicht im physischen Speicher befindet. Diese Exception vom Typ *fault* wird durch alle Befehle ausgelöst, die entweder ein Segmentregister laden oder auf Systemsegmente zugreifen. Einzelheiten zu Segmenten finden Sie im Kapitel »Speicherverwaltung« auf Seite 394.

#SS *Stack Segment Fault*; diese Exception vom Typ *fault* löst die CPU aus, wenn beim Laden des Stack-Segmentregisters (SS) oder beim Zugriff auf das Stacksegment ein Fehler festgestellt wird.

#GP *General Protection*; diese Exception vom Typ *fault* haben die meisten Anwender und Programmierer »lieben« gelernt, da sie so »aussagekräftig« ist. Die CPU löst sie immer dann aus, wenn eine Schutzverletzung auftritt. Dies kann beim Versuch des Zugriffs auf Segmente mit höheren Privileganforderungen sein, als sie der Zugreifer hat, beim Versuch des Zugriffs auf geschützte I/O-Ports oder bei sonstigen Zugriffsverletzungen. Jeder Zugriff auf den Speicher und jede andere Prüfung der Privilegien kann Grund für eine #GP sein.

Page Fault; bei jedem Zugriff auf den Speicher prüft die Speicherverwaltung (im Rahmen des Paging-Mechanismus), ob die gewählte *page* verfügbar ist oder nicht. Ist das nicht der Fall, wird diese Exception vom Typ *fault* ausgelöst. **#PF**

FPU Error (math fault); die CPU löst diese Exception vom Typ *fault* aus, wenn der Befehl WAIT/FWAIT oder jeder andere FPU-Befehl eine unbehandelte FPU-Exception anzeigt. **#MF**

Alignment Check; bei jedem Zugriff auf den Speicher prüft die CPU, ob die Daten entsprechend der Erfordernisse ausgerichtet sind oder nicht. Ist dies nicht der Fall, löst sie diese Exception vom Typ *fault* aus. **#AC**

Machine Check; die CPU löst nach einem *machine check* diese Exception vom Typ *abort* aus, falls sich ein Fehler ergeben sollte. Die genauen Gründe für diese Exception sind abhängig von der CPU und deren Möglichkeiten (»*machine specific*«). **#MC**

SIMD Floating Point Error; die CPU löst diese Exception vom Typ *fault* aus, sobald ein bei einer SSE- oder SSE2-Instruktion aufgetretener Fehler festgestellt wird. Achtung: Es werden lediglich Fehler signalisiert, die analog der FPU-Befehle (#NM) während Fließkomma-Berechnungen auftreten. Somit ist #XF das SIMD-Pendant zu #NM. Bei der Bearbeitung von Integers im Rahmen von MMX können nur die hier behandelten CPU-Exceptions auftreten. **#XF**

1.2 FPU-Operationen

Seit dem 80486 von Intel denkt man an FPU, wenn es um Fließkommazahlen geht: FPU steht für *floating point unit* und bezeichnet den Teil (»*unit*«) des Prozessor-Chips, der für Fließkomma-Berechnungen zuständig ist. Die FPU als integrierter Teil der CPU ist das Ergebnis einer Evolution, die mit dem Prozessorgespann 8086/8087 begann und mit dem 80386/80387 endete. Denn vor dem 80486 war für alle Fließkommabelange ein eigenständiger Chip zuständig, der häufig als »numerischer Co-Prozessor« oder »*numeric processing extension*«, kurz NPX, bezeichnet wurde. Verschiedene Gründe (Performance, parallele Befehlsverarbeitung, etc.) haben es jedoch sinnvoll erscheinen lassen, diesen NPX auf dem CPU-Chip zu realisieren und ihn stärker der Kontrolle der CPU zu unterstellen. Fertig war die FPU. FPU und NPX meinen daher, im Kontext dieses Buches betrachtet, das Gleiche. Es wird daher *FPU oder NPX?*

durchgängig der Begriff FPU verwendet, auch wenn bei der Betrachtung historischer Co-Prozessoren ein realer Chip betroffen sein sollte.

FPU-Datenformate

Es gibt eigentlich nur ein Datenformat, das die FPU kennt und mit dem sie Berechnungen durchführt: den »double-extended precision floating point value«, der in diesem Buch als ExtendedReal bezeichnet wird. Alle anderen Daten wie SingleReals, DoubleReals, BCDs und auch Integers, die als Operanden für FPU-Befehle in Frage kommen, werden in dieses 80-Bit-Datum konvertiert, bevor sie in ein FPU-Register gelangen, oder aus ihm übersetzt, bevor sie in den Speicher zurückgeschrieben werden. Zu Einzelheiten über die Darstellung der verwendeten Datentypen siehe das Kapitel »Elementardaten« auf Seite 788.

FPU-Register

Die FPU verfügt über acht »riesige« Register, in denen die arithmetischen Berechnungen ablaufen. Es sind die 80 Bit breiten Register R0 bis R7. Darüber hinaus verfügt sie über drei 16-Bit-Register, das control register, das status register und das tag register, sowie über zwei 48 Bit breite Pointer-Register. Das 11-Bit-Register Op fasst die signifikanten 11 Bit des Opcodes, auf den LIP zeigt. Die Register sind in Abbildung 1.28 dargestellt:

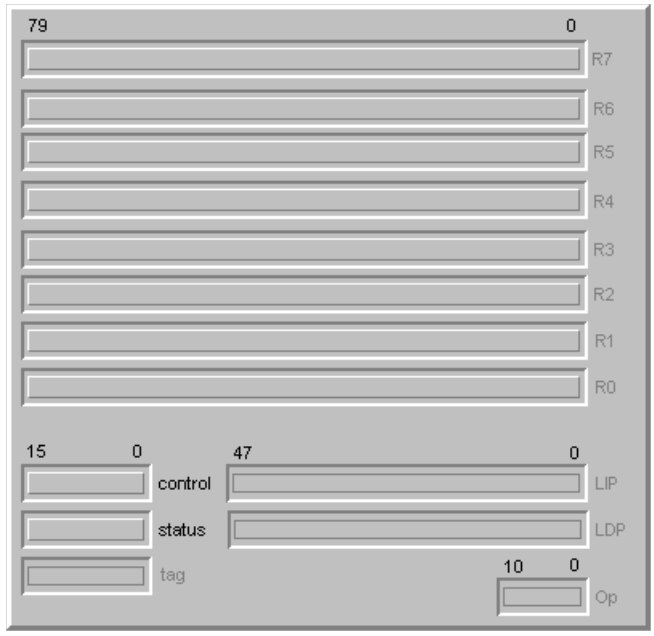


Abbildung 1.28: Die Register der FPU (»floating point unit«) bzw. der NPX (»numeric processing extension«, »arithmetic co-processor«)

Direkt ansprechbar sind hierbei lediglich das status und das control register. Das bedeutet, dass nur diese beiden Register als Operanden bei bestimmten FPU-Befehlen eingesetzt werden können. Das tag register wird von der FPU verwaltet und ist, wie LIP, LDP und Op, dem Zugriff des Programmierers entzogen. Und auch das status register kann nur ausgelesen werden. Einen Befehl, der in das status register schreiben kann, gibt es nicht!

LIP und LDP verweisen auf den zuletzt von der FPU ausgeführten Befehl sowie, falls erforderlich, den dabei verwendeten Operanden. Das *last instruction pointer register* LIP enthält hierzu die Adresse des zuletzt bearbeiteten Coprozessor-Befehls (der pointer zeigt dabei auf das erste zur Befehlssequenz gehörende Präfix, falls vorhanden), das *last data pointer register* LDP die Adresse des zuletzt verwendeten Datums. Benötigte der zuletzt ausgeführte Befehl keinen Operanden, ist der Inhalt von LDP undefiniert. Op, *opcode*, enthält den Opcode des Befehls, besser: seine um eventuelle Präfixe und die »oberen« 5 Bits gekürzten zwei Bytes, die den Befehl codieren. (Jeder FPU-Befehl besteht aus genau zwei Code-Bytes, wobei Byte 1 immer mit den Bits 11011_b beginnt (»ESC«-Sequenz). Daher dienen lediglich die »untersten« drei Bits 0 bis 2 des ersten Bytes sowie die acht Bits des zweiten Bytes (= 11 Bits) der Codierung der FPU-Befehle.)

Wozu diese drei Register? Anders als CPU-Exceptions, die entweder unmittelbar vor oder nach dem auslösenden CPU-Befehl erzeugt werden, werden FPU-Exceptions meist erst unmittelbar vor dem nächsten FPU-Befehl erzeugt! (Eine Ausnahme: der CPU-Befehl WAIT prüft auch, ob eine FPU-Exception anhängig ist.) Das bedeutet, dass zwischen dem FPU-Befehl, der die exception verursacht, und dem, der sie auslöst, beliebig viele CPU-Befehle liegen können. Daher ist es in den seltensten Fällen möglich, zum Zeitpunkt der Bearbeitung der FPU-Exception durch den exception handler aus dem Inhalt des EIP (extended instruction pointer) der CPU auf die Adresse des Befehls zu schließen, der für die FPU-Exception verantwortlich ist. Die FPU muss daher die Adresse des Exception-Verursachers zwischenspeichern. Analoges gilt für den Datenzeiger, da sich ja zwischen Verursachung der Exception und deren Auslösung z.B. die Segmentadresse des Datensegmentes geändert haben könnte. Op, LIP und LDP sind also als Informationsquelle für FPU-Exceptionhandler gedacht und unverzichtbar. Für alle anderen Belange sind sie unwichtig.



Wenn aber Op, LIP und LDP nicht direkt ansprechbar sind – wie kommt man dann an ihre Informationen heran? Indirekt! Und zwar, indem mit einem entsprechenden Befehl (z.B. FSAVE) ein Speicherabbild aller Register der FPU erzeugt wird und dann die entsprechenden Speicherstellen ausgelesen werden. Auf diese Weise ist es auch möglich, im gewissen Rahmen den Inhalt der Register zu ändern: In einem zunächst erzeugten Speicherabbild der FPU-Register werden die gewünschten Änderungen vorgenommen und dieses Abbild dann in die FPU-Register zurückgeschrieben (z.B. mit FRSTOR). Dies klappt aber nicht mit jedem Register und allen Daten. Wir werden darauf zurückkommen.

tag register

Analoges gilt auch für das tag register. Auch dieses Register kann nur über ein Speicherabbild der FPU-Register ausgelesen und ggf. verändert werden. In der Regel ist aber die direkte Auswertung des tag registers nicht erforderlich, da man an seine Information auch anders gelangen kann. Es enthält acht Zwei-Bit-Felder, die über den aktuellen Zustand der acht Rechenregister Auskunft geben:

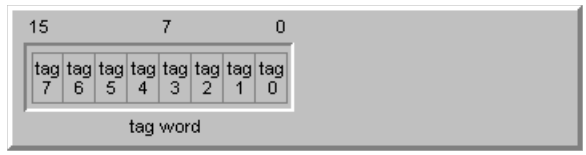


Abbildung 1.29: Speicherabbild des Tag-Registers der FPU

Ist das dem jeweiligen tag zugeordnete Rechenregister leer, so hat tag den Wert 11_b (= empty). Der Wert 00_b (= valid) zeigt an, dass ein gültiges Datum in Realzahldarstellung enthalten ist. 01_b (= zero) wird zur Markierung einer »Null« im Rechenregister verwendet, während 10_b (= special) einen Sonderfall signalisiert: Das Register enthält dann entweder eine NaN, eine Denormale, eine Infinite oder ein Datum in einem nicht unterstützten Format.

Wozu dient das tag register? Alle Informationen, die es anzeigt, können ja auch aus dem Registerinhalt selbst gewonnen werden. So sind die Kriterien für eine NaN, eine Infinite oder Denormale bekannt (vgl. »Codierung von Fließkommazahlen« auf Seite 788), auch kann festgestellt werden, ob alle Bits der Zahl gelöscht und ihr Wert damit 0 ist. Richtig – und falsch! Zwar können die Werte »valid«, »zero« und »special« durch Inaugenscheinnahme des Registerinhaltes festgestellt werden, nicht aber der Wert »empty«! Denn da bei der Codierung von Realzahlen keine Bitkombination existiert, die signalisieren würde »es gibt

mich überhaupt nicht!«, muss über das tag festgelegt werden, ob die aktuelle Bit-Konstellation im Register Müll von vorherigen Berechnungen ist oder ein aktuelles Datum.



Und dies ist genau die Hauptaufgabe des tag registers: anzuzeigen, ob das Register leer ist (11_b) oder nicht (10_b, 01_b, 00_b). Alle darüber hinaus gehenden Informationen sind lediglich dazu da, in bestimmten Situationen dem Programmierer zeitaufwändige Überprüfungen des Registerinhaltes auf Validität oder auf das Vorliegen des Wertes »0« zu ersparen. Konsequenterweise wird auch, wenn man durch Rückspeichern eines Speicherabbildes der FPU-Register die Tag-Felder lädt, lediglich geprüft, ob der Wert »empty« in das betreffende Tag-Feld zurückgeschrieben werden soll. Dann wird das dazugehörige Register tatsächlich als leer markiert. Alle anderen zurückgeschriebenen Werte (»valid«, »zero« und »special«) lösen lediglich eine Überprüfung des entsprechenden zurückzuschreibenden Registerinhaltes aus, die dann die tags anhand der neuen Registerinhalte korrekt setzt.

control word
status word

Neben den eigentlichen Rechenregistern sind daher lediglich die beiden 16-Bit-Register control register und status register von Interesse. Sie sind mit dem EFlags-Register der CPU vergleichbar und dienen der Steuerung der FPU (control word) oder machen bestimmte Zustände der FPU nach einem FPU-Befehl sichtbar (status word). Abbildung 1.30 zeigt den Aufbau dieser Register.

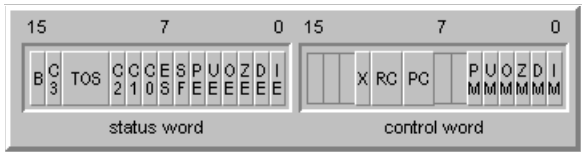


Abbildung 1.30: Speicherabbild des Status- und Kontrollregisters der FPU

Exception-Masken

Die Bits 0 bis 5 des control words (Abbildung 1.30, rechts) sind so genannte Maskenbits. Werden diese Bits gesetzt, »maskieren« sie die dazugehörige Exception: Sie wird dann nicht ausgelöst! Ist das jeweilige Bit dagegen gelöscht, so führt das Vorliegen einer entsprechenden Ausnahmesituation zum Auslösen der dazugehörigen Exception. Es gibt sechs Quellen von Exceptions, die auf diese Weise maskiert werden können:

- invalid operation (I)
- denormal operand (D)

- division by zero (Z)
- overflow (O)
- underflow (U)
- precision (P)

Detailliertere Informationen zu den Exceptions finden Sie weiter unten im Kapitel »FPU-Exceptions«.

precision control

Das Zwei-Bit-Feld precision control (PC; Bit 8 und 9 des control words) steuert die Genauigkeit, mit der die FPU arbeitet. Dies mag zunächst verwundern, da die Rechenregister der FPU ja 80 Bit breit sind und somit intern mit Daten vom Typ ExtendedReal arbeiten. Und das ist auch richtig und damit die Voreinstellung für die Genauigkeit (PC = 11_b). Doch ist eine ExtendedReal nach IEEE Std. 754 nicht definiert und somit Berechnungen mit diesen Daten nicht IEEE-konform. Daher gibt es zwei Einstellungen, die für IEEE-Konformität sorgen, indem sie die beiden einzigen standardisierten Realzahlen (SingleReal und DoubleReal) als Basis der Berechnungen definieren: PC = 00_b lässt die FPU mit der Genauigkeit einer SingleReal arbeiten, PC = 10_b mit der einer DoubleReal. Der Wert PC = 01_b ist reserviert.

Intern drückt sich dies so aus, dass bei PC = 00_b (SingleReal, 24 signifikante Mantissenbits, 8 Exponentenbits) die Bits 0 bis 39 (sie kodieren im Register die über 24 hinausgehenden Stellen der Mantisse) und die Bits 72 bis 78 (sie kodieren die über 8 hinausgehenden Stellen des Exponenten) auf »0« gesetzt und fortan *auch intern* nicht mehr berücksichtigt werden. Analog wird bei PC = 10_b (DoubleReal, 53 signifikante Mantissenbits, 11 Exponentenbits) mit den Bits 0 bis 10 sowie 75 bis 78 verfahren (vgl. Abbildung 1.31).

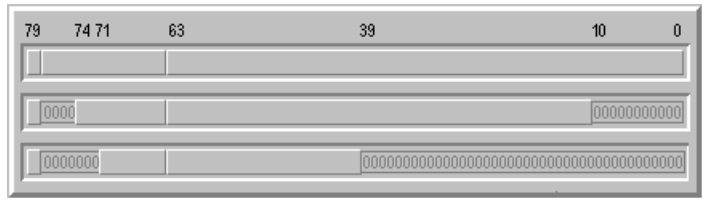


Abbildung 1.31: Unterschiede der FPU-internen Zahlendarstellung gemäß der verschiedenen Werte für precision control



Das Arbeiten mit PC = 00_b ist *nicht* das Gleiche wie das Laden einer SingleReal im Modus PC = 11_b! Wie Sie der Abbildung entnehmen können, werden bei PC = 00_b auch intern nur 24 Mantissen- und 8 Exponen-

tenbits bei Berechnungen verwendet. Laden Sie dagegen eine SingleReal unter PC = 11_b, wird sie zunächst in das interne Format ExtendedReal konvertiert. Dann wird intern solange mit Zwischenergebnissen höchster Genauigkeit gearbeitet, bis das Ergebnis nach Konversion als SingleReal in den Speicher zurückgeschrieben wird. Das hat Auswirkungen! Denn damit unterscheiden sich die Berechnungen, vor allem in Ketten, mit sehr hoher Wahrscheinlichkeit: Während es bei niedriger interner Genauigkeit schnell zu Über- oder Unterschreitungen des Wertebereichs mit den daraus folgenden Konsequenzen (Exceptions, Rundungen, »Nullsetzungen«, Ungenauigkeiten) kommen kann, die sich in Kettenrechnungen potenzieren können, ist dies bei hoher interner Präzision sehr viel seltener (wenn überhaupt) der Fall.



Sollten Sie mit anderen Genauigkeiten als der von ExtendedReals arbeiten (PC ≠ 11_b), so denken Sie bitte an eine weitere Quelle für Exceptions! Wenn Sie z.B. mit SingleReal-Genauigkeit (PC = 00_b) arbeiten und eine DoubleReal laden wollen, führt das fast unweigerlich zu einer exception, da die zu ladende DoubleReal mit einiger Wahrscheinlichkeit nicht im SingleReal-Format darzustellen ist.



Verwenden Sie daher PC und eine Herabsetzung der Rechengenauigkeit nur dann, wenn es wirklich notwendig ist und Sie, aus welchen Gründen auch immer, absolut konform zum IEEE Std. 754 sein müssen. Es macht einfach in der Regel keinen Sinn, bewusst falsche oder nicht exakte (besser: exaktere) Ergebnisse zu erzeugen, nur weil die Prozessoren mit precision control in der Lage sind, ungenauer zu rechnen und der IEEE-Standard von ExtendedReals keine Kenntnis nimmt. Ich muss gestehen: Mir fällt kein vernünftiger Grund ein, von der Standardeinstellung PC = 11_b abzuweichen – es sei denn, man programmiert einen Emulator für einen Chip, der nur mit der entsprechenden Genauigkeit arbeiten kann! Dann allerdings sollte sich die Emulation tatsächlich genauso verhalten wie das Original.

rounding control

Sehr viel sinnvoller als das Feld precision control ist dagegen das Zwei-Bit-Feld rounding control (RC) des control registers. Es steuert, wie die FPU im Falle von Rundungen des Ergebnisses vorzugehen hat. So codieren die Bitstellungen

00_b: Runden zur nächsten (geraden) Ziffer (Standardvorgabe); dies ist der Modus, den wir üblicherweise unter »Runden« verstehen: Ist der zu rundende Wert »näher« an der kleineren Zahl, wird zu ihr abgerundet, liegt er näher bei der größeren, wird er zu ihr aufge-

rundet. Kleiner Unterschied zu »unserem gewohnten« Runden: Liegt der zu rundende Wert exakt in der Mitte zwischen kleinerem und größerem Wert, kann also weder ab- noch aufgerundet werden, so wird zur nächsten geraden Zahl gerundet. Also abgerundet, wenn die kleinere Zahl die gerade Zahl ist, oder aufgerundet, wenn es die größere ist. (»Unsere gewohnte« Art der Rundung führt in einem solchen Fall grundsätzlich eine Aufrundung durch: 0.5 liegt genau in der Mitte zwischen der nächsthöheren (1.0) und nächstkleineren (0.0) Zahl und wird üblicherweise immer aufgerundet! Würden wir wie der Prozessor runden, so müssten wir 0.5 auf 0.0 abrunden, während 1.5 auf 2.0 auf- und 2.5 ebenfalls auf 2.0 abgerundet wird.)

01_b: Abrunden (= Runden in Richtung -∞); erklärt sich wohl genauso von selbst wie

10_b: Aufrunden (= Runden in Richtung +∞).

11_b: Abschneiden (= Runden in Richtung 0); bei dieser Rundungsart werden die »überflüssigen« Stellen einfach verworfen und nicht berücksichtigt.



Gerundete Ergebnisse sind »nicht exakte« Ergebnisse! Falls der Prozessor die Notwendigkeit zum Runden sieht, kann er offensichtlich das Ergebnis nicht exakt darstellen. Aus diesem Grunde setzt er als Folge des Rundens das precision exception flag PE und löst, so eine gesetzte precision exception mask PM dies nicht verbietet, eine precision exception (#P) aus.

infinity control Dieses Bit hat ab dem 80387 keine Bedeutung mehr, weshalb es in der Abbildung auch als »X« dargestellt ist. Es kann zwar noch gesetzt oder gelöscht werden, jedoch ohne weitere Auswirkungen: Die FPU ignoriert dieses Flag vollständig. Der Grund, warum es überhaupt noch existiert, ist Abwärtskompatibilität zum 8087/80287. Diese Co-Prozessoren kannten noch zwei verschiedene Modelle für »Unendlichkeiten« (siehe Codierung von Fließkommazahlen). Mit Bit 12, damals noch *infinity control* (IC) genannt, konnte zwischen dem projektiven (IC = 0) und affinen (IC = 1) Modell gewählt werden. Seit dem 80387 jedoch gibt es für alle FPUs nur noch das affine Modell.

status register Zunächst fällt bei der Betrachtung des status words, des Inhalts des status registers (vgl. Abbildung 1.30 auf Seite 191, links), auf, dass mit den Bits 0 bis 5 Flags existieren, die eine gewisse Namensverwandtschaft mit den korrespondierenden Flags aus dem control register haben.

exception flags

Und so ist es auch. Tritt eine der weiter oben genannten sechs Ausnahmebedingungen auf, so setzt die FPU das dazugehörige Flag im status register, bevor sie aus dem control register liest, wie im Folgenden weiterzuverfahren ist: Ist die korrespondierende exception mask gesetzt, ist die exception maskiert und wird nicht ausgelöst. Ist die Maske dagegen nicht gesetzt, so wird die korrespondierende Exception vor dem nächsten FPU-Befehl ausgelöst.

Zwei weitere Bits des status registers spielen ebenfalls beim exception handling eine Rolle: ES, exception summary, und SF, stack fault. SF dient der Unterscheidung der Ursache einer invalid operation exception (#I). Details erfahren Sie im Kapitel »FPU-Exceptions« auf Seite 529. ES ist, wenn Sie so wollen, eine Zusammenfassung aller unmaskierten exception flags. Es entsteht durch OR-Verknüpfung aller Bits 0 bis 6 des status registers, die laut Bit 0 bis 5 des control registers nicht maskiert sind, und ist damit immer dann gesetzt, wenn irgendein anderes unmaskiertes exception bit oder SF gesetzt ist.

busy flag

Wie das ehemalige infinity control flag (IC) im control register ist das busy flag (B) im status register ein Relikt aus dem Computer-Pleistozän um das Gespann 8086/8087: Es diente damals dem 8086 als Signal, ob der Co-Prozessor 8087 noch mit Berechnungen beschäftigt (busy) war oder nicht. Heute ist es nur noch aus Gründen der Abwärtskompatibilität vorhanden und hat den gleichen Wert wie das exception summary flag (ES).

condition code

An dieser Stelle wird es nun so richtig interessant! Die Bits 8 bis 10 und 14 stellen den »condition code« (CC) dar. Sie übernehmen damit bei der FPU die gleiche Aufgabe wie die status flags im EFlags-Register der CPU bei Integer-Berechnungen: Sie signalisieren den aktuellen Zustand der FPU-Register nach einer FPU-Instruktion. Je nach Instruktion können natürlich die Ergebnisse unterschiedliche Bedeutung haben und die Flags des condition code damit unterschiedliche Bedingungen darstellen. So gibt es bestimmte Zustände nach Vergleichen zweier Zahlen, aufgrund der Untersuchung eines Datums, nach aus welchem Grund auch immer unvollständig durchgeführten Operationen oder aufgrund sonstiger Widrigkeiten des (Programmierer-)Lebens.



Bei der Betrachtung der CPU wurde festgestellt, dass das EFlags-Register mit seinen Statusflags eine Hilfestellung der Interpretation des Ergebnisses von arithmetischen Berechnungen gibt, indem die Zustände der Statusflags Grundlage für bedingte Befehle (Jcc, SETcc, etc.) und da-

mit Programmverzweigungen sind. Mit den condition codes haben wir bei der FPU eine vergleichbare Ausgangssituation. Doch leider gibt es keine CPU-Befehle (die ja für Programmverzweigungen verantwortlich sind!), die durch FPU-Flags zu steuern wären. Daher müssen die condition codes irgendwie »in die CPU« kommen. Wie könnte das erfolgen?

Betrachten wir hierzu einmal die beiden betroffenen Register gemeinsam: das EFlags-Register der CPU und das status word des status registers der FPU.

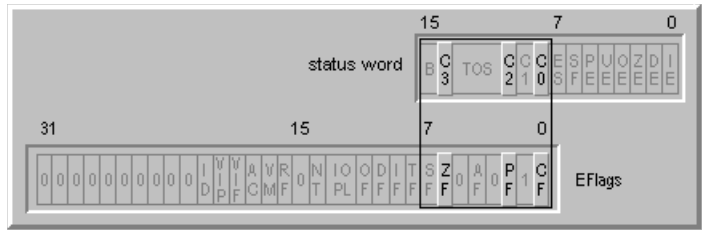


Abbildung 1.32: Korrespondenzen zwischen Statusflags der FPU (condition code) und Statusflags im EFlags-Register der CPU

Wie Sie in Abbildung 1.32 sehen können, hätten die condition codes C3, C2 und C0 im zero flag, parity flag und carry flag einen Partner, wenn man das höherwertige Byte aus dem status register in das niedrigstwertige Byte des EFlags-Registers bekommen könnte. Wenn jetzt dann auch noch die Bedeutungen der condition code flags mit denen des EFlags-Registers so übereinstimmten, dass man die aus dem CPU-Befehlssatz kommenden bedingten Befehle einsetzen könnte, hätten wir eine elegante Art und Weise, wie die CPU auf FPU-Flags reagieren könnte.

FSTSW und SAHF

Bedingung #1 ist erfüllbar, wenn auch über einen kleinen Umweg. Den Befehl FSTSW gibt es in einer Kombination mit dem »Register« AX (FSTSW AX), der das status word aus dem FPU-Statusregister in AX kopiert. Von dort kann das »obere« Byte (AH) mittels SAHF (*st*ore *AH* in *f*lags) in das niedrigstwertige Byte im EFlags-Register kopiert werden.



Ab der P6-Familie von Intel (Pentium Pro, Pentium II und Pentium III) gibt es Pendants zu den FPU-Vergleichsbefehlen, die direkt die EFlags-Register benutzen. Der Umweg über das status register und die Befehlskombination FSTSW AX – SAHF ist damit obsolet. Wir werden weiter unten darauf zurückkommen. Übrigens: Den Befehl FSTSW AX gibt es erst ab dem 80386. Davor konnte FSTSW direkt kein Register an-

sprechen, es musste eine Speicherstelle involviert werden: FSTSW [WordVar] – MOV AX, [WordVar] – SAHF.

Und was Bedingung #2 betrifft: Die Intel-Ingenieure waren so klug, nach Vergleichen C3 zur Unterscheidung der Gleichheit heranzuziehen, weshalb es eine zum zero flag identische Bedeutung hat, ihm also entspricht. C2, das mit dem wenig benutzten parity flag kommuniziert, übernimmt die Aufgabe, die Nicht-Vergleichbarkeit zu signalisieren. Und C0, das FPU-Pendant zum carry flag, schließlich entscheidet, welcher der beiden Operatoren größer ist. C1 spielt eine untergeordnete Rolle, da es nach den meisten Instruktionen in der Regel auf »0« gesetzt ist, falls nicht ein FPU-Stack-Über- oder Unterlauf stattgefunden hat. Oder es wird bei »nicht exakten« (also gerundeten) Ergebnissen verwendet, um anzuzeigen, ob auf- oder abgerundet wurde. Alles in allem Informationen, die nur in Spezialfällen interessant sind (oder welchen Wert messen Sie der Erkenntnis bei, dass nach einer Subtraktion die 63ste binäre Nachkommastelle abgerundet wurde?) Es ist daher nicht zwingend erforderlich, ihm ein Pendant in EFlags zur Seite zu stellen: Falls jemand diese Informationen braucht, soll er das status word »von Hand« auswerten!



Bitte beachten Sie einen wesentlichen Unterschied zu der Situation mit Integers. Integers können vorzeichenlos oder vorzeichenbehaftet sein. Wie wir in Kapitel »CPU-Operationen« gesehen haben, trägt die CPU dem Rechnung, indem sie zwei »Flag-Sätze« zur Interpretation definiert: Die für vorzeichenlose Integers (zero flag, carry flag) und die für vorzeichenbehaftete (zero flag, sign flag und overflow flag). Bei der FPU kommen nur vorzeichenbehaftete Zahlen zum Einsatz.

Unglücklicherweise nun stimmen die condition code flags nicht mit den bei vorzeichenbehafteten Integers involvierten Statusflags überein (sign flag, zero flag overflow flag), sondern mit denen vorzeichenloser (zero flag, carry flag). Dies bedeutet, dass Sie bei der FPU nach arithmetischen Befehlen mit (*vorzeichenbehafteten*) Realzahlen nur die bedingten Befehle einsetzen können, die im Falle der CPU bei der Verwendung arithmetischer Instruktionen mit *vorzeichenlosen* Integers genutzt werden.

CC und Vergleiche

Bei Vergleichen kommt es darauf an, festzustellen, ob der eine Operand größer ist als der andere und, wenn ja, welcher. Wichtig ist auch, zu prüfen, ob der Vergleich überhaupt erlaubt ist oder nicht, was der Fall ist,

wenn mindestens einer der beiden Operanden keine gültige Realzahl ist. Letzteres wird durch C2 signalisiert: Ist C2 gesetzt, so sind die Operanden miteinander nicht vergleichbar, weshalb die anderen condition flags keine Bedeutung haben (sie sind dann auf »1« gesetzt). Ein gelöschtes C2 zeigt: Der Vergleich geht in Ordnung, C0 und C3 codieren das Ergebnis des Vergleichs. So sind die beiden Operanden wertmäßig gleich, wenn C3 gesetzt ist. Ist Operand 1, der Zieloperand, größer als Operand 2, der Quelloperand, so ist C0 gelöscht, andernfalls gesetzt. C1 ist »0«, wenn alles OK ist; andernfalls zeigt es im Rahmen einer stack fault exception (#IS) an, ob ein Stacküberlauf (C1 = 1) oder -unterlauf (C1 = 0) die Ursache für die exception war.



Sie können nach Vergleichsbefehlen die bedingten Befehle verwenden, die Sie im Falle von Ganzzahlvergleichen (CMP) mit vorzeichenlosen Integers verwenden würden, wie das im folgenden (wenig sinnvollen!) Codefragment dargestellt wird. Die Verwendung der für vorzeichenbehaftete Integers gedachten bedingten Befehle (z.B. JL, JLE, JG, JGE) würde hier zu sehr schwer aufzufindenden Programmierfehlern führen: Das von diesen Befehlen geprüfte overflow flag wird durch SAHF nicht angetastet und das sign flag hat kein korrespondierendes condition code flag, sondern »korrespondiert« mit dem busy flag (und damit dem ES flag) aus dem status word.

```
        FSTSW   AX
        SAHF
        JP      NC      ; jump on parity
        JE      Equal   ; jump if equal
        JB      Less    ; jump if below
        JBE     LOE     ; jump if below or equal
        JA      Greater ; jump if above
        JAE     GOE     ; jump if above or equal
NC:                     ; nicht vergleichbare Operatoren
Equal:                  ; Operatoren sind gleich
Less:                   ; Operand 1 < Operand 2
LOE:                    ; Operand 1 ≤ Operand 2
Greater:                ; Operand 1 > Operand 2
GOE:                    ; Operand 1 ≥ Operand 2
```

top of stack

Bleiben noch die Bits 11 bis 13 des status words. Sie stellen den »top of stack« (TOS) dar und werden als vorzeichenlose 3-Bit-Integer betrachtet, die damit Werte zwischen 0 und 7 annehmen können. Der TOS hat eine herausragende Bedeutung, da sich die meisten FPU-Befehle auf ihn beziehen und ihn als impliziten Operanden verwenden. So können

z.B. die Ladebefehle FLD, FST/FSTP usw. den zu übertragenden Wert nur mit dem TOS austauschen und Vergleiche finden grundsätzlich mit dem TOS statt. Wir kommen bei der Besprechung der einzelnen Befehle darauf zurück.

Rechenregister

Das TOS-Feld im status word hat eine weitere wesentliche Bedeutung, die damit zusammenhängt, dass die Rechenregister der FPU nicht direkt angesprochen werden können! Mit anderen Worten: Es gibt keinen FPU-Befehl, dem Sie einen der Registernamen R0 bis R7 als Operand übergeben könnten! Der Grund hierfür ist einfach: Die FPU arbeitet mit einem »Registerstapel«, einem register stack. Solche Stapel werden immer dann eingesetzt, wenn Berechnungen nach »umgekehrt polnischer Notation« (UPN) erfolgen.

UPN

Was ist UPN? Im Prinzip nichts Aufregendes! Sondern schlicht und ergreifend eine im Vergleich zur »normalen« etwas andere Art der bei Berechnungen verwendeten Semantik, die in letzter Konsequenz hardwareseitig durch einen stack unterstützt werden muss. Wir alle haben in der Schule gelernt, dass man die Bildung einer Summe aus zwei Zahlen wie folgt darstellt:

```
Summand1 + Summand2 = Summe
```

Übertragen auf die Eingabe z.B. in einen Taschenrechner heißt das: Tippe Summand1 in den Rechner, drücke die Operationstaste (hier: die Additionstaste) und tippe Summand2 in den Rechner. Danach drücke die Gleichheitstaste, um das Ergebnis angezeigt zu bekommen. Dies nennt man die »algebraische« Notation, da sie sich direkt an algebraische Konventionen anlehnt. Zur Realisierung sind nur zwei »Register« erforderlich: Der Speicher für einen Operanden und das Register, in das gerade die Eingabe erfolgt. Die mathematische Operation sorgt dafür, dass die Eingabe des ersten Operanden abgeschlossen wird, indem er aus dem Eingaberegister in den Speicher übertragen wird, um Platz für die Eingabe des zweiten Operanden zu schaffen. Die Gleichheitstaste wiederum löst dann die eigentliche, zu diesem Zeitpunkt schon bekannte mathematische Verknüpfung der Inhalte von Speicher und Eingaberegister aus und sorgt für die Anzeige.

Die »umgekehrt polnische« Notation dagegen kümmert sich erst um das Erfassen der beiden Operanden, bevor die Operation ausgelöst wird.

```
Summand1 ↑ Summand2 += Summe
```

Hierbei bedeutet »↑«: zwischenspeichern und »+=«: addieren zu. Das heißt, dass die Eingabe des ersten Summanden durch expliziten Transfer in ein Register abgeschlossen werden muss, um Platz für den zweiten Operanden zu schaffen, der nun eingegeben wird. Bis zu diesem Zeitpunkt ist noch nicht bekannt, welche Operation nun durchgeführt werden soll! Beim jetzt folgenden Drücken der »Additionstaste« passieren mehrere Dinge: Die Eingabe des zweiten Operanden wird abgeschlossen, die mathematische Verknüpfung durchgeführt und das Ergebnis angezeigt. Auch in diesem Fall sind zwei Register erforderlich.

Ohne nun in eine Diskussion über das Für und Wider eintreten (Verfechter beider »Glaubensrichtungen« können tagelang Gründe für den Beweis anführen, dass ihre Methode die bessere ist!) oder Beweisversuche auf die eine oder andere Art anstellen zu wollen: Die UPN ist aufgrund ihrer Kompaktheit gerade im Ingenieurs- und wissenschaftlichen Bereich sehr beliebt. So arbeiten auch heute noch die meisten Taschenrechner aus dem technisch/wissenschaftlichen Bereich mit UPN. Und aus gleichen Gründen auch die FPU.

FPU-Stack

Deren acht Register bilden dazu einen Stapel. Und jeder Stapel hat ein »oberes Ende«, engl.: top of stack. Dieser TOS zeigt also das Register an, das »oben« ist. Aber ist das nicht klar? Es ist das »oberste« Register oder, wenn man so will, das mit der höchsten Nummer, wenn man »unten« bei 0 zu zählen beginnt. Ja – wenn man einen statischen Stapel hat! Und nein – wenn man einen dynamischen Stapel hat!

Um das zu erläutern, betrachten Sie bitte Abbildung 1.33, links, und stellen Sie sich einfach vor, Sie wollen umziehen. Zum Verpacken Ihrer Habe stehen Ihnen acht Kisten und zwei Regale zur Verfügung: Das im Keller ist ein Hängeregal und kann sieben leere Kisten aufnehmen, wobei die Kisten von oben beginnend untereinander gehängt werden müssen, das im Erdgeschoss ist ein »normales« Regal, in dem die Kisten aufeinander gestellt werden müssen, und kann acht Kisten aufnehmen. Der Keller ist gerade so hoch, dass das Regal hineinpasst: Es gibt keinerlei »Luft nach oben oder unten«. Falls Sie mehrere Kisten von oben nach unten oder umgekehrt umschichten wollen, können Sie dies nur »am Stück« – Umsortieren der Kisten außerhalb der Regale ist nicht möglich! Aus Platzgründen können Sie die Kisten nur in einem der beiden Regale aufbewahren, nicht etwa »frei beweglich« in der Wohnung. Und noch etwas: Sie können nur an die Kiste frei heran, die im Erdgeschoss »ganz oben« und daher von oben frei zugänglich, also die »Spitze des Kistenstapels« ist.

Um nun den Umzug so effektiv wie möglich zu gestalten, beschriften Sie die Kisten außen mit einem Stichwort, was drin ist: »Küche«, »Esszimmer«, »Wohnzimmer«, »Arbeitszimmer« usw. Abbildung 1.33, links zeigt diese Ausgangssituation. Ganz oben, also Kistenstapelspitze und damit frei zugänglich, ist die Kinderzimmer-Kiste. In diese Kiste sammeln Sie nun alles um sich herum ein.

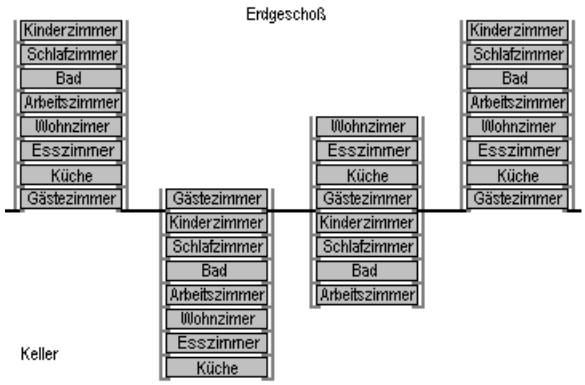


Abbildung 1.33: Illustration der Arbeitsweise des FPU-Stacks mit Hilfe eines Stapels Umzugskisten

Anschließend möchten Sie die nächste Kiste füllen. Doch Ihr Partner, dessen Aufgabe das Heranschaffen der Utensilien aus den verschiedenen Zimmern ist, hat noch nichts aus dem Schlafzimmer geholt, auch nicht aus dem Bad, sondern aus dem Gästezimmer. ›Dumme Sache‹, denken Sie sich, muss ich also umschichten. Daher holen Sie zunächst die Kinderzimmer-Kiste, bringen sie in den Keller und hängen sie oben ins Regal. Ebenso verfahren Sie mit der Schlafzimmer-Kiste, der Bad-Kiste usw., bis Sie endlich an die Gästezimmer-Kiste kommen. Das Ergebnis dieser »Hochstapelei« sehen Sie in Abbildung 1.33, Mitte links. Und plötzlich ist das Gästezimmer bzw. die ihm zugeordnete Kiste TOS! Nachdem Sie alles um sich herum in die Kiste verfrachtet haben, stapeln Sie wie gehabt um, um an die Wohnzimmer-Kiste heranzukommen (Abbildung 1.33, Mitte rechts). Denn es sammeln sich dank Ihres Partners Wohnzimmer-Einrichtungsgegenstände um Sie. »Sag mal, Schatz, wo ist eigentlich die Kiste mit den Sachen aus dem Gästezimmer? Ich kann dein Gekritzel nicht lesen!«, fordert Sie nun Ihr Partner – er hatte etwas vergessen. Sie denken kurz nach: ›Die Wohnzimmer-Kiste ist ganz oben, dann kommt das Esszimmer, die Küche ...‹. »Die vierte von oben!«, antworten Sie.

In der Zwischenzeit hat Ihr Partner noch Sachen aus dem Kinderzimmer geholt, die er dort übersehen hatte, und in Ihrem Umfeld deponiert. Schnell machen Sie sich daran, die Kisten umzustapeln, um die restlichen Spielsachen endlich aus den Augen zu bekommen! Das Ergebnis sehen Sie in Abbildung 1.33, rechts. »In der vierten Kiste von oben, sagtest du? Da finde ich nur deinen Computer-Kram!«, meldet sich Ihr Partner. ›Vierte?‹, denken Sie und fragen »Wieso vierte?« »Das G-ä-s-t-e-z-i-m-m-e-r! Die Sachen aus dem Gästezimmer!«, reklamiert Ihr Partner. Sie haben schnell nachgerechnet: »Achte von oben!« »Warum sagtest du dann vierte?«, will Ihr Partner wissen. »Weil es eben noch die vierte war.«

So dynamisch geht es bei der FPU zu! Beschriften Sie die Kisten – Pardon! Register – von ganz unten nach oben mit R0 bis R7, denken Sie daran, dass bei Computern die Nummerierung immer mit »0« und nicht mit »1« beginnt, halten Sie die Register fest und verschieben den TOS (im Beispiel von eben war der TOS fest – immer oben! – und die Register mussten bewegt werden, was aber zum gleichen Ergebnis führt!) und schon haben Sie den FPU-Stack. Er wird gebildet aus den festen 80-Bit-Registern, die aber dynamisch und somit indirekt adressiert werden: In Form eines Bezugs auf den jeweiligen top of stack. Das bedeutet: Die Namen, die Sie als Operanden in FPU-Befehlen verwenden können, heißen ST(0) bis ST(7), was für stack #0 bis stack #7 steht. ST(0) ist immer auch der TOS. Um welches Hardware-Register (R0 bis R7) es sich dabei handelt, sagt Ihnen das Feld TOS im status register. Aber eigentlich braucht Sie das nicht wirklich zu interessieren!

Betrachten Sie nun einmal Abbildung 1.34. Sie sehen dort die acht Rechenregister R0 bis R7 sowie die wichtigsten Teile aus dem control word (precision control und rounding control) und dem status word (TOS, condition code). Ebenfalls verzeichnet sind die tag fields des tag registers. Das TOS-Feld im status register enthält den Wert $011_b = 3_d$. Damit wird der TOS von Register R3 gebildet, das damit über ST(0) angesprochen werden kann und auf diese Weise Bezugspunkt der dynamischen Register-Adressierung ist. Das bedeutet: R3 = ST(0), R4 = ST(1), R5 = ST(2), R6 = ST(3), R7 = ST(4), R0 = ST(5), R1 = ST(6) und R2 = ST(7). Allgemein gilt ST(X) = R(Y), wobei die Relation besteht:

Y = (TOS + X) mod 8 bzw. X = (Y – TOS + 8) mod 8

Wie gesagt, diese Umrechnung braucht Sie nicht zu interessieren, da Sie eh keine Chance haben, die physikalischen Register selbst anzusprechen – es sei denn, Sie gehen wieder über die Speicherabbilder der FPU-Register ...

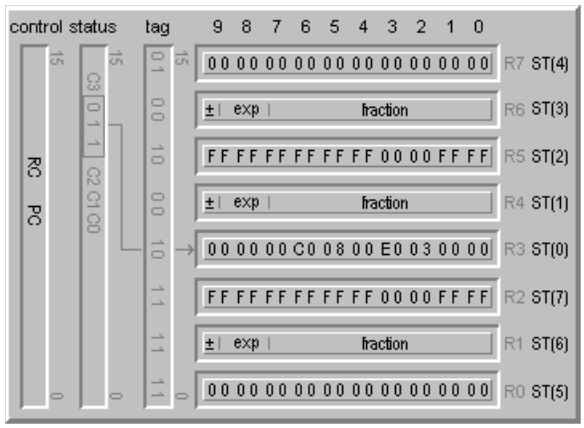


Abbildung 1.34: Speicherabbild der FPU-Register, ihre dynamische Ansprache und Zusammenhänge mit den Tag-, Status- und Kontrollregister der FPU



Dem aufmerksamen Leser wird eine Diskrepanz zwischen dem Umzugsbeispiel aus Abbildung 1.33 und der Abbildung 1.34 aufgefallen sein. So war im Umzugsbeispiel der TOS die oberste Kiste, während in Abbildung 1.34 der TOS das unterste Register ist (Wen stört, dass »unter« dem TOS noch die Stack-Register ST(5) bis ST(7) liegen, möge sie im Geiste oberhalb von ST(4) ansiedeln! Sie liegen lediglich wegen der physikalisch bedingten »Starrheit« der physikalischen Register dort. Sortiert man nach den Stackregistern, liegt der TOS ganz unten!). Warum ist das so? Aufgrund einer Konvention, die sich wegen der Speicherausnutzung im Computer-Pleistozän eingebürgert hat. Und diese Konvention besagt, dass Stacks »von oben nach unten« wachsen, so wie die Stalagtiten in Tropfsteinhöhlen. Sicher sagt Ihnen der Begriff Heap etwas. Dieser Heap hat in Tropfsteinhöhlen sein Pendant in den Stalagmiten, die »von unten nach oben« wachsen. So wie Stalagmiten und Stalagtiten aufeinander zu wachsen, wachsen auch Heap und Stack aufeinander zu. Und daher liegt die Spitze eines Stacks immer an dessen unterem Ende, seine Basis am oberen! Falls Sie das im Beispiel oben nachvollziehen möchten, stellen Sie sich einfach auf den Kopf und sortieren Sie so die zu verstauenden Utensilien in die entsprechenden Kisten!

Interpretation!

Auf den ersten Blick scheinen in Abbildung 1.34 die Register ST(4) und ST(5) beide den Wert »0«, die Register ST(2) und ST(7) die gleiche NaN und die Register ST(1), ST(3) und ST(6) jeweils eine Realzahl zu enthalten. Dies ist aber falsch! Betrachtet man die zu den einzelnen Registern

gehörigen Tag-Felder des Tag-Registers, stellt man fest, dass die Register ST(5) bis ST(7) leer sind (tag: 11_b = empty). Nur Register ST(4) enthält daher tatsächlich den Wert »0« (tag: 01_b = zero; alle Bits des Registers gelöscht), und nur Register ST(2) eine NaN (tag: 10_b = special, der Registerinhalt codiert eine NaN). Die Register ST(1) und ST(3) enthalten eine Realzahl (tag: 00b = valid) und ST(0), der top of stack, enthält eine Denormale (tag: 10_b = special; Registerinhalt codiert eine Denormale).

push stack
pop stack

Wer nun bestimmt den Wert im TOS-Feld des status registers und damit den top of stack? Antwort: einerseits Sie, wenn Sie wollen, andererseits einige FPU-Befehle. Dabei ist wichtig zu wissen, dass Sie keinen direkten Zugriff auf das TOS-Feld im status register haben (es gibt keinen FPU-Befehl, mit dem Sie das status register beschreiben könnten), also nicht ein bestimmtes Hardwareregister angeben können. Vielmehr können Sie nur durch Inkrementieren oder Dekrementieren des TOS-Feldes mit zwei FPU-Befehlen (FINCSTP, FDECSTP) seinen Inhalt verändern. Genau dieses Inkrementieren und Dekrementieren benutzen auch die TOS-verändernden FPU-Befehle. Das Dekrementieren des TOS-Feldes nennt man »stack pushing«, da das »unter« dem aktuellen TOS liegende physikalische Register TOS wird, der stack somit »nach oben verschoben«, gepusht wird; das Inkrementieren heißt analog »stack popping«.



Denken Sie in diesem Zusammenhang bitte nochmals an die Tropfsteinhöhlen mit nach unten wachsenden Stalagtiten und wundern Sie sich nicht, dass der Stack »nach oben« gepusht wird, wenn er unten wächst. Und beachten Sie den zyklischen Charakter bei der Vergabe der Stack-Nummern. Da es keine negativen Werte für den TOS geben kann, berechnet sich der neue TOS beim Dekrementieren, dem stack pushing, zu $TOS_{neu} = (TOS_{alt} - 1 + 8)$ modulo 8 und beim Inkrementieren, dem stack popping, zu $TOS_{neu} = (TOS_{alt} + 1)$ modulo 8.

Zugegeben: Nicht ganz einfach zu verstehen, das Ganze, aber man gewöhnt sich dran!

FPU-Befehle

Der FPU-Befehlssatz umfasst Befehle zum

- »einfachen« arithmetischen Manipulieren der Daten
- Durchführen transzendentaler Operationen
- Datenvergleich und Klassifizierung von Daten
- Datenaustausch

- Datenkonversion
- Laden von grundlegenden Konstanten
- Verwaltung der FPU

1.2.1 Grundlegende arithmetische Operationen

FADD
FSUB
FMUL
FDIV

Zu den arithmetischen Abläufen ist wohl nicht viel zu sagen: FADD addiert den Quelloperanden (zweiter Operand) zum Zieloperanden (erster Operand), FSUB subtrahiert den Quelloperanden vom Zieloperanden, FMUL multipliziert den Zieloperanden mit dem Quelloperanden und FDIV dividiert den Zieloperanden durch den Quelloperanden. Ein Geheimnis gibt es hier nicht.



Es sei jedoch bemerkt, dass die Befehle nur Realzahlen verarbeiten können. Für Integers gibt es mit FIADD, FISUB, FIMUL und FIDIV spezielle Befehle, die die verwendete Integer in eine Realzahl konvertieren und die arithmetische Operation dann durchführen (vgl. Seite 208). Für BCDs gibt es spezifische Ladebefehle, die die BCD in eine Realzahl umwandeln, bevor sie in arithmetischen Operationen eingesetzt werden können.

Operanden

Zieloperand muss immer ein FPU-Register sein, Quelloperand kann entweder ein FPU-Register oder eine Speicherstelle sein. Ist der Quelloperand eine Speicherstelle, so kann sie entweder eine DoubleReal (8 Bytes) oder eine SingleReal (vier Bytes) beherbergen. Ferner muss einer der beiden Operanden immer ST(0) sein. Das bedeutet, dass eine Operation mit einer arithmetischen Grundrechenart immer den TOS einbezieht.

Die arithmetischen Befehle gibt es in einer Form mit einem oder mit zwei Operanden. Die Ein-Operanden-Form impliziert immer ST(0) als Ziel, wobei der zweite Operand eine Speicherstelle (SingleReal oder DoubleReal) sein muss. Somit sind folgende Operandenkombinationen möglich (XXX steht für FADD, FSUB, FMUL oder FDIV):

- Ein-Operanden-Form; Ziel ist immer ST(0), Quelle eine Speicherstelle
  ```
  XXX Mem32 (≡ XXX ST(0), Mem32);
  XXX Mem64 (≡ XXX ST(0), Mem64)
  ```
- Zwei-Operanden-Form; beide Operanden sind FPU-Register, wobei eines davon der TOS sein muss
  ```
  XXX ST(0), ST(i)
  XXX ST(i), ST(0)
  ```

Im Unterschied zu den Verhältnissen bei der CPU können FPU-Register neben »echten« Zahlen auch Werte wie z.B. ±∞ oder NaNs (not a number) enthalten. Das bedeutet, dass das Ergebnis der arithmetischen Operationen ggf. nicht dem entspricht, was man erwartet. Bei den folgenden Betrachtungen wird davon ausgegangen, dass die Ergebnisse nicht zu einem Über- oder Unterlauf führen und somit entsprechende Exceptions nicht auslösen würden.

FADD In Tabelle 1.12 sind die Ergebnisse dargestellt, die nach FADD mit unterschiedlichen Kombinationen von Werten für den Ziel- und Quelloperanden auftreten können. FPU-Exceptions der Klasse invalid arithmetic operand (#IA) werden lediglich ausgelöst, wenn beide Operanden eine Infinite enthalten, wobei die beiden Infiniten unterschiedliches Vorzeichen besitzen müssen. Sobald eine NaN involviert ist, ist auch das Ergebnis eine NaN. Werden zwei Null-Operanden mit unterschiedlichem Vorzeichen (ja, es gibt bei der FPU +0 und –0!) addiert, ist das Ergebnis immer +0 mit einer Ausnahme: Zeigt des Feld rounding control eine Rundung in Richtung -∞, so ist das Ergebnis der Addition –0.

Quelloperand (src) \ Zieloperand (dest)	-∞	-real	-0	+0	+real	+∞	NaN
-∞	-∞	-∞	-∞	-∞	-∞	#IA	NaN
-real	-∞	-real	src	src	±real, 0	+∞	NaN
-0	-∞	dest	-0	±0	dest	+∞	NaN
+0	-∞	dest	±0	+0	dest	+∞	NaN
+real	-∞	±real, 0	src	src	+real	+∞	NaN
+∞	#IA	+∞	+∞	+∞	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.12: Ergebnisse des Befehls FADD mit unterschiedlichen Werten für Quell- und Zieloperand

FSUB Auch bei FSUB können FPU-Exceptions der Klasse invalid arithmetic operand (#IA) auftreten, wie Tabelle 1.3 zeigt, und zwar immer dann, wenn zwei Infinite gleichen Vorzeichens von einander subtrahiert werden sollen. Auch in diesem Fall ist das Ergebnis ebenfalls eine NaN, wenn mindestens einer der Operanden eine NaN enthält. Auch im Falle der Subtraktion hängt das Vorzeichen der resultierenden Null beider Subtraktion zweier Nullen mit gleichem Vorzeichen davon ab, welchen Wert das Feld rounding control hat: Wird in Richtung -∞ gerundet, resultiert –0, ansonsten +0.

Quelloperand (src) \ Zieloperand (dest)	-∞	-real	-0	+0	+real	+∞	NaN
-∞	#IA	-∞	-∞	-∞	-∞	+∞	NaN
-real	-∞	±real, 0	-src	-src	+real	+∞	NaN
-0	-∞	dest	±0	+0	dest	+∞	NaN
+0	-∞	dest	-0	±0	dest	+∞	NaN
+real	-∞	-real, 0	-src	-src	±real, 0	+∞	NaN
+∞	-∞	-∞	-∞	-∞	-∞	#IA	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.13: Ergebnisse des Befehls FSUB mit unterschiedlichen Werten für Quell- und Zieloperand

Tabelle 1.14 zeigt die Verhältnisse nach FMUL. FPU-Exceptions der Klasse invalid arithmetic operand (#IA) treten hier auf, wenn ein Operand Null ist und der andere eine Infinite. Natürlich resultiert auch hier eine NaN, wenn mindestens einer der Operanden eine NaN enthält. *FMUL*

Quelloperand \ Zieloperand	-∞	-real	-0	+0	+real	+∞	NaN
-∞	+∞	+∞	#IA	#IA	-∞	-∞	NaN
-real	+∞	+real	+0	-0	-real	-∞	NaN
-0	#IA	-real	+0	-0	-0	#IA	NaN
+0	#IA	-real	-0	+0	+0	#IA	NaN
+real	-∞	±real	-0	+0	+real	+∞	NaN
+∞	-∞	-∞	#IA	#IA	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.14: Ergebnisse des Befehls FMUL mit unterschiedlichen Werten für Quell- und Zieloperand

Schließlich zeigt Tabelle 1.15 die Situation nach FSUB. Hier gibt es zwei Möglichkeiten für FPU-Exceptions: Sobald beide Operanden Null sind oder beide Operanden eine Infinite enthalten, wird eine invalid arithmetic operand exception (#IA) ausgelöst. Enthält der Zieloperand eine gültige Realzahl und der Quelloperand eine Null, wird im Falle unmaskierter Exceptions die zero divide exception (#Z) ausgelöst. Das Ergebnis im Zieloperanden ist dann eine Infinite. Ist die #Z maskiert, wird kein Ergebnis in das Zielregister geschrieben! *FDIV*

Quelloperand \ Zieloperand	-∞	-real	-0	+0	+real	+∞	NaN
-∞	#IA	+0	+0	-0	-0	#IA	NaN
-real	+∞	+real	+0	-0	-real	-∞	NaN
-0	+∞	#Z	#IA	#IA	#Z	-∞	NaN
+0	-∞	#Z	#IA	#IA	#Z	+∞	NaN
+real	-∞	-real	-0	+0	+real	+∞	NaN
+∞	#IA	-0	-0	+0	+0	#IA	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.15: Ergebnisse des Befehls FDIV mit unterschiedlichen Werten für Quell- und Zieloperand

Condition Code C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte. Bei einer inexact-result exception (#P) ist C1 = 0, wenn nicht aufgerundet wurde, nach Aufrundung ist C1in diesem Falle 1.

Top of Stack FADD, FSUB, FMUL und FDIV sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Ausnahme: Falls der Assembler die operandenlose Form zulässt, wird sie als FADDP, FSUBP, FMULP bzw. FDIVP interpretiert. Siehe dort.

FIADD
FISUB
FIMUL
FIDIV FIADD, add integer to floating-point value, FISUB, subtract integer from floating-point value, FIMUL; multiply floating-point value by integer, und FIDIV, divide floating-point value by integer, ergänzen die arithmetischen Grundrechenarten um die Möglichkeit, auch Integers als Quelloperanden zu verwenden. Diese Integer werden vor der arithmetischen Verknüpfung in das Real-Format konvertiert. Ansonsten verhalten sich FIADD, FISUB, FIMUL und FIDIV absolut analog zu den in vorangehenden Abschnitt besprochenen Versionen, die nur Realzahlen als Operanden akzeptieren. Bei den folgenden Betrachtungen wird davon ausgegangen, dass die Ergebnisse nicht zu einem Über- oder Unterlauf führen und somit entsprechende Exceptions nicht auslösen würden.

FIADD In Tabelle 1.16 sind die Ergebnisse dargestellt, die nach FIADD mit unterschiedlichen Kombinationen von Werten für den Ziel- und Quelloperanden auftreten können. Im Unterschied zu Realzahlen können bei Integers nur positive Werte, negative Werte oder die Null auftreten. Sobald eine NaN involviert ist, ist auch das Ergebnis eine NaN.

Quellop. (src) \ Zieloperand (dest)	-∞	-real	-0	+0	+real	+∞	NaN
-integer	-∞	-real	src	src	±real, 0	+∞	NaN
0	-∞	dest	±0	+0	dest	+∞	NaN
+integer	-∞	±real, 0	src	src	+real	+∞	NaN

Tabelle 1.16: Ergebnisse des Befehls FIADD mit unterschiedlichen Werten für Quell- und Zieloperand

FISUB

Auch bei FISUB ist das Ergebnis ebenfalls eine NaN, wenn der Zieloperand eine NaN enthält, wie Tabelle 1.17 zeigt. Auch im Falle der Subtraktion hängt das Vorzeichen der resultierenden Null bei der Subtraktion zweier Nullen mit positivem Vorzeichen davon ab, welchen Wert das Feld rounding control hat: Wird in Richtung -∞ gerundet, resultiert –0, ansonsten +0.

Quellop. (src) \ Zieloperand (dest)	-∞	-real	-0	+0	+real	+∞	NaN
-integer	-∞	±real, 0	-src	-src	+real	+∞	NaN
0	-∞	dest	-0	±0	dest	+∞	NaN
+integer	-∞	-real, 0	-src	-src	±real, 0	+∞	NaN

Tabelle 1.17: Ergebnisse des Befehls FISUB mit unterschiedlichen Werten für Quell- und Zieloperand

FIMUL

Tabelle 1.18 zeigt die Verhältnisse nach FIMUL. FPU-Exceptions der Klasse invalid arithmetic operand (#IA) treten hier auf, wenn die Integer Null ist und die Realzahl eine Infinite. Natürlich resultiert auch hier eine NaN, wenn der Zieloperand eine NaN enthält.

Quellop. (src) \ Zieloperand (dest)	-∞	-real	-0	+0	+real	+∞	NaN
-integer	+∞	+real	+0	-0	-real	-∞	NaN
0	#IA	-real	-0	+0	+0	#IA	NaN
+integer	-∞	±real	-0	+0	+real	+∞	NaN

Tabelle 1.18: Ergebnisse des Befehls FIMUL mit unterschiedlichen Werten für Quell- und Zieloperand

FIDIV

Schließlich zeigt Tabelle 1.19 die Situation nach FIDIV. Hier gibt es zwei Möglichkeiten für FPU-Exceptions: Sobald die Integer Null ist (und somit durch Null dividiert werden soll!) und die Real eine Realzahl, wird im Falle unmaskierter Exceptions die zero divide exception (#Z) ausgelöst. Das Ergebnis im Zieloperanden ist dann eine Infinite. Ist die #Z

maskiert, wird kein Ergebnis in das Zielregister geschrieben! Enthalten beide Operanden eine Null (0 / 0), wird eine invalid arithmetic operand exception (#IA) ausgelöst.

		Zieloperand (dest)						
		-∞	-real	-0	+0	+real	+∞	NaN
Quellop. (src)	-integer	+∞	+real	+0	-0	-real	-∞	NaN
	0	-∞	#Z	#IA	#IA	#Z	+∞	NaN
	+integer	-∞	-real	-0	+0	+real	+∞	NaN

Tabelle 1.19: Ergebnisse des Befehls FIDIV mit unterschiedlichen Werten für Quell- und Zieloperand

Operanden Da gemäß dem unter FADD/FSUB/FMUL/FDIV dargestellten immer ein Operand der TOS sein muss, kommt für FIADD, FISUB, FIMUL und FIDIV nur die Ein-Operanden-Form für die Operanden in Frage. Quelle kann eine Integer (Mem16) oder eine LongInt (Mem32) sein

- Ein-Operanden-Form; Ziel ist immer ST(0), Quelle eine Speicherstelle

```
XXX Mem16 (≡ XXX ST(0), Mem16);
XXX Mem32 (≡ XXX ST(0), Mem32)
```

Condition Code C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte. Bei einer inexact-result exception (#P) ist C1 = 0, wenn nicht aufgerundet wurde, nach Aufrundung ist C1 = 1.

Top of Stack FIADD, FISUB, FIMUL und FIDIV sind Stack-neutral, was bedeutet, dass sich der Inhalt des TOS-Feldes im status register der FPU und dadurch der Zustand des FPU-Stacks durch die Operation nicht ändert. Der Inhalt des Zieloperanden wird mit dem Ergebnis der Operation überschrieben.

FADDP
FSUBP
FMULP
FDIVP

Häufig kommt es vor, dass nach einer arithmetischen Berechnung der Inhalt der Quelle nicht mehr interessiert. In diesem Fall belegt er ein FPU-Register unnütz mit Beschlag und müsste entfernt werden. Dies kann man automatisieren. Hierzu gibt es eine Version der Grundrechenarten, die im Anschluss an die Operation den Stack POPpt und somit die Quelle vom Stack entfernt.

Ansonsten verhalten sich die »P«-Befehle analog zu den »P«-freien und es gilt hier, was dort gesagt wurde.

Operanden

GePOPpt werden kann nur der TOS. Daher ist es logisch, dass der TOS auch nur als Quelloperand fungieren kann. Da bei allen arithmetischen FPU-Befehlen das Ziel ein FPU-Register sein muss, kommen für die POP-(= »P«-)Versionen nur Formen in Frage, die ein FPU-Register und den TOS benutzen. Dies kann über eine operandenlose Form erfolgen oder über eine Zwei-Operanden-Form, in der der TOS explizit als Quelle angegeben werden muss.

Somit verfügen die Befehle über folgende Formen (XXX steht für FADDP, FSUBP, FMULP und FDIVP):

- Operandenlose Form; Ziel ist immer ST(1), Quelle ST(0)
  `XXX (≡ XXX ST(1), ST(0))`
- Zwei-Operanden-Form; Ziel ist ein FPU-Register, Quelle ST(0)
  `XXX ST(i), ST(0)`



Manche Assembler erlauben auch eine operandenlose Form für die »P«-freien Versionen, also FADD, FSUB, FMUL und FDIV. Diese werden dann jedoch in die Opcodes von FADDP, FSUBP, FMULP bzw. FDIVP übersetzt.

Da nach der Operation ein POPpen des FPU-Stack erfolgt, ist auch klar, warum das Ziel nicht der TOS sein darf: Wäre das erlaubt, so würde das Ergebnis der Operation sofort wieder vernichtet, da durch das POPpen ST(0) als leer markiert und der Stackpointer inkrementiert wird.

Condition Code

C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte. Bei einer inexact-result exception (#P) ist C1 = 0, wenn nicht aufgerundet wurde, nach Aufrundung ist C1 = 1.

Top of Stack

FADDP, FSUBP, FMULP und FDIVP markieren den TOS als »empty« und inkrementieren (modulo 8) den Stackpointer im TOS-Feld des status register der FPU, POPpen also dadurch den Stack. Neuer TOS wird somit ST(1).

FSUBR
FSUBRP
FISUBR
FDIVR
FDIVRP
FIDIVR

Addition und Multiplikation sind kommutative Operationen. Das bedeutet, dass das Ergebnis unabhängig davon ist, ob der Quelloperand zum Zieloperanden addiert wird oder umgekehrt: Im Ziel liegt immer der gleiche Wert.

Dies ist bei den nicht-kommutativen Operationen Subtraktion bzw. Division anders. Hier spielt die Reihenfolge der Operanden sehr wohl

eine Rolle. Um dem Rechnung zu tragen und nicht einen unnötigen, zeitaufwändigen Austausch der Inhalte zweier Register bzw. eines Registers und einer Speicherstelle zu provozieren, gibt es alle Spielarten der betreffenden Befehle in einer »R«-Version. Diese zeichnet sich dadurch aus, dass die Reihenfolge der Operanden umgekehrt (»reverted«) wird:

Ziel := Quelle ○ Ziel anstelle von Ziel := Ziel ○ Quelle,

wobei »○« für die Operation Subtraktion bzw. Division steht. Alles andere bleibt gleich!



Bei der Nutzung der Tabelle 1.13, Tabelle 1.15, Tabelle 1.17 und Tabelle 1.19 beachten Sie bitte, dass hier aufgrund der Änderung der Operandenreihenfolge bei der Berechnungen formal Ziel- und Quelloperand vertauscht werden müssen.

Für weitere Informationen siehe die korrespondierenden »R«-freien Befehle

FSQRT Was wäre Fließkomma-Arithmetik ohne Wurzelbildung! Daher gibt es mit FSQRT, *square root*, auch einen Befehl, der dies ermöglicht.

Operanden Quell- und Zieloperand sind bei diesem Befehl impliziert: Es handelt sich beide Male um den TOS. Das bedeutet, der Wert im TOS wird durch seine Quadratwurzel ersetzt. Damit gibt es nur eine Möglichkeit, den Befehl aufzurufen:

```
FSQRT
```

Bei der folgenden Betrachtung wird davon ausgegangen, dass das Ergebnis nicht zu einem Über- oder Unterlauf führt und somit eine entsprechende Exceptions auslösen würde. Tabelle 1.20 zeigt dann den Inhalt des TOS vor und nach der Operation. Bei dem Versuch, negative Werte zu verwenden, wird eine invalid arithmetic operand exception #IA ausgelöst.

Quelloperand	-∞	-real	-0	+0	+real	+∞	NaN
Zieloperand	#IA	#IA	-0	+0	+real	+∞	NaN

Tabelle 1.20: Ergebnisse des Befehls FSQRT mit unterschiedlichen Werten als Eingaben

Condition Code

C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte. Bei einer inexact-result exception (#P) ist C1 = 0, wenn nicht aufgerundet wurde, nach Aufrundung ist C1 = 1.

Top of Stack

FSQRT ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FABS

FABS, absolute value, ersetzt den Wert im TOS mit seinem absoluten Wert, indem er das Vorzeichenbit (das MSB) löscht.

Operanden

Quell- und Zieloperand sind bei diesem Befehl impliziert: Es handelt sich beide Male um den TOS. Das bedeutet, der Wert im TOS wird durch seinen Betrag ersetzt. Damit gibt es nur eine Möglichkeit, den Befehl aufzurufen:

```
FABS
```

Bei der folgenden Betrachtung wird davon ausgegangen, dass das Ergebnis nicht zu einem Über- oder Unterlauf führt und somit eine entsprechende Exceptions auslösen würde. Tabelle 1.21 zeigt dann den Inhalt des TOS vor und nach der Operation. Bei dem Versuch, negative Werte zu verwenden, wird eine invalid arithmetic operand exception #IA ausgelöst

Quelloperand	-∞	-real	-0	+0	+real	+∞	NaN
Zieloperand	+∞	+real	+0	+0	+real	+∞	NaN

Tabelle 1.21: Ergebnisse des Befehls FABS mit unterschiedlichen Werten als Eingaben

Condition Code

C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte.

Top of Stack

FABS ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FCHS

FCHS, change sign, dreht das Vorzeichen der im TOS befindlichen Zahl um.

Operanden

Quell- und Zieloperand sind bei diesem Befehl impliziert: Es handelt sich beide Male um den TOS. Das bedeutet, der Wert im TOS wird

durch negierten Wert ersetzt. Damit gibt es nur eine Möglichkeit, den Befehl aufzurufen:

```
FCHS
```

Bei der folgenden Betrachtung wird davon ausgegangen, dass das Ergebnis nicht zu einem Über- oder Unterlauf führt und somit eine entsprechende Exception auslösen würde. Tabelle 1.22 zeigt dann den Inhalt des TOS vor und nach der Operation. Bei dem Versuch, negative Werte zu verwenden, wird eine invalid arithmetic operand exception #IA ausgelöst.

Quelloperand	-∞	-real	-0	+0	+real	+∞	NaN
Zieloperand	+∞	+real	+0	+0	+real	+∞	NaN

Tabelle 1.22: Ergebnisse des Befehls FCHS mit unterschiedlichen Werten als Eingaben

Condition Code C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte.

Top of Stack FCHS ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FPREM
FPREM1

FPREM und FPREM1, *partial remainder*, bilden den Rest einer Division (*modulus*) des Zieloperanden durch den Quelloperanden:

Remainder := Destination MOD Source



Welchen praktischen Wert haben diese Befehle? Was könnte bei Realzahlen, die ja per definitionem von ihrem Nachkommaanteil leben (sonst wären es Integers), erforderlich machen, Divisions*reste* zu bilden, vor allem, nachdem der Divisor selbst eine Realzahl darstellt? Antwort: FPREM/FPREM1 werden immer dann eingesetzt, wenn Werte als Parameter von periodischen Funktionen eingesetzt werden sollen, die Funktions-Argumente also auf die verwendete Periode abgebildet werden sollen/müssen. FPREM/FPREM1 reduzieren das Argument dann solange, bis es kleiner als die Periode ist. So kann z.B. zur Berechnung des Tangens das zu übergebende Argument auf den gültigen Wertebereich $0 \leq \text{Argument} < \pi/4$ abgebildet werden, indem es FPREM/FPREM1 mit einem Divisor $\pi/4$ übergeben wird.

Das Divisionsergebnis wird durch iterative Subtraktion des Divisors (Quelloperand) vom Dividenden (Zieloperand) erhalten. Die Iteration erfolgt solange, bis *Remainder* < *Source* ist. Hierbei ist aber lediglich eine maximale Reduktion des Dividenden-Exponenten um 63 möglich (was bedeutet, dass der Dividend nicht größer als $2^{63} \cdot$ Divisor sein darf). Sollte eine Restbildung innerhalb dieses Bereiches möglich sein, also *Rest* < *Divisor*, ist der Rest vollständig gebildet worden. Andernfalls spricht man von einem »Teil-Rest« (*partial remainder*), der dem Befehl auch den Namen gegeben hat. Dieser partial remainder kann durch erneutes Aufrufen von FPREM/FPREM1 erneut reduziert werden, und zwar so lange, wie der Programmierer das in einer Schleife für nötig hält.

Formal entspricht die iterative Subtraktion folgender Rechenvorschrift:

```
D := Integer(LOG2(Dividend) – LOG2(Divisor))
if D < 64
then
       N := Integer*(Dividend / Divisor)
else
       C ·  32 · C · 63 (implementation dependend)
       X := Dividend / 2^(D-C)
       N := Integer(X / Divisor)
R := IterateSubtraction(Dividend, Divisor, N)
```

Im ersten Schritt wird die Anzahl der erforderlichen Subtraktionen ermittelt. Hier gibt es zwei Möglichkeiten:

- Die Werte von Dividend und Divisor liegen *nicht* um mehr als 63 binäre Größenordnungen auseinander. Dann wird die Zahl der Iterationen ermittelt, indem der Quotient aus Dividend und Divisor ermittelt wird. Und genau hier liegt auch der Unterschied zwischen FPREM und FPREM1:

  Während FPREM durch einfaches Abschneiden des Nachkommateils (`Integer*:=Truncate(Dividend/Divisor)`) die Zahl der Iterationen bestimmt, rundet FPREM1 zur nächsten Integer auf oder ab (`Integer*:=RoundNearest(Dividend/Divisor)`). FPREM realisiert somit ein Verhalten, das im 8087 und 80287 implementiert wurde, bevor der Standard IEEE 745 ins Leben gerufen worden war. Daher sollte FPREM nur verwendet werden, wenn Kompatibilität zum 80287/8087 erforderlich ist. In jedem anderen Fall ist FPREM1 der Vorzug zu geben, der den IEEE Standard 745 implementiert!



- Die Werte von Dividend und Divisor liegen mehr als 63 binäre Größenordnungen auseinander. Dann wird zunächst der Dividend durch Division mit einem aus einer implementationsabhängigen Kostanten gebildeten Wert auf ein Intervall reduziert, das sicherstellt, dass nicht mehr als die in der implementationsabhängigen Konstanten enthaltene Anzahl Iterationen erfolgen können (32 bis 63). Mit diesem Wert wird N analog zum ersten Fall bestimmt, nur dass die Integerbildung hier in jedem Fall durch Abschneiden des Nachkommateils des Quotienten erfolgt. Das Verfahren stellt sicher, dass der partial remainder implementationsabhängig um mindestens 32, maximal aber um 63 Größenordnungen gegenüber dem Dividenden reduziert wird.

Schließlich wird der Rest gebildet, indem vom Dividenden N-mal der Divisor abgezogen wird.



Das aber bedeutet, dass das Ergebnis immer korrekt ist, weil kein Auf- bzw. Abrunden bei einer Division erforderlich ist. Somit kann keine precise exception #P auftreten!



Aufgrund der Art, wie die Anzahl der Iterationen bestimmt werden, ergeben sich einige Unterschiede im Ergebnis zwischen FPREM und FPREM1, sobald eine vollständige Restbildung erfolgen konnte. Da FPREM bei der Berechnung der Iterationszahl grundsätzlich den Nachkommaanteil abschneidet, ist die Zahl durchgeführter Iterationen immer kleiner, als theoretisch (mit Nachkommateil) benötigt würde, um Null zu erreichen. Somit bleibt immer ein Rest, der das gleiche Vorzeichen wie der Dividend hat. Ferner ist der Betrag des Restes immer kleiner als der Divisor. FPREM1 dagegen rundet die Anzahl der Iterationen. Das bedeutet, dass entweder weniger oder mehr Iterationen durchgeführt werden, als theoretisch (mit Nachkommateil) benötigt würde, um Null zu erreichen. Somit kann das Vorzeichen des Restes das Gleiche sein wie das des Dividenden (Abrundung der Iterationszahl), es kann jedoch auch entgegengesetzt sein (Aufrundung des Dividenden). Dadurch verschiebt sich auch die Größenordnung des vollständigen Restes: Sie liegt nun im Intervall] –0.5 · Divisor; +0.5 · Divisor [.

Operanden FPREM/FPREM1 haben nur implizite Operanden. In ST(0) muss der Dividend stehen. Es ist somit erstes Quell- und Zielregister der Opera-

tion. In ST(1) steht als zweiter Quelloperand der Divisor. FPREM/FPREM1 haben somit folgende Struktur:

```
FPREM
FPREM1
```

Bei den folgenden Betrachtungen wird davon ausgegangen, dass die Ergebnisse nicht zu einem Über- oder Unterlauf führen und somit entsprechende Exceptions nicht ausgelöst werden. In Tabelle 1.23 sind dann die Ergebnisse dargestellt, die nach FPREM/FPREM1 mit unterschiedlichen Kombinationen von Werten für den Dividenden (erster Quelloperand) und den Divisor (zweiter Quelloperand) auftreten können. FPU-Exceptions der Klasse invalid arithmetic operand (#IA) werden ausgelöst, wenn der Dividend eine Infinite ist oder beide Quelloperanden Null (unabhängig vom Vorzeichen) enthalten. Ist nur der Divisor Null, wird die zero devide exception #Z ausgelöst. Sobald eine NaN involviert ist, ist auch das Ergebnis eine NaN.

Zweiter Quelloperand (Divisor) \ erster Quelloperand (Dividend, ST(0))	-∞	-real	-0	+0	+real	+∞	NaN
-∞	#IA	ST(0)	-0	+0	ST(0)	#IA	NaN
-real	#IA	±real, -0	-0	+0	±real, +0	#IA	NaN
-0	#IA	#Z	#IA	#IA	#Z	#IA	NaN
+0	#IA	#Z	#IA	#IA	#Z	#IA	NaN
+real	#IA	±real, -0	-0	+0	±real, +0	#IA	NaN
+∞	#IA	ST(0)	-0	+0	ST(0)	#IA	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.23: Ergebnisse der Befehle FPREM und FPREM1 mit unterschiedlichen Werten für Divisor und Dividenden

Bitte beachten Sie, dass die grau unterlegten Ergebnisse für den Befehl FPREM1 gelten. Für FPREM ist das Vorzeichen der resultierenden Realzahl immer gleich dem des Dividenden.



Condition Code

Bei einer Stack-Unter-/Überlauf-Exception (#IS) ist C0 gelöscht, wenn ein Stack-Unterlauf erfolgte.

Die wesentlichste Aufgabe des Condition Code aber ist, zu signalisieren, ob die Restbildung vollständig erfolgen konnte oder nicht. Dies übernimmt das Flag C2. Ist es gelöscht, so erfolgte eine vollständige Restbildung, da die Größenordnungen von Dividend und Divisor sich nicht um mehr als 63 unterschieden haben. Konnte dagegen nur ein partial remainder gebildet werden, ist C2 gesetzt. In diesem Fall ist zu

prüfen, inwieweit ein erneutes Aufrufen von FPREM/FPREM1 sinnvoll sein könnte.



Wenn man etwas nachdenkt, wird man feststellen, dass die Berechnung der Anzahl erforderlicher Subtraktionen nach `N:=Integer*(Dividend/Divisor)` (s.o.) im Falle vollständiger Restbildung den Vorkommateil des Quotienten erzeugt. Obwohl dieser Wert während der Operation erzeugt und benutzt wird, wird er für das Ergebnis dennoch verworfen: Als Ergebnis wird der Divisionsrest zurückgegeben, also `Ergebnis = Dividend – N · Divisor`. Doch das Verwerfen von N erfolgt nicht vollständig! C3, C1 und C0 enthalten die niedrigerwertigen Bits 2 bis 0 von N, also die »untersten« drei Bits der Anzahl erforderlicher Iterationen, und zwar in der Beziehung C3 = Bit_2; C1 = Bit_1; C0 = Bit_0. Dies ist hilfreich, da diese Condition-Code-Bits als 3-Bit-Zahl interpretiert werden können, die zusammengenommen einen Wert zwischen 0 und 7 darstellen. Auf diese Weise stellt der Condition Code das Ergebnis einer Integer-Division des ganzzahligen Divisionsergebnisses mit dem Divisor 8 dar: `CC := N MOD 8`.



Wozu das Ganze? Divisionsergebnis einer Division – und noch nicht einmal vollständig? Was sollten die untersten drei Bits des ganzzahligen Quotienten der Division `Dividend / Divisor` schon aussagen? Denken Sie bitte auch hier an das Einsatzgebiet der Befehle FPREM/FPREM1: periodische Funktionen. Die untersten drei Bits des Quotienten stellen den Oktanden im Einheitskreis dar, auf den sich der Divisionsrest bezieht. Dieser Wert ist bei verschiedenen Berechnungen periodischer Funktionen wichtig, unter anderem bei der Berechnung des Tangens eines Wertes.

Top of Stack

FPREM bzw. FPREM1 sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Dividend im TOS wird durch das Ergebnis überschrieben.

1.2.2 Trigonometrische Operationen

Eine Teilmenge der transzendenten Operationen, die die FPU beherrscht, sind die trigonometrischen Operationen sin, cos und tan. Befehle für die »inversen« trigonometrischen Funktionen gibt es nicht. Allerdings gibt es analog zu FPTAN den »inversen« Befehl FPATAN, mit dem nicht nur der arctan, sondern auch der arcsin und arccos berechnet werden kann.



In der Evolution der Intel-Prozessoren wurden auch die implementierten trigonometrischen Funktionen geändert. So begann der 8087 mit dem einzigen trigonometrischen Befehl FPTAN, der den »partiellen« Tangens berechnete. Aus ihm wurden dann sowohl der tangens als auch cotangens und gar sinus und cosinus berechnet. Mit dem 80387 kamen die Befehle FSIN, FCOS und FSINCOS hinzu, sodass nicht mehr der Weg über FPTAN gegangen werden musste. FPTAN bewegte sich somit immer mehr in Richtung Berechnung des Tangens. Allerdings wurde aus Kompatibilitätsgründen der Name »partieller Tangens« beibehalten. Beachten Sie daher, dass sich die Implementationen der trigonometrischen Befehle prozessorabhängig verändern können! Einzelheiten hierzu entnehmen Sie bitte dem Kapitel »Historie« ab Seite 874.

FSIN
FCOS

FSIN bildet den Sinus, FCOS den Cosinus des Wertes, der im Operanden übergeben wird. Der Wert des Operanden muss dabei als Radiant im Intervall $]-2^{63};+2^{63}[$ liegen. Tut er das nicht, bleibt der TOS unverändert.

Operanden

FSIN bzw. FCOS haben nur einen impliziten Operanden. In ST(0) muss das Argument stehen. Es ist somit Quell- und Zielregister der Operation. FSIN/FCOS haben damit folgende Befehlsstruktur:

```
FSIN
FCOS
```

In Tabelle 1.24 sind die Ergebnisse dargestellt, die nach FSIN bzw. FCOS mit unterschiedlichen Argumenten auftreten können. FPU-Exceptions der Klasse invalid arithmetic operand (#IA) werden ausgelöst, wenn der Dividend eine Infinite ist. Sobald eine NaN involviert ist, ist auch das Ergebnis eine NaN.

Quelloperand	-∞	-real	-0	+0	+real	+∞	NaN
FSIN	#IA	[-1;+1]	-0	+0	[-1;+1]	#IA	NaN
FCOS	#IA	[-1;+1]	+1	+1	[-1;+1]	#IA	NaN

Tabelle 1.24: Ergebnisse der Befehle FSIN und FCOS mit unterschiedlichen Argumenten

Condition Code

Liegt das Argument innerhalb des Intervalls $]-2^{63};+2^{63}[$ können der Sinus bzw. Cosinus gebildet werden. C2 wird dann gelöscht und signalisiert ein korrektes Ergebnis. Andernfalls wird C2 gesetzt und der TOS bleibt unverändert.



FSIN und FCOS lösen keine Exception aus, wenn das Argument außerhalb des gültigen Bereiches liegt. Es liegt in der Verantwortung des Programmierers, C2 zu prüfen.



Das Argument kann ggf. mittels FPREM/FPREM1 auf das gewünschte Intervall reduziert werden!

Top of Stack FSIN bzw. FCOS sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Das Argument im TOS wird durch das Ergebnis überschrieben.

FSINCOS Sollen sowohl der Sinus als auch der Cosinus berechnet werden, so kann FSINCOS verwendet werden. Dieser Befehl arbeitet schneller als die konsekutive Ausführung von FSIN und FCOS und spart dabei ansonsten erforderliche Befehle zur Stackverwaltung (PUSHen des Stack) ein.

Operanden Wie FSIN und FCOS hat auch FSINCOS nur einen impliziten Operanden, den TOS. Dort liegt das Argument, mit dem der Sinus gebildet wird. Somit lässt sich FSINCOS wie folgt verwenden:

```
FSINCOS
```

Zu den möglichen Ergebnissen in Abhängigkeit des eingesetzten Arguments siehe Tabelle 1.24 auf Seite 219.

FSINCOS legt den berechneten Sinus im TOS ab. Dann dekrementiert es den stack pointer im TOS-Feld des status registers, um Platz für den Cosinus zu schaffen. In den neuen TOS wird dann der berechnete Cosinus abgelegt.



Zur Bildung des Tangens muss nun lediglich ST(1) durch ST(0) dividiert werden. Werden Sinus und Cosinus nicht weiter benötigt, kann dies unter POPen des Stack erfolgen, um den durch FSINCOS belegten zusätzlichen Platz wieder frei zu machen. Da zur Tangens-Bildung der Quotient aus Sinus und Cosinus gebildet wird, kann hierzu der Befehl FDIVRP eingesetzt werden.

Auch der Cotangens kann so gebildet werden. Da es sich hierbei um den Quotienten aus Cosinus und Sinus handelt, erfolgt das am besten durch FDIVP.

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Überlauf (C1 = 1) oder ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten, so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0).

C2 zeigt an, ob das Argument im gültigen Bereich gelegen hat (C2 = 1) oder nicht. Ist es nicht gesetzt, so hat kein Stack-PUSH stattgefunden und im TOS liegt noch das Argument.

C0 und C3 sind undefiniert.

Top of Stack

FSINCOS führt einen Stack-PUSH durch: Der Inhalt des TOS-Feldes im status register der FPU wird dekrementiert, sodass der TOS zu ST(1) wird. Dies ist jedoch nur möglich, wenn das alte ST(7), das neuer TOS wird, als »empty« markiert ist. Andernfalls wird eine stack overflow exception (#IS) ausgelöst.

FPTAN

FPTAN bildet den Tangens des Wertes, der im Operanden übergeben wird. Der Wert des Operanden muss dabei als Radiant im Intervall $]-2^{63};+2^{63}[$ liegen. Tut er das nicht, unterbleibt die Bildung des Tangens.

Der Name »partial tangens«, der hinter dem Mnemonic steht (vgl. »FPTAN« auf Seite 874 im Kapitel »Historie«), ist historisch bedingt. In Wirklichkeit liefert seit dem 80387 der Befehl FPTAN den »echten« Tangens zurück, wenn auch aus Kompatibilitätsgründen an etwas ungewöhnlicher Stelle im ST(1), nicht – wie FSIN oder FCOS – im TOS!

Bitte beachten Sie Implementationsunterschiede des Befehls bei verschiedenen Prozessoren (vgl. »FPTAN« auf Seite 887).



Operanden

FPTAN hat nur einen impliziten Operanden. In ST(0) muss das Argument stehen. Es ist somit Quellregister der Operation. FPTAN hat daher folgende Befehlsstruktur:

```
FPTAN
```

Während der Operation wird das Argument im TOS durch seinen Tangens ersetzt und anschließend die Konstante 1.0 auf den Stack gePUSHt, sodass der eigentliche Tangens nun in ST(1) steht.

In Tabelle 1.25 sind die Ergebnisse dargestellt, die nach FPTAN mit unterschiedlichen Argumenten auftreten können. FPU-Exceptions der Klasse invalid arithmetic operand (#IA) werden ausgelöst, wenn der Dividend eine Infinite ist. Sobald eine NaN involviert ist, ist auch das Ergebnis eine NaN.

Quelloperand	-∞	-real	-0	+0	+real	+∞	NaN
ST(1)	#IA	±real	-0	+0	±real	#IA	NaN
ST(0)		1.0	1.0	1.0	1.0		1.0

Tabelle 1.25: Ergebnisse des Befehls FPTAN mit unterschiedlichen Argumenten



Mathematisch gesehen müsste FPTAN den Wert +∞/-∞ zurückgeben, wenn ein Argument von $\pi/2$ (= 90°) / - $\pi/2$ (=270°) übergeben wird. Das ist aber nicht der Fall! Es wird zwar eine große Zahl berechnet (bei mir: $\pm 1.633 \cdot 10^{16}$), die jedoch nicht so groß ist, dass man sie als ±∞ interpretieren könnte. Berücksichtigen Sie dies bitte, wenn Sie das Ergebnis von FPTAN interpretieren wollen/müssen. So führt z.B. die Prüfung darauf, ob im ST(1) eine Infinite verzeichnet ist, grundsätzlich zum Ergebnis: Nein!

Condition Code Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Überlauf (C1 = 1) oder ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten, so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0).

C2 zeigt an, ob das Argument im gültigen Bereich gelegen hat (C2 = 1) oder nicht. Ist es nicht gesetzt, so hat kein Stack-PUSH stattgefunden und im TOS liegt noch das Argument.

C0 und C3 sind undefiniert.

Top of Stack FPTAN führt einen Stack-PUSH durch: Der Inhalt des TOS-Feldes im status register der FPU wird dekrementiert, sodass der TOS zu ST(1) wird. Dies ist jedoch nur möglich, wenn das alte ST(7), das neuer TOS wird, als »empty« markiert ist. Andernfalls wird eine stack overflow exception (#IS) ausgelöst.

FPATAN FPATAN ist das »Gegenstück« zu FPTAN. Es bildet aus den Werten in ST(1) und ST(0) den Arcus Tangens, also den zum Verhältnis der Gegen- und Ankathete gehörigen Winkel als Radiant. Im Gegensatz zu FPTAN spielt hierbei auch bei modernen Prozessoren der Begriff »partiell« eine wesentliche Rolle. Anders als bei Bildung des Tangens erwartet jeder NPX bzw. jede FPU tatsächlich die Werte für die Gegenkathete in ST(1), also den Y-Achsenabschnitt des Punktes, und den Wert für die Ankathete, also den X-Achsenabschnitt des Punktes in ST(0). Selbstverständlich arbeitet FPATAN auch dann korrekt, wenn in ST(1) und ST(0) die durch FPTAN gebildeten Pseudo-Gegen- und Ankatheten stehen.

Beim Arcus Sinus ist aber nicht die Ankathete bekannt, sondern die Hypotenuse. Analog ist bei Arcus Cosinus nicht die Gegenkathete bekannt, sondern ebenfalls die Hypotenuse. Es kann aber nur der Arcus Tangens gebildet werden, bei dem Gegen- und Ankathete bekannt sein müssen, weshalb die trigonometrischen Umkehrfunktionen als Funktion des Arcus Tangens ausgedrückt werden müssen. Es gelten folgende Beziehungen:

$asin(x) = atan(x / \sqrt{(1 - x^2)})$
$acos(x) = atan(\sqrt{(1 - x^2)} / x)$
$acot(x) = atan(1 / x)$
$asec(x) = atan(\sqrt{(x^2 - 1)})$
$acsc(x) = atan(1 / \sqrt{(x^2 - 1)})$

Mit diesen Voraussetzungen und korrekten Inhalten von ST(1) und ST(0) kann somit mittels des FPATAN-Befehls jede andere Arcus-Funktion berechnet werden.



Bitte beachten Sie Implementationsunterschiede des Befehls bei verschiedenen Prozessoren. So gibt es für FPUs/NPXe ab dem 80387 keinerlei Beschränkungen für die Inhalte der Operanden, während beim 8087 und 80287 folgende Beziehung eingehalten werden musste:

$0 \leq | ST(1) | \leq | ST(0) | < +\infty$

Operanden

FPATAN hat zwei implizite Operanden. In ST(1) steht der Wert der Gegenkathete, in ST(0) der Wert der Ankathete. Selbstverständlich ist auch möglich, in ST(1) den Quotienten aus Gegen- und Ankathete zu übergeben und in ST(0) dann die Konstante 1.0 (vgl. Ergebnis des FPTAN-Befehls). FPATAN hat daher folgende Befehlsstruktur:

```
FPATAN
```

Nach der Berechnung wird das Argument in ST(1) durch den Arcus Tangens ersetzt und ST(0) als »empty« markiert. Anschließend erfolgt ein POPpen des Stack, sodass der Arcus Tangens schließlich im TOS zurückgegeben wird.

In Tabelle 1.26 sind die Ergebnisse dargestellt, die nach FPATAN mit unterschiedlichen Kombinationen von Werten für die Gegen- und Ankathete auftreten können. Exceptions werden in keinem Fall ausgelöst.

ST(1) \ ST(0)	-∞	-real	-0	+0	+real	+∞	NaN
-∞	-¾ π	-½ π	-½ π	-½ π	-½ π	-¼π	NaN
-real	- π	[-π;-½ π]	-½ π	-½ π	[-½ π; -0]	-0	NaN
-0	- π	-π	-π	-0	-0	-0	NaN
+0	+ π	+π	+π	+0	+0	+0	NaN
+real	+ π	[+π;+½ π]	+½ π	+½ π	[+½ π; +0]	+0	NaN
+∞	+¾π	+½ π	+½ π	+½ π	+½ π	+¼π	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.26: Ergebnisse des Befehls FPATAN mit unterschiedlichen Argumenten



Bitte beachten Sie, dass in Tabelle 1.26 die Ergebnisse in den Fällen, in denen eine Division zweier Infiniter oder zweier Nullen durcheinander erfolgt, das Ergebnis nicht durch eine »echte« Division gewonnen wird, die zu Exceptions führen müsste. Vielmehr wird ein spezieller Algorithmus verwendet, der die Ergebnisse korrekt berechnet.

Condition Code Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Überlauf (C1 = 1) oder ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten, so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0). C0, C2 und C3 sind undefiniert.

Top of Stack FPATAN führt einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert wird, bevor das POPpen erfolgt.

1.2.3 Andere transzendente Operationen

Weitere transzendente Operationen sind die logarithmische und die Exponentialfunktion.

FYL2X
FYL2XP1 FYL2X, *compute* $y \cdot log_2(x)$, bildet zunächst einmal den logarithmus dualis (Logarithmus zur Basis 2, ld(x)) des Arguments x. Soll nur dieser duale Logarithmus gebildet werden, ist als Faktor y der Wert 1.0 zu übergeben.



Y kann jedoch auch andere Werte annehmen, was insbesondere deshalb von Bedeutung ist, da dadurch auch Logarithmen zu anderen Basen gebildet werden können, wenn man den Logarithmus zu einer Basis (hier: a = 2) bilden kann:

$log_b(x) = (1 / log_a(b)) \cdot log_a(x)$

Das bedeutet konkret, dass als Skalierungsfaktor *y* des Befehls FYL2X der Kehrwert des dualen Logarithmus (*a = 2*) der gewünschten Basis angegeben werden kann, um den Logarithmus zu der gewünschten Basis zu erhalten. Diese Konstanten gibt es bereits vorprogrammiert für die Basen 10 (FLDL2T; *load logarithmus dualis of ten*) und e (FLDL2E; *load logarithmus dualis of e*), sodass der dekadische und natürliche Logarithmus schnell gebildet werden können. Unschön ist dabei lediglich, dass zunächst der Kehrwert dieser Konstanten gebildet werden muss. Doch Hilfe ist da: Setzt man in obiger Formel *x = a*, so erhält man

$log_b(a) = (1 / log_a(b)) \cdot log_a(a) = (1 / log_a(b))$

und somit die allgemeine Formel

$log_b(x) = log_b(a) \cdot log_a(x)$

bzw. für a = 2 und damit dem logarithmus dualis als »Grundlage«

$log_b(x) = log_b(2) \cdot ld(x)$

Für die Basen 10 und e sind auch hier die Konstanten einfach verfügbar: Sie können mittels der Ladebefehle FLDLG2, *load logarithmus decalis of two*, und FLDLN2, *load logarithmus naturalis of two*, geladen werden.

Mit diesen Konstanten ist einfach der dekadische Logarithmus ($\log_{10}(x) \equiv \lg(x)$) bzw. der natürliche Logarithmus ($\log_e(x) \equiv \ln(x)$) berechenbar, während für die allgemeine Form tatsächlich erst der Kehrwert des Ergebnisses der Funktion FYL2X mit der gewünschten Basis als Argument und eine Skalierungsfaktor 1.0 gebildet werden muss:

$lg(x) = [FLDLG2] \cdot ld\ (x)$
$ln(x) = [FLDLN2] \cdot ld\ (x)$
$log_b(x) = (1 / FYL2X(b, 1.0)) \cdot ld\ (x)$



Auf der beiliegenden CD-ROM ist eine Unit realisiert, die die Berechnung des logarithmus dualis, logarithmus naturalis, logarithmus decalis, logarithmus hexadecimalis sowie des allgemeinen Logarithmus zu einer beliebigen Basis vorstellt.



FYL2XP1, *compute* $y \cdot log_2(x+1)$ ist eine Variante von FYL2X mit eingeschränktem Wertebereich. So muss das Argument x im Intervall $-(1 - \sqrt{2} / 2) \leq x \leq +(1 - \sqrt{2} / 2)$ liegen. Dieser Wertebereich liegt nahe bei 0 (und entspräche somit einem Wert sehr nach bei +1 für FYL2X!) und liefert mit FYL2XP1 sehr viel genauere Ergebnisse, als sie FYL2X in der Nähe des Wertes +1 generieren könnte. FYL2XP1 wird daher häufig bei Zinseszins- und Rentenberechnungen eingesetzt.

Operanden FYL2X und FYL2XP1 haben zwei implizite Operanden. In ST(1) steht ein Skalierungsfaktor (y), in ST(0) der zu logarithmierende Wert (x). FYL2X und FYL2XP1 haben daher folgende Befehlsstruktur:

```
FYL2X
FYL2XP1
```

Nach der Logarithmierung wird das Argument in ST(1) durch den skalierten Logarithmus ersetzt und ST(0) als »empty« markiert. Anschließend erfolgt ein POPpen des Stack, sodass das Ergebnis schließlich im TOS zurückgegeben wird.

In Tabelle 1.27 sind die Ergebnisse dargestellt, die nach FYL2X mit unterschiedlichen Kombinationen von Werten für Argument und Skalierungsfaktor auftreten können. Die Funktion liefert im in Tabelle 1.27 grau unterlegten Bereich umso ungenauere Ergebnisse, je näher das Argument der Funktion bei +1 liegt. Für diesen Fall gibt es daher die Funktion FYL2XP1, deren mögliche Ergebnisse in Tabelle 1.27 dargestellt sind.

Exceptions des Typs invalid arithmetic operand exception (#IA) werden bei FYL2X ausgelöst, wenn das Argument kleiner Null ist, gleich Null oder »gleich« +∞ und mit Null skaliert werden soll oder 1 ist und mit ±∞ skaliert werden soll. Eine zero divide exception #Z wird ausgelöst, wenn das Argument des Logarithmus Null ist und mit einer Realzahl skaliert werden soll. Wie gewohnt liefert die Funktion eine NaN zurück, sobald eine NaN übergeben wird.

ST(1) \ ST(0)	-∞	-real	±0	0<real<1	+1	1<real<∞	+∞	NaN
-∞	#IA	#IA	+∞	+∞	#IA	-∞	-∞	NaN
-real	#IA	#IA	#Z	+real	-0	-real	-∞	NaN
-0	#IA	#IA	#IA	+0	-0	-0	#IA	NaN
+0	#IA	#IA	#IA	-0	+0	+0	#IA	NaN
+real	#IA	#IA	#Z	-real	+0	+real	+∞	NaN
+∞	#IA	#IA	-∞	-∞	#IA	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.27: Ergebnisse des Befehls FYL2X mit unterschiedlichen Argumenten

Ob FYL2XP1 eine Exception auslöst oder nicht, wenn ein Argument übergeben wird, das außerhalb des Intervalls [-(1 - √2/2) ; +(1 - √2/2)] liegt, ist implementationsabhängig. Daher sollte man sich nicht auf die Auslösung von Exceptions verlassen und diese für die Entwicklung allgemein lauffähiger Programme heranziehen. Die betroffenen Operand-Kombinationen sind in Tabelle 1.28 mit einem Fragezeichen versehen. Ansonsten gibt es nur dann Grund für Exceptions, wenn das Argument der Logarithmusbildung Null ist und mit ±∞ skaliert werden soll. In diesem Fall erfolgt die Auslösung einer invalid arithmetic operand exception #IA.

ST(1) \ ST(0)	-∞≤real<-C	-C≤real<-0	-0	+0	+0<real≤+C	+C<real≤+∞	NaN
-∞	?	+∞	#IA	#IA	-∞	?	NaN
-real	?	+real	+0	-0	-real	?	NaN
-0	?	+0	+0	-0	-0	?	NaN
+0	?	-0	-0	+0	+0	?	NaN
+real	?	-real	-0	+0	real	?	NaN
+∞	?	-∞	#IA	#IA	+∞	?	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.28: Ergebnisse des Befehls FYL2XP1 mit unterschiedlichen Argumenten. C = (1 - √2/2)

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Überlauf (C1 = 1) oder ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten, so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0). C0, C2 und C3 sind undefiniert.

Top of Stack FYL2X und FYL2XP1 führen einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert wird, bevor das POPpen erfolgt.

F2XM1 F2XM1, *compute* $2^x - 1$, ist die Umkehrfunktion zu FYL2XP1. Sie erhebt das Argument zur Potenz von 2 und subtrahiert die Konstante 1.0. Einen Skalierungsfaktor wie bei FYL2XP1 gibt es jedoch nicht! Das Argument muss im Intervall [-1.0;+1.0] liegen, somit kann F2XM1 nicht dazu verwendet werden, allgemein Potenzen von 2 zu berechnen.



Leider gibt es die zu FYL2X umgekehrte Funktion *F2X* nicht, die das könnte! Sie muss von Hand programmiert werden. Hierzu kann neben F2XM1 auch der weiter unten besprochene Befehl FSCALE herangezogen werden, der ebenfalls Argumente zur Potenz von 2 erhebt, allerdings nur ganzzahlige. Nach

$$2^x = 2^{INT+FRC} = 2^{FRC} \cdot 2^{INT}$$

kann mit F2XM1 der Nachkommateil (*FRC*) potenziert und das Ergebnis als Skalierungsfaktor für FSCALE verwendet werden, das seinerseits den Vorkommateil (*INT*) potenziert.



Analog FYL2X/FYL2XP1 können auch Potenzen zu anderen Basen generiert werden. Hierbei macht man sich die Beziehung

$$y^x = 2^{x \cdot ld(y)}$$

zu Nutze, wobei ld(y) der logarithmus dualis der gewünschten Basis ist: $ld(y) = log_2(y)$. Gibt man nun als Faktor die Konstanten an, die mittels FLDL2T ($\log_2(10)$) oder FLDL2E ($\log_2(e)$) geladen werden können, ist eine Erhebung zur Potenz der Basis 10 bzw. e berechenbar, oder sogar, wenn man vorher mittels FYL2X den ld(y) bildet, zu jeder beliebigen Basis *y*:

$10^x = 2^{x \cdot [FLDL2T]}$
$e^x = 2^{x \cdot [FLDL2E]}$
$y^x = 2^{x \cdot FYL2X(Y, 1.0)}$



Auf der beiliegenden CD-ROM ist eine Unit realisiert, die die Berechnung der Potenzen zur Basis 2, e, 10 und 16 sowie zu einer beliebigen Basis vorstellt, indem sie FSCALE in Verbindung mit F2XM1 benutzt.

Operanden

Quell- und Zieloperand sind bei F2XM1 Befehl impliziert: Es handelt sich beide Male um den TOS. Das bedeutet, der Wert im TOS wird durch das Ergebnis der Funktion ersetzt. Damit gibt es nur eine Möglichkeit, den Befehl aufzurufen:

```
F2XM1
```

Tabelle 1.29 zeigt den Inhalt des TOS vor und nach der Operation. Analog zu FYL2XP1 ist das Verhalten des Prozessors implementationsabhängig, wenn Werte außerhalb des gültigen Wertebereiches übergeben werden. Dies ist in der Tabelle mit einem Fragezeichen markiert. Das bedeutet, dass Exceptions ausgelöst werden können, aber nicht müssen. Software, die auf unterschiedlichen Prozessortypen lauffähig sein soll, sollte daher nicht davon ausgehen, dass Exceptions ausgelöst werden.

Quelloperand	$-\infty \le real < -1$	$-1 \le real < -0$	-0	+0	$+0 < real \le +1$	$+1 < real \le +\infty$	NaN
Zieloperand	?	[-0.5 ; -0[	-0	+0	]+0 ; +1.0]	?	NaN

Tabelle 1.29: Ergebnisse des Befehls F2XM1 mit unterschiedlichen Argumenten

Condition Code

Bitte beachten Sie, dass F2XM1 den um den Faktor 1.0 erniedrigten Wert für die Potenz zurückgibt! Um den korrekten Wert der Potenzierung $y = 2^x$ zu erhalten, muss daher noch der Faktor 1.0 zum Ergebnis addiert werden!



Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Überlauf (C1 = 1) oder ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten, so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0). C0, C2 und C3 sind undefiniert.

Top of Stack

F2XM1 ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

1.2.4 Operationen zum Datenvergleich und Datenklassifizierung

FCOM
FCOMP
FCOMPP

FUCOM
FUCOMP
FUCOMPP

FCOM, *compare floating point values*, ist das FPU-Pendant zu CMP (vgl. Seite 69). FCOM vergleicht zwei Fließkommazahlen und legt das Ergebnis des Vergleichs im condition code ab. Wie CMP bewerkstelligt FCOM das, indem vom ersten Operanden der zweite Operand abgezogen wird und die Flags des condition codes anhand des Ergebnisses dieser Subtraktion gesetzt wird. Das temporäre Ergebnis wird dann verworfen, sodass die Inhalte von erstem und zweitem Operanden unverändert bleiben.

FCOMP, *compare floating point value and pop stack*, und FCOMPP, *compare floating point value and pop stack twice*, sind zwei Abarten von FCOM, die entweder einmal (FCOMP) oder zweimal (FCOMPP) den FPU-Stack POPpen, nachdem der Vergleich durchgeführt wurde.

FUCOM, *unordered compare floating point values*, FUCOMP, *unordered compare floating point values and pop stack*, und FUCOMPP, *unordered compare floating point values and pop stack twice*, sind siamesische Zwillinge zu FCOM/FCOMP/FCOMPP, die sich nur in ihrem Verhalten unterscheiden, wenn einer oder beide Operanden bestimmte Formen von NaNs sind.

Operanden

FCOM/FUCOM und FCOMP/FUCOMP sind in einer operandenlosen und in einer Ein-Operanden-Form realisiert, FCOMPP/FUCOMPP kommen nur in einer operandenlosen Form vor. In jedem Fall ist der erste Operand impliziert, es handelt sich um den TOS. In der operandenlosen Form ist auch der zweite Operand impliziert, es ist ST(1). Andernfalls kann der zweite Operand ein FPU-Register oder eine Speicherstelle sein. In diesem Fall können nur SingleReals oder DoubleReals im Speicher adressiert werden. Somit sind folgende Befehlsformen möglich:

- Operandenlose Form (XXX steht für FCOM, FCOMP, FCOMPP, FUCOM, FUCOMP oder FUCOMPP)
  ```
  XXX(≡ XXX ST(0), ST(1));
  ```
- Ein-Operanden Form, Operand ist ein FPU-Register (XXX steht für FCOM, FCOMP, FUCOM oder FUCOMP)
  ```
  XXX ST(i)(≡ XXX ST(0), ST(i))
  ```
- Ein-Operanden Form, Operand ist eine Speicherstelle (XXX steht für FCOM, FCOMP, FUCOM oder FUCOMP):
  ```
  XXX Mem32(≡ XXX ST(0), Mem32)
  XXX Mem64(≡ XXX ST(0), Mem64)
  ```

Anders als bei CMP sind jedoch einige Spezialfälle zu berücksichtigen, die bei einem Vergleich von Fließkommazahlen auftreten können. Das liegt daran, dass bei der Codierung von Fließkommazahlen auch »außergewöhnliche« Zahlen wie Infinite, Denormale und NaNs auftreten können (vgl. »Codierung von Fließkommazahlen« auf Seite 788). So prüfen FCOM/FCOMP/FCOMPP und ihre »ungeordneten« Zwillinge vor dem Vergleich, ob einer der Operanden eine NaN oder ein nicht unterstütztes Format (z.B. Pseudo-Zahlen) enthält. Ist das der Fall, wird entweder eine invalid arithmetic operand exception (#IA) ausgelöst oder, wenn diese markiert ist, der condition code auf »unordered« gesetzt. Infinite werden als »echte« Zahlen angesehen, die kleiner als die kleinste oder größer als die größte Finite sind und somit einen echten Vergleich erlauben.

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Unterlauf (C1 = 0) stattgefunden hat.

Die weiteren Flags des condition code werden anhand des Ergebnisses des Vergleichs wie folgt gesetzt: Ist der erste Operand kleiner als der zweite, so ist die temporäre Differenz negativ und C0, da mit dem carry flag im EFlags-Register kommuniziert, ist gesetzt, andernfalls gelöscht. Sind beide Operanden gleich groß, so wird das mit dem zero flag kommunizierende C3 gesetzt.

Sind die Operanden nicht vergleichbar, weil einer (oder beide) eine NaN oder ein nicht unterstütztes Format darstellen, wird entweder eine invalid arithmetic operand exception #IA ausgelöst oder, wenn sie maskiert ist, der condition code wie folgt kodiert: Neben dem mit dem parity flag kommunizierenden C2 werden auch C3 und C0 gesetzt. Tabelle 1.30 zeigt die Zusammenhänge.



An dieser Stelle machen sich auch die Unterschiede zwischen FCOMx und FUCOMx bemerkbar: FCOMx interpretieren alle NaNs als NaN, unterscheiden nicht zwischen sNaNs und qNaNs (vgl. »Codierung von Fließkommazahlen« auf Seite 788) und lösen somit eine #IA aus bzw. setzen im maskierten Fall den condition code. FUCOMx dagegen unterscheiden sehr wohl zwischen qNaNs und sNaNs. Quiet NaNs machen hier ihrem Namen Ehre und sind »still«, was bedeutet, dass sie in keinem Fall eine Exception auslösen, sondern nur den condition code auf »unordered« setzen. Signalling NaNs dagegen lösen eine #IA aus, es sei denn, diese Exception-Art ist maskiert. Dann wird auch in diesem Fall nur der condition code auf »unordered« gesetzt.

Bedingung	C3	C2	C0
erster Operand (TOS) > zweiter Operand	0	0	0
erster Operand (TOS) < zweiter Operand	0	0	1
erster Operand (TOS) = zweiter Operand	1	0	0
Operanden nicht vergleichbar (»unordered«)	1	1	1

Tabelle 1.30: Stellung der Flags des Condition Code nach Vergleichen mit FCOM/FCOMP/FCOMPP



Falls einer oder beide Operanden eine Null enthalten, wird kein Unterschied beim Vorzeichen gemacht und –0 und +0 als 0 gleichgesetzt. Somit liefert z.B. der Vergleich von –0 und +0 als condition code 100_b.



Die Flags C3, C2 und C0 des condition code der FPU korrespondieren mit den Statusflags zero, parity und carry im EFlags-Register der CPU (vgl. Seite 196). Daher ist es möglich, nach einem geeigneten Transfer des condition codes in das EFlags-Register (vgl. Seiten 196, 197) auf das Ergebnis des Vergleiches mit bedingten Befehlen wie einem bedingten Jump (vgl. »Jcc« auf Seite 103) zu reagieren.

Top of Stack

FCOM/FUCOM sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FCOMP/FUCOMP führen einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert wird, bevor das POPpen erfolgt.

FCOMPP/FUCOMPP führen einen doppelten Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird um zwei inkrementiert, sodass ST(2) zum TOS wird. Hierzu werden ST(0) und ST(1) als »empty« markiert wird, bevor das POPpen erfolgt.

FICOM FICOMP

FCOM/FCOMP/FCOMPP können nur Fließkommazahlen mit einander vergleichen. Häufig ist aber auch gewünscht, eine Zahl mit einer Integer zu vergleichen. Hierzu dienen die Befehle FICOM, *compare floating point value with integer*, und FICOMP, *compare floating point value with integer and pop stack*. Sie vergleichen eine Fließkommazahl mit einer Integer.

Operanden

FICOM und FICOMP konvertieren die im Operanden übergebene Integer in das ExtendedReal-Format. Daher kommt als Operand nur der

zweite Operand in Frage, der eine Speicherstelle adressiert, die eine Integer oder eine LongInt enthält. Somit muss der zweite Operand explizit vorgegeben werden. Wie bei allen Vergleichsbefehlen der FPU ist der erste Operand implizit vorgegeben: Es ist der TOS.



Es gibt verschiedene Arten von Integer: vorzeichenbehaftete und vorzeichenlose Zahlen mit einem, zwei, vier und acht Byte Umfang (vgl. »Codierung von Integers« auf Seite 801). FICOM/FICOMP interpretieren die an der angegebenen Stelle stehende Integer immer als vorzeichenbehaftete Integer (SmallInt und LongInt). ShortInts (ein Byte Umfang) und QuadInts (8 Bytes Umfang) können nicht zum Vergleich herangezogen werden. Da die größte nutzbare Integer somit eine LongInt mit 31 Bit Genauigkeit (exklusive Vorzeichenbit!) ist und im ExtendedReal-Format die Mantisse exakt 64 Bit umfasst, ist eine vollständige und fehlerlose Konvertierung vom Integer- ins Fließkomma-Format möglich. Entsprechende Exceptions werden daher nicht ausgelöst.



FICOM/FICOMP sind Spezialfälle der »allgemeinen« Befehle FCOM/FCOMP. Sie dienen dazu, neben Fließkommazahlen auch Integer in Vergleichen zu nutzen. Da innerhalb der FPU-Register aber alle Zahlen im ExtendedReal-Format vorliegen, können FICOM/FICOMP keine FPU-Register als Quelle bzw. Ziel übergeben werden. Hierzu dienen die entsprechenden Versionen von FCOM/FCOMP.

FICOM/FICOMP können somit in folgender Form aufgerufen werden, um eine SmallInt oder LongInt für den Vergleich zu nutzen (XXX steht für FICOM bzw. FICOMP):

```
XXX Mem16    (≡ XXX ST(0), Mem16);
XXX Mem32    (≡ XXX ST(0), Mem32);
```

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Unterlauf (C1 = 0) stattgefunden hat.

Die weiteren Flags des condition code werden anhand des Ergebnisses des Vergleichs wie folgt gesetzt: Ist die Fließkommazahl kleiner als die Integer, so ist die temporäre Differenz negativ und C0, das mit dem carry flag im EFlags-Register kommuniziert, ist gesetzt, andernfalls gelöscht. Sind beide Operanden gleich groß, so wird das mit dem zero flag kommunizierende C3 gesetzt. Sind die Operanden nicht vergleichbar, weil der Quelloperand eine NaN oder ein nicht unterstütztes Format darstellt, wird neben dem mit dem parity flag kommunizierenden C2 auch C3 und C0 gesetzt. Tabelle 1.31 zeigt die Zusammenhänge.

Bedingung	C3	C2	C0
Fließkommazahl (TOS) > Integer	0	0	0
Fließkommazahl (TOS) < Integer	0	0	1
Fließkommazahl (TOS) = Integer	1	0	0
Operanden nicht vergleichbar (»unordered«)	1	1	1

Tabelle 1.31: Stellung der Flags des Condition Code nach Vergleichen mit FICOM/FICOMP



Falls einer oder beide Operanden eine Null enthalten, wird kein Unterschied beim Vorzeichen gemacht und –0 und +0 als 0 gleichgesetzt. Somit liefert z.B. der Vergleich von –0 und +0 als condition code 100_b



Die Flags C3, C2 und C0 des condition code der FPU korrespondieren mit den Statusflags zero, parity und carry im EFlags-Register der CPU (vgl. Seite 196). Daher ist es möglich, nach einem geeigneten Transfer des condition codes in das EFlags-Register (vgl. Seiten 196, 197) auf das Ergebnis des Vergleiches mit bedingten Befehlen wie einem bedingten Jump (vgl. »Jcc« auf Seite 103) zu reagieren.

Top of Stack

FICOM ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FICOMP führt einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert wird, bevor das POPpen erfolgt.

FCOMI
FCOMIP

FUCOMI
FUCOMIP

FCOM und FCOMP bzw. FUCOM und FUCOMP haben einen entscheidenden Nachteil: Wenn man auf das Ergebnis reagieren will, muss ein mehr oder weniger umständlicher Weg (vgl. Seite 196) benutzt werden, um den condition code in das EFlags-Register zu bekommen. Einfacher, eleganter und durchaus effizienter wäre es, wenn der Vergleich direkt die entsprechenden Flags im EFlags-Register setzen könnte – und den condition code ignorieren würde. Genau das machen FCOMI, *compare floating point values and set eflags register*, und FCOMIP, *compare floating point values and set eflags register and pop stack* sowie ihre »ungeordneten« Zwillinge FUCOMI, *unordered compare floating point values and set eflags register*, und FUCOMIP, *unordered compare floating point values and set eflags register and pop stack*. (Bitte frage mich niemand, warum das »I« im Mnemonic für »set eflags« steht!)

Operanden

Bei FCOMI/FUCOMI und FCOMIP/FUCOMIP ist der erste Operand explizit angegeben, aber nicht wählbar, es handelt sich um den TOS. Der zweite Operand muss ein FPU-Register sein. Somit ist folgende Befehlsform möglich (XXX steht für FCOMI, FCOMIP, FUCOMI, FUCOMIP):

```
XXX ST(0), ST(i)
```

Analog FCOM/FCOMP/FUCOM/FUCOMP sind auch hier einige Spezialfälle zu berücksichtigen, die bei einem Vergleich von Fließkommazahlen auftreten können. Das liegt daran, dass bei der Codierung von Fließkommazahlen auch »außergewöhnliche« Zahlen wie Infinite, Denormale und NaNs auftreten können (vgl. »Codierung von Fließkommazahlen« auf Seite 788). So prüfen FCOMI/FCOMIP/FUCOMI/FUCOMIP vor dem Vergleich, ob einer der Operanden eine NaN oder ein nicht unterstütztes Format (z.B. Pseudo-Zahlen) enthält. Ist das der Fall, wird entweder eine invalid arithmetic operand exception (#IA) ausgelöst oder, wenn diese markiert ist, der condition code auf »unordered« gesetzt. Infinite werden als »echte« Zahlen angesehen, die kleiner als die kleinste oder größer als die größte Finite sind und somit einen echten Vergleich erlauben.

Statusflags (!)

Das Ergebnis des Vergleichs wird durch Setzen der Flags des EFlags-Registers der CPU signalisiert. Hierbei werden nur die Flags verwendet, die mit den entsprechenden Flags des condition codes der CPU korrespondieren. Ist daher der erste Operand kleiner als der zweite, so ist die temporäre Differenz negativ und CF, das mit C0 im condition code kommuniziert, ist gesetzt, andernfalls gelöscht. Sind beide Operanden gleich groß, so wird das zero flag, das mit C3 kommuniziert, gesetzt.

Sind die Operanden nicht vergleichbar, weil einer (oder beide) eine NaN oder ein nicht unterstütztes Format darstellen, wird entweder eine invalid arithmetic operand exception #IA ausgelöst oder, wenn sie maskiert ist, der condition code wie folgt kodiert: Neben dem parity flag werden auch zero flag und carry flag gesetzt. Tabelle 1.32 zeigt die Zusammenhänge.



An dieser Stelle machen sich auch die Unterschiede zwischen FCOMIx und FUCOMIx bemerkbar: FCOMIx interpretieren alle NaNs als NaN, unterscheiden nicht zwischen sNaNs und qNaNs (vgl. »Codierung von Fließkommazahlen« auf Seite 788) und lösen somit eine #IA aus bzw. setzen im maskierten Fall den condition code. FUCOMIx dagegen unterscheiden sehr wohl zwischen qNaNs und sNaNs. Quiet NaNs machen hier ihrem Namen Ehre und sind »still«, was bedeutet, dass sie in

keinem Fall eine Exception auslösen, sondern nur den condition code auf »unordered« setzen. Signalling NaNs dagegen lösen eine #IA aus, es sei denn, diese Exception-Art ist maskiert. Dann wird auch in diesem Fall nur der condition code auf »unordered« gesetzt.

Bedingung	ZF	PF	CF
erster Operand (TOS) > zweiter Operand	0	0	0
erster Operand (TOS) < zweiter Operand	0	0	1
erster Operand (TOS) = zweiter Operand	1	0	0
Operanden nicht vergleichbar (»unordered«)	1	1	1

Tabelle 1.32: Stellung der Flags des Condition Code nach Vergleichen mit FCOMI/FCOMIP/FUCOMI/FUCOMIP



Falls einer oder beide Operanden eine Null enthalten, wird kein Unterschied beim Vorzeichen gemacht und –0 und +0 als 0 gleichgesetzt. Somit liefert z.B. der Vergleich von –0 und +0 als condition code 100_b



Da bei diesen Befehlen die Statusflags zero, parity und carry im EFlags-Register der CPU (vgl. Seite 196) direkt gesetzt werden, ist es möglich auf das Ergebnis des Vergleiches direkt mit bedingten Befehlen wie einem bedingten Jump (vgl. »Jcc« auf Seite 103) zu reagieren.

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Unterlauf (C1 = 0) stattgefunden hat. C0, C2 und C3 sind undefiniert (!!).

Top of Stack

FCOMI/FUCOMI sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FCOMIP/FUCOMIP führen einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert wird, bevor das POPpen erfolgt.

FTST

FTST, *test TOS*, ist ein verkappter FCOM-Befehl: Er vergleicht den Inhalt des TOS mit dem Wert 0.0 und prüft somit, ob der im TOS stehende Wert kleiner, gleich oder größer Null ist. Somit ist FTST nicht etwa die FPU-Version des CPU-Befehls TEST, das ja bitweise vergleicht!

Operanden

FTST hat nur implizite Operanden: der erste Operand ist der TOS und der zweite die Konstante 0.0. Somit kann FTST nur in folgender Form verwendet werden:

```
FTST   (≡ FTST ST(0), 0.0)
```

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, ob ein Stack-Unterlauf (C1 = 0) stattgefunden hat.

Die weiteren Flags des condition code werden anhand des Ergebnisses des Vergleichs wie folgt gesetzt: Ist die im TOS stehende Fließkommazahl kleiner als Null, so ist die temporäre Differenz negativ und C0, das mit dem carry flag im EFlags-Register kommuniziert, ist gesetzt, andernfalls gelöscht. Ist der TOS gleich Null, so wird das mit dem zero flag kommunizierende C3 gesetzt. Sind die Operanden nicht vergleichbar, weil der Quelloperand eine NaN oder ein nicht unterstütztes Format darstellt, wird neben dem mit dem parity flag kommunizierenden C2 auch C3 und C0 gesetzt. Tabelle 1.33 zeigt die Zusammenhänge.

Bedingung	C3	C2	C0
Fließkommazahl (TOS) > 0.0	0	0	0
Fließkommazahl (TOS) < 0.0	0	0	1
Fließkommazahl (TOS) = 0.0	1	0	0
Operanden nicht vergleichbar (»unordered«)	1	1	1

Tabelle 1.33: Stellung der Flags des Condition Code nach Vergleichen mit FTST



Falls der TOS eine Null enthält, wird kein Unterschied beim Vorzeichen gemacht und –0 und +0 als 0 gleichgesetzt. Somit liefert z.B. der Vergleich von –0 und +0 als condition code 100_b



Die Flags C3, C2 und C0 des condition code der FPU korrespondieren mit den Statusflags zero, parity und carry im EFlags-Register der CPU (vgl. Seite 196). Daher ist es möglich, nach einem geeigneten Transfer des condition codes in das EFlags-Register (vgl. Seiten 196, 197) auf das Ergebnis des Vergleiches mit bedingten Befehlen wie einem bedingten Jump (vgl. »Jcc« auf Seite 103) zu reagieren.

Top of Stack

FTST ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FXAM FXAM, *examine TOS*, klassifiziert den im Operanden übergebenen Wert und stellt somit fest, ob eine NaN, ein nicht unterstütztes Format (z.B. Pseudozahlen), eine echte Zahl etc. vorliegt.

Operanden FXAM hat nur einen impliziten Operanden: den TOS. Somit kann FXAM nur in folgender Form verwendet werden:

```
FXAM   (≡ FXAM ST(0))
```

Condition Code C1 enthält das Vorzeichen des geprüften Wertes. Es ist immer eine Kopie des MSB des TOS, selbst wenn das Register »empty« markiert ist. Die weiteren Flags des condition code werden anhand des Ergebnisses des Vergleichs gemäß Tabelle 1.34 gesetzt.

Bedingung	C3	C2	C0
nicht unterstütztes Format	0	0	0
NaN (sowohl sNaNs als auch qNaNs)	0	0	1
»echte« Zahlen (Finite)	0	1	0
±∞ (Infinite)	0	1	1
Null (±0)	1	0	0
leeres Register (»empty«)	1	0	1
Denormale	1	1	0

Tabelle 1.34: Stellung der Flags des Condition Code nach Vergleichen mit FTST



Die Flags C3, C2 und C0 des condition code der FPU korrespondieren mit den Statusflags zero, parity und carry im EFlags-Register der CPU (vgl. Seite 196). Daher ist es möglich, nach einem geeigneten Transfer des condition codes in das EFlags-Register (vgl. Seiten 196, 197) auf das Ergebnis des Vergleiches mit bedingten Befehlen wie einem bedingten Jump (vgl. »Jcc« auf Seite 103) zu reagieren.

Top of Stack FXAM ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

1.2.5 Operationen zum Datenaustausch

FLD Wie bekommt man eigentlich Daten in die FPU-Register? Die CPU verfügt über den Befehl MOV, um die CPU-Register zu bedienen. Bei FPUs und NPXen heißt der entsprechende Befehl FLD, *load floating point value*.

Damit eine Fließkommazahl geladen werden kann, wird der FPU-Stack gePUSHt, bevor der Quelloperand übernommen werden kann. Das bedeutet, dass ST(7) vor dem Laden »empty« markiert sein muss.



Bitte beachten Sie, dass nicht irgendein Register des FPU-Stacks »empty« sein muss, sondern ST(7)! Selbst wenn außer ST(7) der gesamte Stack unbenutzt sein sollte, wird eine Stack-Überlauf-Exception beim Versuch ausgelöst, einen Wert mittels FLD zu laden. Der Grund ist, dass FLD den Stack PUSHt, bevor der Wert in den TOS geladen werden kann. Somit wird ST(7) neuer TOS. Ist ST(7) daher nicht leer, muss der Ladeversuch scheitern!



FLD kann nur mit Fließkommazahlen als Operanden arbeiten. Für die anderen beiden Zahlenarten, die die FPU verarbeiten kann, Integers und BCDs, gibt es eigene Ladebefehle (vgl. FILD/FIST auf Seite 243 bzw. FBLD/FBSTP auf Seite 245).

Operanden

Als expliziten Quell-Operanden akzeptiert FLD entweder ein anderes FPU-Register (inklusive des aktuellen TOS!) oder eine Speicherstelle. An der Speicherstelle darf jedoch nur eine Fließkommazahl stehen. Impliziter Ziel-Operand ist immer der TOS!



Es gibt drei verschiedene Arten von Fließkommazahlen: SingleReals, DoubleReals und ExtendedReals (vgl. »Codierung von Fließkommazahlen« auf Seite 788). FLD konvertiert SingleReals und DoubleReals automatisch in das ExtendedReal-Format, bevor der Wert geladen wird.

FLD kann also in folgenden Befehlssequenzen aufgerufen werden:

- Quelloperand ist ein FPU-Register
  ```
  FLD ST(i)(≡ FLD ST(0), ST(i));
  ```
- Quelloperand ist eine Speicherstelle mit einer Single-, Double- oder ExtendedReal
  ```
  FLD Mem32(≡ FLD ST(0), Mem32);
  FLD Mem64(≡ FLD ST(0), Mem64);
  FLD Mem80(≡ FLD ST(0), Mem80);
  ```

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Überlauf (C1 = 1) stattgefunden hat. C0, C2 und C3 sind undefiniert.

Top of Stack

FLD führt einen Stack-PUSH durch: Der Inhalt des TOS-Feldes im status register der FPU wird dekrementiert, sodass der TOS zu ST(1) wird. Dies ist jedoch nur möglich, wenn das alte ST(7), das neuer TOS wird, als »empty« markiert ist. Andernfalls wird eine stack overflow exception (#IS) ausgelöst.

FLD1
FLDZ
FLDPI
FLDL2E
FLDL2T
FLDLG2
FLDLN2

Es gibt spezielle Ladefunktionen, die bestimmte Konstanten laden können: FLD1, *load 1.0*, und FLDZ, *load zero*, laden die Konstanten 1.0 bzw. 0.0. Eine weitere wichtige Konstante, die für viele trigonometrischen Berechnungen erforderlich ist, ist π, sodass es auch zum Laden von ihr einen Befehl gibt: FLDPI, *load* π. Vier weitere Konstanten finden vor allem bei den transzendenten Funktionen Verwendung, sodass es vier weitere Befehle gibt, die diese Konstanten laden: FLDL2E, *load logarithmus dualis of e*, FLDL2T, *load logarithmus dualis of ten*, FLDLG2, *load logarithmus decalis of two*, und FLDLN2, *load logarithmus naturalis of two*, laden den $\log_2(e)$, $\log_2(10)$, $\log_{10}(2)$ bzw. $\log_e(2)$. Analog dem allgemeinen Ladebefehl FLD PUSHen auch diese Befehle zunächst den Stack, bevor sie die jeweilige Konstante in den neuen TOS legen.



FLDL2E, FLDL2T, FLDLG2 und FLDLN2 spielen eine wesentliche Rolle bei der Berechnung von Logarithmen zur Basis 10 und e sowie zur Erhebung in die Potenzen der Basen 10 und e. (Vgl. hierzu FYL2XP1 auf Seite 224 bzw. F2XM1 auf Seite 228.)



Intern wird mit einer 66-Bit-Darstellung der Konstanten gearbeitet. Sie wird gemäß der Rundungsvorschrift im Feld rounding control des control registers auf den 64-Bit-Wert einer ExtendedReal gerundet, bevor sie im FPU-Register abgelegt wird. Da diese Rundung einen »exakten« 64-Bit-Wert liefert, wird *keine* inexact result exception #P ausgelöst!

Operanden

Der Zieloperand ist bei diesen Befehlen impliziert, es handelt sich um den TOS. Der Quelloperand ist jeweils eine Chip-intern dargestellte Konstante. Das bedeutet, der Wert im TOS nach dem PUSHen wird durch die betreffende interne Konstante ersetzt. Damit gibt es nur eine Möglichkeit, die Befehle aufzurufen:

FLD1
FLDZ
FLDPI
FLDL2E
FLDL2T
FLDLG2
FLDLN2

Condition Code

Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Überlauf (C1 = 1) stattgefunden hat. C0, C2 und C3 sind undefiniert.

Top of Stack

Alle Konstanten-Ladebefehle führen einen Stack-PUSH durch: Der Inhalt des TOS-Feldes im status register der FPU wird dekrementiert, sodass der TOS zu ST(1) wird. Dies ist jedoch nur möglich, wenn das alte ST(7), das neuer TOS wird, als »empty« markiert ist. Andernfalls wird eine stack overflow exception (#IS) ausgelöst.

FST
FSTP

FST, *store floating point value*, ist das Gegenstück zu FLD: Dieser Befehl speichert eine Fließkommazahl ab. FSTP, *store floating point value and pop*, führt die gleiche Aktion durch, POPpt aber den FPU-Stack nach dem Speichervorgang.



FST/FSTP können nur mit Fließkommazahlen als Operanden arbeiten. Für die anderen beiden Zahlenarten, die die FPU verarbeiten kann, Integers und BCDs, gibt es eigene Speicherbefehle (vgl. FIST/FISTP auf Seite 243 bzw. FBLD/FBSTP auf Seite 245).

Operanden

Als expliziten Ziel-Operanden akzeptieren FST/FSTP entweder ein anderes FPU-Register oder eine Speicherstelle. Impliziter Quell-Operand ist immer der TOS!



Es gibt drei verschiedene Arten von Fließkommazahlen: SingleReals, DoubleReals und ExtendedReals (vgl. »Codierung von Fließkommazahlen« auf Seite 788). FST/FSTP konvertieren je nach Operandengröße die FPU-intern als ExtendedReal vorliegende Fließkommazahl automatisch in ein SingleReal- oder DoubleReal-Format, bevor der Wert gespeichert wird. ExtendedReals können nur durch FSTP in den Speicher gebracht werden.

FST/FSTP können also wie folgt aufgerufen werden (XXX steht für FST bzw. FSTP):

- Zieloperand ist ein FPU-Register
  ```
  XXX ST(i) (≡ XXX ST(i), ST(0));
  ```
- Zieloperand ist eine Speicherstelle mit einer Single- oder DoubleReal
  ```
  XXX Mem32 (≡ XXX Mem32, ST(0));
  XXX Mem64 (≡ XXX Mem64, ST(0));
  ```

FSTP kann zusätzlich auch ExtendedReals in eine Speicherstelle kopieren

```
FSTP Mem80   (≡ FSTP Mem80, ST(0));
```



Die Konvertierung aus dem internen ExtendedReal-Format in das Zielformat erfolgt für die Mantisse anhand der Rundungsvorschriften, die im Feld rounding control des control register verzeichnet sind. Der Exponent wird an die Breite und Schiefe des Ziel-Formats angepasst (vgl. »Codierung von Fließkommazahlen« auf Seite 788). Sollte der zu konvertierende Wert des Exponenten zu groß ein, um im Ziel-Format dargestellt werden zu können, wird eine numeric overflow exception (#O) ausgelöst. Falls die exception unmaskiert ist, wird der Wert nicht gespeichert.

Falls eine Null (±0), Infinite (±∞) oder NaN abgespeichert werden soll, werden die »untersten« Bits der im ExtendedReal-Format vorliegenden Mantisse und Exponenten abgeschnitten. Auf diese Weise bleibt der Charakter der »Sonderzahl« auch im Zielformat erhalten.

Falls die zu speichernde Zahl denormalisiert vorliegt, wird *keine* denormal exception (#D), sondern eine numeric underflow exception (#U) ausgelöst.



FST/FSTP ist der einzige FPU-Befehl, der keine stack overflow exception #IS auslöst, sobald versucht wird, in ein nicht »empty« markiertes FPU-Register zu schreiben! Der ursprüngliche Inhalt dieses Registers wird einfach überschrieben.

Condition Code Wenn eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten, so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0). C0, C2 und C3 sind undefiniert.

Top of Stack FST ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FSTP führt einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert wird, bevor das POPpen erfolgt.

FILD
FIST
FISTP

FILD, *load integer*, ist ein spezialisierter FLD-Befehl, der als Quelloperanden Integers und keine Fließkommazahlen akzeptiert. Die zu ladende Integer wird während des Ladevorgangs in das interne Extended-Real-Format konvertiert. Analog speichern FIST, *store integer*, und FISTP, *store integer and pop stack*, eine im ExtendedReal-Format vorliegende Zahl als Integer ab. Der Unterschied zwischen FIST und FISTP ist wie bei FST/FSTP der, dass FISTP nach dem Abspeichern den FPU-Stack POPpt.

Operanden

Als expliziten Quell-Operanden akzeptiert FILD eine Speicherstelle. Impliziter Ziel-Operand ist immer der TOS!



Es gibt verschiedene Arten von Integer: vorzeichenbehaftete und vorzeichenlose Zahlen mit einem, zwei, vier und acht Byte Umfang. (vgl. »Codierung von Integers« auf Seite 801). FILD interpretiert die an der angegebenen Stelle stehende Integer immer als vorzeichenbehaftete Integer (SmallInt, LongInt und QuadInt). ShortInts (ein Byte Umfang) können nicht geladen werden. Da die größte ladbare Integer eine QuadInt mit 63 Bit Genauigkeit (exklusive Vorzeichenbit!) ist und im ExtendedReal-Format die Mantisse exakt 64 Bit umfasst, ist eine vollständige und fehlerlose Konvertierung vom Integer- ins Fließkomma-Format möglich. Entsprechende Exceptions werden daher nicht ausgelöst.

Auch FIST und FISTP akzeptieren als expliziten Ziel-Operanden eine Speicherstelle und implizieren, wie FST/FSTP, den TOS als Quell-Operanden.



Im Gegensatz zum Laden einer Integer kann es beim Abspeichern einer Zahl im ExtendedReal-Format als Integer sehr wohl zu Ausnahmesituationen kommen: Ist die zu konvertierende Zahl keine Integer (genauer: besitzt die Zahl im ExtendedReal-Format einen von Null verschiedenen Nachkommaanteil), wird sie gemäß der im Feld rounding control des control register angegebenen Rundungsmethode gerundet (vgl. »rounding control« auf Seite 193). Wenn dann diese Zahl nicht in das Format des Zieloperanden passt, eine als Integer nicht darstellbare Infinite ($\pm\infty$) oder eine NaN vorliegt, wird eine invalid arithmetic operand exception #IA ausgelöst. (Ist diese Exception-Art maskiert, wird eine infinite Integer abgespeichert. ACHTUNG! Infinite Integer sind Interpretationssache. Sie sind identisch mit dem kleinsten darstellbaren Wert der entsprechenden Integer.)

Tabelle 1.35 stellt die möglichen Ergebnisse tabellarisch dar. In der Tabelle bedeuten M: MaxInt, also im Zielformat maximal darstellbare Integer; ±integer: gemäß rounding control gerundete Integer; {0,±1}: je nach Rundungsart 0 oder ±1. Exceptions werden in Form einer invalid arithmetic operand exception nur bei Über-/Unterschreitung der maximal darstellbaren Integer oder bei Vorliegen einer NaN ausgelöst.

Quelle	[-∞;-M[	[-M;-1]	]-1; -0[	-0	+0	]+0;+1[	[+1;+M]	]>M;+∞]	NaN
Ziel	#IA	-integer	{0,-1}	0	0	{0,+1}	+integer	#IA	#IA

Tabelle 1.35: Ergebnisse nach FIST/FISTP mit unterschiedlichen Werten als Eingaben



FILD und FIST/FISTP sind Spezialfälle der »allgemeinen« Befehle FLD und FST/FSTP. Sie dienen dazu, neben Fließkommazahlen auch Integer in die FPU zu laden oder in den Speicher abzulegen. Da innerhalb der FPU-Register aber alle Zahlen im ExtendedReal-Format vorliegen, können FILD bzw. FIST/FISTP keine FPU-Register als Quelle bzw. Ziel übergeben werden. Hierzu dienen die entsprechenden Versionen von FLD bzw. FST/FSTP.

FILD kann somit in folgender Form aufgerufen werden, um eine SmallInt, LongInt oder QuadInt zu laden:

```
FILD Mem16   (≡ FILD ST(0), Mem16);
FILD Mem32   (≡ FILD ST(0), Mem32);
FILD Mem64   (≡ FILD ST(0), Mem64);
```

FIST/FISTP können zum Abspeichern einer SmallInt oder LongInt wie folgt verwendet werden (XXX steht für FIST bzw. FISTP):

```
XXX Mem16    (≡ XXX Mem16, ST(0));
XXX Mem32    (≡ XXX Mem32, ST(0));
```

FISTP kann auch in das QuadInt-Format konvertieren und das Ergebnis in eine Speicherstelle kopieren:

```
FISTP Mem64  (≡ FSTP Mem64, ST(0));
```

Condition Code

Wenn bei Ausführung von FILD eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Überlauf (C1 = 1) stattgefunden hat. C0, C2 und C3 sind undefiniert.

Wenn bei Ausführung von FIST/FISTP eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten,

so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0). C0, C2 und C3 sind undefiniert.

Top of Stack

FILD führt einen Stack-PUSH durch: Der Inhalt des TOS-Feldes im status register der FPU wird dekrementiert, sodass der TOS zu ST(1) wird. Dies ist jedoch nur möglich, wenn das alte ST(7), das neuer TOS wird, als »empty« markiert ist. Andernfalls wird eine stack overflow exception (#IS) ausgelöst.

FIST ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FISTP führt einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert wird, bevor das POPpen erfolgt.

FBLD
FBSTP

FBLD, *load BCD*, ist wie FILD ein spezialisierter FLD-Befehl, der als Quelloperanden gepackte BCDs und keine Fließkommazahlen akzeptiert. Die zu ladende BCD wird während des Ladevorgangs in das interne ExtendedReal-Format konvertiert. Analog speichert FBSTP, *store BCD and pop stack*, eine im ExtendedReal-Format vorliegende Zahl als gepackte BCD ab.

Operanden

Als expliziten Quell-Operanden akzeptiert FBLD nur eine Speicherstelle. Impliziter Ziel-Operand ist immer der TOS!



Es gibt verschiedene Arten von BCDs: CPU-BCDs und FPU-BCDs (vgl. »Codierung von Integers« auf Seite 801 und »Binary Coded Decimals« auf Seite 809). FBLD interpretiert die an der angegebenen Stelle stehende BCD immer als FPU-BCD. Da die größte ladbare BCD 18 Ziffern hat und im ExtendedReal-Format die Mantisse 64 Bit (> 18 dezimale Stellen) umfasst, ist eine vollständige und fehlerlose Konvertierung vom BCD- ins Fließkomma-Format möglich. Entsprechende Exceptions werden daher nicht ausgelöst.

Auch FBSTP akzeptiert als expliziten Ziel-Operanden nur eine Speicherstelle und impliziert, wie FSTP, den TOS als Quell-Operanden.



BCDs sind sehr selten benutzte Spezialfälle von Integer-Darstellungen. Es gibt heute nur noch sehr wenige Anlässe, BCDs zu benutzen und Zahlen in dieses Format zu konvertieren. Daher gibt es auch nur einen

Befehl, BCDs in den Speicher zu transferieren: FBSTP, das nach dem Transfer den FPU-Stack POPpt. Die von FST und FIST bekannte »P«-freie Form von FBSTP gibt es nicht!



Im Gegensatz zum Laden einer BCD kann es beim Abspeichern einer Zahl im ExtendedReal-Format als BCD und damit Spezialfall einer Integer sehr wohl zu Ausnahmesituationen kommen: Ist die zu konvertierende Zahl keine Integer (genauer: besitzt die Zahl im ExtendedReal-Format einen von Null verschiedenen Nachkommaanteil), wird sie gemäß der im Feld rounding control des control register (vgl. »rounding control« auf Seite 193) angegebenen Rundungsmethode gerundet. Wenn dann diese Zahl nicht in das Format des Zieloperanden passt, eine als Integer nicht darstellbare Infinite (±∞) oder eine NaN vorliegt, wird eine invalid arithmetic operand exception #IA ausgelöst. (Ist diese Exception-Art maskiert, wird eine infinite Integer abgespeichert. ACHTUNG! Infinite Integer sind Interpretationssache. Sie sind identisch mit dem kleinsten darstellbaren Wert der entsprechenden Integer.)

Tabelle 1.36 stellt die möglichen Ergebnisse tabellarisch dar. In der Tabelle bedeuten M: MaxInt, also im Zielformat maximal darstellbare Integer; ±integer: gemäß rounding control gerundete Integer; {0,±1}: je nach Rundungsart 0 oder ±1. Exceptions werden in Form einer invalid arithmetic operand exception nur bei Über-/Unterschreitung der maximal darstellbaren Integer oder bei Vorliegen einer NaN ausgelöst.

Quelle	[-∞;-M[	[-M;-1]	]-1; -0[	-0	+0	]+0;+1[	[+1;+M]	]>M;+∞]	NaN
Ziel	#IA	-BCD	{0,-1}	-0	+0	{0,+1}	+BCD	#IA	#IA

Tabelle 1.36: Ergebnisse nach FBSTP mit unterschiedlichen Werten als Eingaben



FBLD und FBSTP sind Spezialfälle der »allgemeinen« Befehle FLD und FST/FSTP. Sie dienen dazu, neben Fließkommazahlen auch BCDs als spezielle Integers in die FPU zu laden oder in den Speicher abzulegen. Da innerhalb der FPU-Register aber alle Zahlen im ExtendedReal-Format vorliegen, können FBLD bzw. FBSTP keine FPU-Register als Quelle bzw. Ziel übergeben werden. Hierzu dienen die entsprechenden Versionen von FLD bzw. FST/FSTP.

FBLD kann daher nur wie folgt aufgerufen werden:

```
FBLD Mem80   (≡ FBLD ST(0), Mem80);
```

FBSTP wird wie folgt verwendet:

```
FBSTP Mem80  (≡ FBSTP Mem80, ST(0));
```

Condition Code

Wenn bei Ausführung von FBLD eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Überlauf (C1 = 1) stattgefunden hat. C0, C2 und C3 sind undefiniert.

Wenn bei Ausführung von FBSTP eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Unterlauf (C1 = 0) stattgefunden hat. Ist eine inexact result exception #P aufgetreten, so zeigt es an, ob aufgerundet wurde (C1 = 1) oder nicht (C1 = 0). C0, C2 und C3 sind undefiniert.

Top of Stack

FBLD führt einen Stack-PUSH durch: Der Inhalt des TOS-Feldes im status register der FPU wird dekrementiert, sodass der TOS zu ST(1) wird. Dies ist jedoch nur möglich, wenn das alte ST(7), das neuer TOS wird, als »empty« markiert ist. Andernfalls wird eine stack overflow exception (#IS) ausgelöst.

FBSTP führt einen Stack-POP durch: Der Inhalt des TOS-Feldes im status register der FPU wird inkrementiert, sodass ST(1) zum TOS wird. Hierzu wird ST(0) als »empty« markiert, bevor das POPpen erfolgt.

FXCH

Die meisten FPU-Befehle haben in irgendeiner Weise den TOS involviert. Sei es, dass Daten nur in den TOS geladen (FLD/FILD/FBLD/FLD1/FLDZ/FLDPI etc.) oder aus ihm abgespeichert (FST/FIST/FBSTP, etc.) werden können, benutzen alle arithmetischen Befehle (Addition, Subtraktion, Division, Multiplikation etc.), alle transzendenten Befehle (Sinus, Logarithmus, etc.) und alle Konvertierungs- und Vergleichsbefehle den TOS zumindest in Form eines von mehreren Operanden.

Häufig aber sollen bestimmte Operationen auf ein anderes Register angewendet werden, ohne die Stack-Struktur durcheinander zu bringen. Da dies physikalisch nicht möglich ist, sollte es zumindest möglich sein, kurzfristig den Inhalt von TOS und einem anderen FPU-Register auszutauschen. Dies ist entweder umständlich mit Hilfe einer temporären Variablen möglich, die bei der Tauschaktion mittels der Befehle FSTP – FLD eingesetzt werden müsste ...

... oder einfacher und ohne temporäre Variable mit FXCH, *exchange register contents*. FXCH tauscht einfach den Inhalt eines FPU-Registers mit dem TOS aus.

Operanden

FXCH gibt es in zwei Varianten: Der operandenlosen Variante und der Variante mit einem Operanden. Da FXCH aber zwei Registerinhalte austauscht, ist somit zumindest ein Operand immer impliziert: der TOS. Wird der zweite Operand explizit angegeben, muss es sich um ein

FPU-Register handeln. Wird er ebenfalls impliziert, handelt es sich um ST(1).

FCXH kann also wie folgt aufgerufen werden:

- Operandenlose Variante
  `FXCH (≡ FXCH ST(0), ST(1));`
- Ein-Operanden-Variante
  `FXCH ST(i) (≡ FXCH ST(0), ST(i));`

Condition Code Wenn bei Ausführung von FXCH eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Überlauf (C1 = 1) stattgefunden hat. C0, C2 und C3 sind undefiniert.

Top of Stack FXCH ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FCMOVcc FCMOVcc, *move FPU register content on CPU condition cc*, ist in mehrfacher Hinsicht etwas Besonderes:

1. FCMOV ist ein bedingter FPU-Befehl. Bedingte Befehle sind bei der FPU sehr selten – genauer gesagt: es gibt nur diesen! –, da das Vorliegen bestimmter Bedingungen in der Regel von der CPU festgestellt und entsprechend beantwortet werden muss, z.B. durch Programmverzweigung.
2. Dieser FPU-Befehl macht seine Aktionen abhängig von Flagstellungen im EFlags-Register der CPU. Somit ist er einer der wenigen FPU-Befehle, die direkt mit der CPU kommunizieren.
3. FCMOVcc ist die FPU-Version des CPU-Befehls CMOVcc (vgl. »CMOVcc« auf Seite 117) und ergänzt dessen Möglichkeiten um Fließkommazahlen.



Wie der CMOVcc-Befehl auch, ist FCMOVcc nicht auf allen Prozessoren, selbst neueren Datums, implementiert. Ob FCMOVcc benutzt werden kann, kann mittels der feature flags des CPUID-Befehls ermittelt werden. Falls das CMOV-Flag (Bit 15 der feature flags) und das FPU-Flag (Bit 0 der feature flags) gesetzt sind, wird FCMOVcc unterstützt.



Bitte beachten Sie, dass die Bedingungen *cc*, die FCMOVcc beherrscht, nicht die gleichen sind wie bei CMOVcc! Aufgrund der Tatsache, dass bei der Benutzung von Fließkommazahlen nur die Flags zero, carry

und parity ausgewertet werden können, da nur sie im condition code der FPU eine Entsprechung besitzen (vgl. Seite 196), können zum einen nicht alle und zum anderen auch nicht die »logischen« Bedingungen und ihre Mnemonics von Seite 43 zum Einsatz kommen (vgl. Seite 197).

Aufgrund der Auswertung der in diesem Fall zuständigen Statusflags gibt es die in Tabelle 1.37 dargestellten Bedingten MOV-Befehle der FPU:

Befehl	FPU-MOV, wenn	Prüfung
FCMOVB	kleiner (below)	CF=1
FCMOVBE	kleiner oder gleich (below or equal)	CF=1 | ZF = 1
FCMOVE	gleich (equal)	ZF= 1
FCMOVNB	nicht kleiner (not below)	CF=0
FCMOVNBE	nicht kleiner oder gleich	CF=0 & ZF=0
FCMOVNE	nicht gleich (not equal)	ZF=0
FCMOVU	nicht vergleichbar (unordered)	PF=1
FCMOVNU	vergleichbar (not unordered)	PF=0

Tabelle 1.37: FCMOVcc-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags

Operanden



Bitte beachten Sie, dass die Assembler nicht wie im Fall der CPU-CMOVcc-Befehle redundante Synonyme definieren (FCMOVA = FCMOVNBE oder FCMOVNAE = FCMOVB)!

Die FPU versteht unter MOV ein *FLD ST(0), ST(i)*. Das bedeutet, der Inhalt des explizit (!) als Zieloperanden anzugebenden TOS wird mit dem Inhalt aus dem im ebenfalls explizit angegebenen Quelloperanden überschrieben. Es sind somit folgende Operandenkombinationen möglich:

```
FCMOVB   ST(0), ST(i)
FCMOVBE  ST(0), ST(i)
FCMOVE   ST(0), ST(i)
FCMOVNB  ST(0), ST(i)
FCMOVNBE ST(0), ST(i)
FCMOVNE  ST(0), ST(i)
FCMOVU   ST(0), ST(i)
FCMOVNU  ST(0), ST(i)
```



Beachten Sie hierbei, dass gemäß dem FLD-Befehl der Inhalt des TOS in jedem Fall überschrieben wird, egal, ob das Register »empty« markiert ist oder nicht. Exceptions werden hierbei nicht ausgelöst!

Condition Code Wenn bei Ausführung von FCMOVcc eine stack over-/underflow exception #IS ausgelöst wird, signalisiert C1, dass ein Stack-Unterlauf (C1 = 0) stattgefunden hat. C0, C2 und C3 sind undefiniert.

Top of Stack FCMOVcc ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

1.2.6 Operationen zur Datenkonversion

FRNDINT FRNDINT, *round to integer*, rundet den Wert im TOS auf den nächsten Integer-Wert gemäß der im Feld rounding control des control register festgelegten Rundungsart (vgl. »rounding control« auf Seite 193).



Gemäß der Intel-Definition exakter Fließkommaergebnisse ruft die Rundung zu einer Integer ein nicht exaktes Ergebnis hervor. (So merkwürdig es klingt, aber es gilt: Die Fließkommazahl ist im gewählten Zielformat nicht darstellbar, das Ergebnis also nicht exakt!) Daher wird durch FRNDINT eine inexact result exception #P ausgelöst, falls der Inhalt des Quelloperanden nicht bereits eine Integer war (und sich somit nichts geändert hat!).



Befindet sich eine Infinite im TOS, so wird der Inhalt des TOS nicht verändert. Das bedeutet, Infinite können nicht gerundet werden, weshalb in diesem Fall auch keine Exceptions ausgelöst werden. sNaNs und Denormale dagegen lösen entsprechende Exceptions (#IA bzw. #D) aus.

Operanden Quell- und Zieloperand sind bei diesem Befehl impliziert: Es handelt sich beide Male um den TOS. Das bedeutet, der Wert im TOS wird durch seinen zur nächsten Integer gerundeten Wert ersetzt. Damit gibt es nur eine Möglichkeit, den Befehl aufzurufen:

```
FRNDINT
```

Condition Code C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte. Bei einer inexact-result exception (#P) ist C1 = 0, wenn nicht aufgerundet wurde, nach Aufrundung ist C1 = 1.

Top of Stack

FRNDINT ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FXTRACT

FXTRACT, *extract exponent and significant*, trennt den Exponenten und die Mantisse (*significant*) einer Fließkommazahl. Nach der Operation findet sich im TOS die Mantisse der Fließkommazahl, in ST(1) der Exponent.



Der Exponent in ST(1) ist eine Integer, die jedoch in Fließkommadarstellung vorliegt. Sie ist der »wahre« Exponent der Fließkommazahl, nicht etwa der »Schiefe Exponent«, der für Fließkommadarstellungen verwendet wird (vgl. »»Unschärfen« und Ungenauigkeiten in diesem Buch« auf Seite 776). Der Exponent der Mantisse in ST(0) ist 0 und wird somit als Schiefer Exponent mit dem Wert $3FFF dargestellt.



FXTRACT leistet wertvolle Dienste, wenn zu verschiedenen Zwecken die Mantisse und er Exponent einer Zahl getrennt dargestellt werden sollen. Ferner ist auf diese Weise eine einfache Transformation einer Zahl in ein Zahlensystem mit einer anderen Basis möglich.

Operanden

Quell- und beide Zieloperanden sind bei diesem Befehl impliziert: Es handelt sich um den TOS bzw. den TOS und einen gePUSHten Wert. Das bedeutet, der Wert im TOS wird durch den Exponenten ersetzt und anschließend die Mantisse auf den Stack gePUSHt. Damit gibt es nur eine Möglichkeit, den Befehl aufzurufen:

```
FXTRACT
```

Condition Code

C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte, C1 = 1, wenn ein Stack-Überlauf erfolgte.

Top of Stack

FXTRACT führt einen Stack-PUSH durch: Der Inhalt des TOS-Feldes im status register der FPU wird dekrementiert, sodass der TOS zu ST(1) wird. Dies ist jedoch nur möglich, wenn das alte ST(7), das neuer TOS wird, als »empty« markiert ist. Andernfalls wird eine stack overflow exception (#IS) ausgelöst.

FSCALE

FSCALE, *scale*, berechnet das Ergebnis der Funktion $y = a \cdot 2^x$, wobei *a* ein beliebiger Skalierungsfaktor ist und *x* ein ganzzahliger Wert sein muss. FSCALE kann als Umkehrfunktion zu FXTRACT aufgefasst wer-

den, da es eine Mantisse (a) und einen Exponenten (x) zur Basis 2 zu einer vollständigen, binär codierten Fließkommazahl kombiniert.



FSCALE kann zusammen mit F2XM1 verwendet werden, um beliebige Potenzen von 2 zu berechnen. Multipliziert man den/die Exponenten vor der Potenzierung mit geeigneten Konstanten, so ist jede beliebige Potenz berechenbar (vgl. F2XM1 auf Seite 228).



Auf der beiliegenden CD-ROM ist eine Unit realisiert, die die Berechnung der Potenzen zur Basis 2, e, 10 und 16 sowie zu einer beliebigen Basis vorstellt, indem sie FSCALE in Verbindung mit F2XM1 benutzt.

Operanden

Beide Quell- und der Zieloperand sind bei diesem Befehl impliziert: Es handelt sich um den TOS und ST(1). Das bedeutet, der Wert im TOS wird durch das Ergebnis der Potenzerhebung ersetzt, ST(1) bleibt unverändert. Damit gibt es nur eine Möglichkeit, den Befehl aufzurufen:

```
FSCALE
```

Bei den folgenden Betrachtungen wird davon ausgegangen, dass die Ergebnisse nicht zu einem Über- oder Unterlauf führen und somit entsprechende Exceptions nicht ausgelöst werden. In Tabelle 1.38 sind dann die Ergebnisse dargestellt, die nach FSCALE mit unterschiedlichen Kombinationen von Werten für Mantisse (Skalierungsfaktor) und Exponent auftreten können. Exceptions werden in keinem Fall ausgelöst.

		ST(0)						
		-∞	-real	-0	+0	+real	+∞	NaN
ST(1)	-integer	-∞	-real	-0	+0	+real	+∞	NaN
	0	-∞	-real	-0	+0	+real	+∞	NaN
	+integer	-∞	-real	-0	+0	+real	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Tabelle 1.38: Ergebnisse des Befehls FSCALE mit unterschiedlichen Argumenten



Bitte beachten Sie auch, dass der Inhalt von ST(1) als Integer interpretiert wird, selbst wenn dort eine Fließkommazahl enthalten ist. In diesem Fall wird nur der Vorkommaanteil verwendet. Aus diesem Grunde gibt es auch keine Unterscheidung zwischen +0 und –0.



Falls sich in ST(1) als Exponent eine Fließkommazahl befindet, verwendet FSCALE nur den Vorkommateil (schneidet also den Nachkommateil ab und benutzt ihn in der Berechnung nicht). FSCALE ist schnell, da lediglich dieser Vorkommateil als neuer Exponent in das Exponentenfeld des TOS addiert wird, also keine »echte« Potenzierung erfolgen muss! Daher kann FSCALE auch dazu verwendet werden, Multiplikationen oder Divisionen mit ganzzahligen Vielfachen von 2 durchzuführen. Hierzu steht der Multiplikand/Dividend in ST(0), der Multiplikator/Divisor als ganzzahlige Potenz von 2 in ST(1). FSCALE ist daher bei Fließkommazahlen so etwas Ähnliches wie die Shift-Befehle bei Integers.



Aufgrund der Arbeitsweise von FSCALE bleibt die Mantisse in ST(0) in der Regel unverändert, es wird lediglich der Exponententeil um den in ST(1) stehenden Wert inkrementiert oder dekrementiert. FSCALE versucht jedoch zu Normalisieren (vgl. »»Unschärfen« und Ungenauigkeiten in diesem Buch« auf Seite 776). Daher kann sich die Mantisse vor und nach der Operation unterscheiden, wenn entweder eine denormalisierte Zahl übergeben wurde, die normalisiert werden konnte, oder eine normalisierte Zahl nun denormalisiert ist. Auch bei Überlauf oder Unterlauf kann sich die Mantisse vor und nach der Operation unterscheiden.

Condition Code

C0, C2 und C3 sind undefiniert. Im Rahmen einer stack overflow/underflow exception (#IS) ist C1 = 0, wenn ein Stack-Unterlauf erfolgte. Bei einer inexact-result exception (#P) ist C1 = 0, wenn nicht aufgerundet wurde, nach Aufrundung ist C1 = 1.

Top of Stack

FSCALE ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

1.2.7 Verwaltungsbefehle

FINIT
FNINIT

FINIT, *initialize FPU*, initialisiert die FPU und stellt alle Register auf ihre Defaultwerte zurück. Dies betrifft insbesondere das control register, dem der Wert $037F eingeschrieben wird, das status register, das vollständig gelöscht wird, und das tag register, das den Wert $FFFF erhält. Das last instruction pointer register LIP sowie das last data pointer register LDP (vgl. Seite 189) werden wie das Register Op gelöscht.

Diese Initialisierungen stellen 64-Bit-Genauigkeit (precision control, PC; vgl. Seite 192), Rundung zur nächsten Integer (rounding control, RC; vgl. Seite 193), Maskierung aller Exceptions (vgl. Seite 191), gelöschte exception flags (vgl. Seite 195) und einen TOS = 0 (vgl. Seite 198) ein und markieren alle FPU-Register »empty« (vgl. Seite 190).

Operanden



Der eigentlich in den hier zu betrachtenden Opcode übersetzte Befehl ist FNINIT. FNINIT, *initialize FPU without checking for pending unmasked FPU exceptions (»no wait«)*, führt genau die angegebenen Aktionen durch. FINIT dagegen übersetzt der Assembler in eine Befehlsfolge aus zwei Befehlen: FWAIT und FNINIT, in dieser Reihenfolge, die auch separat und nacheinander ausgeführt werden. FWAIT dient hierbei dazu, eventuell anhängige, unmaskierte Exceptions auszulösen und damit zu handeln, bevor FNINIT ausgeführt wird (vgl. »FWAIT« auf Seite 267).

Operanden

F(N)INIT hat keine Operanden. Die Befehle werden somit wie folgt benutzt:

```
FINIT
FNINIT
```

Condition Code

Die Flags C0, C1, C2 und C3 des condition code werden durch F(N)INIT gelöscht.

Top of Stack

F(N)INIT ist in der Weise Stack-neutral, dass es eine neue Umgebung und damit eine neue Stack-Konfiguration aufbaut. Nach F(N)INIT hat das Feld TOS des status register den Inhalt 000_b und zeigt damit auf R0.

FLDCW
FSTCW
FNSTCW

FLDCW, *load FPU control word*, lädt einen über den Operanden übergebenen Wert in das control register der FPU (vgl. Abbildung 1.30). Auf diese Weise können Werte für die Genauigkeit (precision control, PC; vgl. Seite 192), mit der die FPU arbeiten soll, für Rundung (rounding control, RC; vgl. Seite 193) und Maskierung der Exceptions (vgl. Seite 191) eingestellt werden.

FSTCW, *store FPU control word*, kopiert den Inhalt des control register der FPU in den Operanden. Es ist damit der Gegenspieler zu FLDCW.



Der eigentlich in den hier zu betrachtenden Opcode übersetzte Befehl ist FNSTCW. FNSTCW, *store FPU control word without checking for pending unmasked FPU exceptions (»no wait«)*, führt genau die angegebenen Aktionen durch. FSTCW dagegen übersetzt der Assembler in eine Befehlsfolge aus zwei Befehlen: FWAIT und FNSTCW, in dieser Reihenfolge, die auch separat und nacheinander ausgeführt werden. FWAIT

dient hierbei dazu, eventuell anhängige, unmaskierte Exceptions auszulösen und damit zu handeln, bevor FNSTCW ausgeführt wird (vgl. »FWAIT« auf Seite 267).

Operanden

FLDCW und F(N)STCW haben einen expliziten Operanden, der eine Speicherstelle adressiert, aus der bzw. in die das Word ausgelesen/gespeichert wird, das Operand des betreffenden Befehls ist. Sie haben daher folgende Befehlsstruktur:

```
FLDCW  Mem16
FSTCW  Mem16
FNSTCW Mem16
```

Condition Code

Die Flags C0, C1, C2 und C3 des condition codes sind nach FLDCW und F(N)STCW undefiniert.

Top of Stack

F(N)STCW und FLDCW sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FSTSW
FNSTSW

FSTSW, *store FPU status word*, speichert den Inhalt des status register (vgl. Seite 191) der FPU in einen über den Operanden übergebenen Wert. Dies ist besonders dann sinnvoll, wenn die Flagstellungen des condition code nach Vergleichen oder Prüfungen ausgewertet werden sollen (vgl. Seite 196).



Falls Sie einen eigenen exception handler programmieren wollen, ist FSTSW auch der geeignete Befehl, um die exception flags auszuwerten. Beachten Sie in diesem Fall, dass die exception flags »sticky« sind und zurückgesetzt werden müssen (vgl. »F(N)CLEX« auf Seite 256).



Beachten Sie bitte, dass es nicht analog zu FLDCW/FSTCW einen Befehl FLDSW gibt, mit dem man ein status word in das status register einlesen könnte. Dieser Befehl machte keinen Sinn, da das Register ja den aktuellen Zustand der FPU, insbesondere was Exceptions betrifft, anzeigt, ein Überschreiben mit einem vorgegebenen Wert somit keinen Sinn hätte. Der einzige vernünftige Zweck könnte darin bestehen, den TOS, warum auch immer, auf einen bestimmten Wert zu setzen. Dies kann jedoch auch über eigenständige Befehle (FINCSTP/FDECSTP) oder das Verändern des Speicherabbildes z.B. nach Auslesen mittels FSAVE und Rückspeichern durch FRSTOR erfolgen, sodass auf einen Befehl FLDSW verzichtet wurde.



Der eigentlich in den hier zu betrachtenden Opcode übersetzte Befehl ist FNSTSW. FNSTSW, *store FPU status word without checking for pending unmasked FPU exceptions (»no wait«)*, führt genau die angegebenen Aktionen durch. FSTSW dagegen übersetzt der Assembler in eine Befehlsfolge aus zwei Befehlen: FWAIT und FNSTSW, in dieser Reihenfolge, die auch separat und nacheinander ausgeführt werden. FWAIT dient hierbei dazu, eventuell anhängige, unmaskierte Exceptions auszulösen und damit zu handeln, bevor FNSTSW ausgeführt wird (vgl. »FWAIT« auf Seite 267).

Operanden F(N)STSW haben einen expliziten Operanden, der eine Speicherstelle adressiert, aus der bzw. in die das Word ausgelesen/gespeichert wird, das Operand des betreffenden Befehls ist.



Darüber hinaus ist es auch möglich, das status word direkt in das AX-Register zu schreiben. Dieser Weg wird üblicherweise benutzt, um die flags des condition codes im Rahmen bedingter Befehle auszuwerten, indem durch `FSTSW AX` und `SAHF` ein Transfer des condition codes in das EFlags-Register (vgl. Seiten 196, 197) vorgenommen wird. Dadurch kann auf die im condition code abgelegten Bedingungen mit bedingten Befehlen wie einem bedingten Jump (vgl. »Jcc« auf Seite 103) reagiert werden.

Die Befehle können daher in folgende Befehlsstruktur aufgerufen werden:

```
FSTSW  Mem16
FSTSW  AX
FNSTSW Mem16
FNSTSW AX
```

Condition Code Die Flags C0, C1, C2 und C3 des condition codes sind nach F(N)STSW undefiniert.

Top of Stack F(N)STSW sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FCLEX
FNCLEX Die Flags des status word, die aufgetretene exceptions signalisieren, sind »sticky«. Hierunter versteht man, dass sie solange gesetzt bleiben, bis sie explizit gelöscht werden. Dies übernimmt FCLEX, clear exception flags. Dieser Befehl löscht die Flags PE, UE, OE, ZE, DE und IE so-

wie das exception summary flag ES, das stack fault flag SF und das busy flag B im status register der FPU (vgl. Seite 191).



Der eigentlich in den hier zu betrachtenden Opcode übersetzte Befehl ist FNCLEX. FNCLEX, *clear FPU exceptions without checking for pending unmasked FPU exceptions (»no wait«)*, führt genau die angegebenen Aktionen durch. FCLEX dagegen übersetzt der Assembler in eine Befehlsfolge aus zwei Befehlen: FWAIT und FNCLEX, in dieser Reihenfolge, die auch separat und nacheinander ausgeführt werden. FWAIT dient hierbei dazu, eventuell anhängige, unmaskierte Exceptions auszulösen und damit zu handeln, bevor FNCLEX ausgeführt wird (vgl. »FWAIT« auf Seite 267).

Operanden

F(N)CLEX haben einen impliziten Zieloperanden: das status register der FPU. Die Befehle werden somit wie folgt benutzt:

```
FCLEX
FNCLEX
```

Condition Code

Die Flags C0, C1, C2 und C3 des condition codes sind nach F(N)CLEX undefiniert.

Top of Stack

F(N)CLEX sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FINCSTP
FDECSTP

FINCSTP, *increment stack pointer*, und FDECSTP, *decrement stack pointer*, sind zwei Befehle, mit denen das TOS-Feld im status register gezielt verändert werden kann. Hierbei inkrementiert FINCSTP den TOS um 1 gemäß der Formel

$TOS_{neu} = (TOS_{alt} + 1)\ modulo\ 8$

sodass nach einem TOS-Wert von sieben zyklisch mit 0 fortgefahren wird. Analog dekrementiert FDECSTP den stack pointer um 1:

$TOS_{neu} = (TOS_{alt} - 1 + 8)\ modulo\ 8.$

Sinn beider Befehle ist, ein neues Register zum top of stack (TOS) zu machen.



Die Inhalte der FPU-Register und der Inhalt des tag register der FPU werden durch FINCSTP und FDECSTP nicht verändert!



Bitte beachten Sie, dass FINCSTP und FDECSTP *kein* PUSHen oder POPpen des Stack bewirken, sondern höchstens ein PUSHen bzw. POPpen vorbereiten! Vielmehr lassen sie lediglich die Hardwareregister zyklisch »rotieren«, sodass der TOS neu definiert wird. Unter PUSHen und POPpen des FPU-Stack versteht man im Allgemeinen ein Inkrementieren oder Dekrementieren des TOS mit nachfolgendem Lade- bzw. Speicherbefehl. Hierzu muss das »unter« dem alten TOS liegende Register in der Regel leer sein (PUSH) bzw. der alte TOS wird vor dem POPpen als leer markiert.

Operanden FINCSTP und FDECSTP haben einen impliziten Zieloperanden: das status register der FPU. Die Befehle werden somit wie folgt benutzt:

```
FINCSTP
FDECSTP
```

Condition Code Die Flags C0, C1, C2 und C3 des condition codes sind nach FINCSTP und DECSTP undefiniert.

Top of Stack FINCSTP inkrementiert das TOS-Feld, sodass der neue TOS das »nächsthöhere« Register der FPU wird. FDECSTP dekrementiert das TOS-Feld und macht auf diese Weise das »nächsttiefere« Register zum TOS.

FFREE Es gibt keine direkte Methode, auf das tag register der FPU zuzugreifen. Dies ist auch deshalb sinnvoll, das dieses Register ja Informationen über die Art der Daten in den FPU-Registern enthält. Somit wird es immer dann aktualisiert, wenn sich am Inhalt eines FPU-Registers etwas ändert.

Dennoch gibt es zumindest einen Grund, gezielt auf das Register bzw. die einzelnen Tag-Felder zugreifen zu können, die den jeweiligen FPU-Registern zugeordnet sind. Und dieser Grund ist, dass man einzelne Register gezielt löschen können sollte. Es gibt nämlich nur eine einzige Methode, den Inhalt eines FPU-Registers wieder loszuwerden: Markieren des Registers als »empty«. Hierbei wird das dazugehörige Tag-Feld mit dem Wert 11_b beschrieben (vgl. Seite 190). Im Rahmen des Stack-POPpens durch einen FPU-Befehl erfolgt das automatisch. Und mit FFREE, *free floating point register*, ist das auch manuell möglich!

Operanden FFREE erwartet als expliziten Operanden das zu leerende FPU-Register. Es wird somit wie folgt benutzt:

```
FFREE ST(i)
```

Die Inhalte der FPU-Register, der Inhalt des TOS-Feldes im status register sowie der Inhalt der nicht betroffenen Tag-Felder im tag register der FPU werden durch FFREE nicht verändert!



Condition Code

Die Flags C0, C1, C2 und C3 des condition codes sind nach FFREE undefiniert.

Top of Stack

FFREE ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FSAVE
FNSAVE
FRSTOR

FSAVE, *save FPU state*, FNSAVE, *save FPU state without checking for pending unmasked FPU exceptions (»no wait«)*, und FRSTOR, *restore FPU state*, sichern die »erweiterte« FPU-Umgebung bzw. laden eine im Speicher abgelegte, »erweiterte« FPU-Umgebung wieder zurück. Neben den Verwaltungsregistern der FPU (control register, status register, tag register, LIP, DIP und last opcode register) gehören bei F(N)SAVE/FRSTOR hierzu auch die acht FPU-Register. In »F(N)SAVE FRST« auf Seite 868 im Kapitel »FPU-, MMX- und XMM-Umgebung« ist ein Speicherabbild einer durch F(N)SAVE/FRSTOR verwalteten FPU-Umgebung dargestellt.

F(N)SAVE und FRSTOR sind Befehle einer Befehlsfamilie, die zur Sicherung bzw. Restauration der FPU-Umgebung dienen. In diese Familie gehören auch die Befehle F(N)STENV und FLDENV (vgl. Seite 264) bzw. FXSAVE und FXRSTOR (vgl. Seite 261). Zu Details der Unterschiede zwischen den Befehlen siehe »FPU-, MMX- und XMM-Umgebung« auf Seite 868.



Der eigentlich in den hier zu betrachtenden Opcode übersetzte Befehl ist FNSAVE. FNSAVE, *store FPU state without checking for pending unmasked FPU exceptions (»no wait«)*, führt genau die angegebenen Aktionen durch. FSAVE dagegen übersetzt der Assembler in eine Befehlsfolge aus zwei Befehlen: FWAIT und FNSAVE, in dieser Reihenfolge, die auch separat und nacheinander ausgeführt werden. FWAIT dient hierbei dazu, eventuell anhängige, unmaskierte Exceptions auszulösen und damit zu handeln, bevor FNSAVE ausgeführt wird (vgl. »FWAIT« auf Seite 267).



F(N)SAVE wird üblicherweise im Rahmen eines task switches ausgeführt oder wenn eine »saubere« Prozessorumgebung erforderlich ist,

z.B. im Rahmen von interrupt handler oder Unterprogrammen. FRSTOR dient dann dazu, nach der Rückkehr den originalen Zustand wiederherzustellen.



Das durch FSAVE gesicherte Speicherabbild entspricht dem FPU-Zustand nach der Abarbeitung aller ggf. anhängigen Exceptions. Werden bei der Wiederherstellung einer FPU-Umgebung durch FRSTOR exception bits im status word gesetzt (weil anhängige Exceptions vor dem Sichern mit FNSAVE nicht bearbeitet wurden), wird unmittelbar nach der Wiederherstellung eine floating point exception ausgelöst. Um dies zu vermeiden, sollten im Speicherabbild, das mit FRSTOR geladen werden soll, alle exception bits gelöscht werden!



F(N)SAVE führt nach der Sicherung der Prozessorumgebung eine Initialisierung gemäß der durch F(N)INIT vorgenommenen Aktionen durch (vgl. »FINIT/FNINIT« auf Seite 253). Dies betrifft das control register, dem der Wert $037F eingeschrieben wird, das status register, das vollständig gelöscht wird, und das tag register, das den Wert $FFFF erhält. Das last instruction pointer register LIP sowie das last data pointer register LDP (vgl. Seite 189) werden wie das Register Op gelöscht.

Diese Initialisierungen stellen 64-Bit-Genauigkeit (precision control, PC; vgl. Seite 192), Rundung zur nächsten Integer (rounding control, RC; vgl. Seite 193), Maskierung aller Exceptions (vgl. Seite 191), gelöschte exception flags (vgl. Seite 195) und einen TOS = 0 (vgl. Seite 198) ein und markieren alle FPU-Register »empty« (vgl. Seite 190).

Operanden

F(N)SAVE und FRSTOR erwarten einen expliziten Operanden, der eine Speicherstelle angibt, in die die FPU-Umgebung abgespeichert wird bzw. aus der sie geladen werden kann. Die Größe des benötigten Speicherbereiches hängt von der Betriebssystemumgebung ab: in 16-Bit-Systemen werden 94 Bytes Platz benötigt, in 32-Bit-Systemen 108 Bytes. F(N)SAVE/FRSTOR werden somit wie folgt benutzt:

```
FSAVE  Mem94; FSAVE  Mem108
FNSAVE Mem94; FNSAVE Mem108
FRSTOR Mem94; FRSTOR Mem108
```



Die Art und Anordnung der durch F(N)SAVE gespeicherten und von FRSTOR geladenen Daten und damit auch die Größe der als Operanden übergebenen Speicherstruktur entnehmen Sie bitte den Abbildungen 5.48 bis 5.51 ab Seite 871. Beachten Sie bitte, dass die Art der Daten

sowie ihre Abfolge im Speicher abhängig vom gewählten Betriebsmodus und von der aktuellen Betriebsumgebung sind.



Bitte beachten Sie auch, dass die Speicherstrukturen, die von F(N)SAVE/FRSTOR verwendet werden, nicht kompatibel sind zu denen von FXSAVE/FXRSTOR und nur eingeschränkt kompatibel zu denen von FSTENV/FLDENV!

Aus diesen Gründen ist wichtig, sicherzustellen, dass von FRSTOR nur Speicherabbilder geladen werden, die im gleichen Betriebsmodus (protected mode, virtual 8086 mode, real mode), in der gleichen Betriebsumgebung (16-Bit- oder 32-Bit-Betriebssysteme) und durch den korrespondierenden Speicherbefehl (F(N)SAVE) hergestellt wurden.

Condition Code

FRSTOR lädt die FPU- bzw. NPX-Umgebung, somit auch den Zustand der Flags C0, C2, C2 und C3 des condition codes.

F(N)SAVE sichert den condition code und löscht dann die Flags C0, C1, C2 und C3.

Top of Stack

Durch das Rückladen der FPU-Umgebung verändert FRSTOR den Wert des im TOS-Feld des status register der FPU und ist damit nicht Stack-neutral. Allerdings wird eine gesamte FPU-Umgebung geladen, sodass zu vorangehenden FPU-Umgebungen keine Beziehung mehr hergestellt werden kann und nicht von einem Stack-PUSH oder -POP gesprochen werden kann.

F(N)SAVE sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FXSAVE
FXRSTOR

FXSAVE, *save FPU, MMX and XMM state*, und FXRSTOR, *restore FPU, MMX and XMM state*, sichern analog zu F(N)SAVE die »erweiterte« FPU-, MMX- und XMM-Umgebung bzw. laden analog FSTOR eine im Speicher abgelegte, »erweiterte« FPU-Umgebung sowie die MMX- und XMM-Umgebung wieder zurück.

Ursprünglich als »schnelle« Version des F(N)SAVE/FRSTOR-Paares gedacht, wurden die Fähigkeiten von FXSAVE und FXRSTOR mit der Implementation der MMX- und XMM-Befehle erheblich erweitert: Neben den Verwaltungsregistern der FPU (control register, status register, tag register, LIP, DIP und last opcode register) gehören hierzu die acht FPU-Register, die ja identisch mit den acht MMX-Registern sind, sowie

das MXCSR-Register (vgl. Seite 361) als control und status register der XMM-Technologie und die acht XMM-Register (vgl. Seite 343). In »FXSAVE FXRSTOR« auf Seite 868 im Kapitel »FPU-, MMX- und XMM-Umgebung« ist ein Speicherabbild einer durch FXSAVE/FXRSTOR verwalteten FPU-, MMX- und XMM-Umgebung dargestellt.



FXSAVE und FXRSTOR sind Befehle einer Befehlsfamilie, die zur Sicherung bzw. Restauration der FPU-Umgebung dienen. In diese Familie gehören auch die Befehle F(N)STENV und FLDENV (vgl. Seite 264) bzw. F(N)SAVE und FRSTOR (vgl. Seite 259). Zu Details der Unterschiede zwischen den Befehlen siehe »FPU-, MMX- und XMM-Umgebung« auf Seite 868.



FXSAVE/FXRSTOR wurden bereits in Prozessoren implementiert, die nicht über eine XMM-Umgebung inklusive der XMM-Register und des MXSCR verfügen! In diesem Fall gelten die entsprechenden Bereiche des Speicherabbilds als reserviert und folgende Informationen zu einer XMM-Umgebung als nicht gültig.



Trotz des Namens ist der Befehl FXSAVE eher vergleichbar mit FNSAVE, da FXSAVE so wie FNSAVE nicht prüft, ob Exceptions anhängig sind, und diese ggf. auslöst! FXSAVE übersetzt der Assembler daher *nicht* analog FSAVE in eine Befehlsfolge mit vorangestelltem FWAIT! Ein weiterer Unterschied zu FSAVE besteht darin, dass FXSAVE *keine* Initialisierung der FPU durchführt, nachdem die Umgebung und die Register gesichert wurden! Das bedeutet, dass nach FXSAVE analog zu F(N)STENV die Inhalte aller FPU-, MMX- und XMM-Register erhalten bleiben. Ist es erforderlich, analog F(N)SAVE eine »saubere« Umgebung zu übergeben, muss dem FXSAVE-Befehl ein F(N)INIT (vgl. Seite 253) folgen!

FXSAVE wird üblicherweise im Rahmen eines task switches von interrupt handler oder Unterprogrammen ausgeführt. FXRSTOR dient dann dazu, nach der Rückkehr den originalen Zustand wiederherzustellen.



Das durch FXSAVE gesicherte Speicherabbild entspricht dem FPU-, MMX- und XMM-Zustand ggf. mit anhängigen Exceptions. Werden bei der Wiederherstellung einer FPU-Umgebung durch FXRSTOR exception bits im status word gesetzt, werden diese nach der Wiederherstellung *nicht* ausgelöst! Daher sollte jedem FXRESTOR ein FWAIT folgen, das anhängige FPU-Exceptions auslöst! Analog werden SIMD-Excep-

tions nicht ausgelöst, falls entsprechende Bits im MXCS-Register gesetzt sind. Da es keinen zu FWAIT für FPU-Befehle analogen Befehl für SIMD-Befehle gibt, wird eine anhängige SIMD-Exception erst mit der nächsten unmaskierten SIMD-Exception ausgelöst!



Falls das Flag OSFXSR im control register #4 nicht gesetzt ist, kann es implementationsabhängig möglich sein, dass die Inhalte der acht XMM-Register sowie des MXSCR nicht zurückgeschrieben werden können. Das bedeutet, dass in diesem Fall eine Sicherung/Restauration der XMM-Umgebung nicht möglich ist.

Operanden

FXSAVE und FXRSTOR erwarten einen expliziten Operanden, der eine Speicherstelle angibt, in die die FPU-, MMX- und XMM-Umgebungen abgespeichert bzw. aus der sie geladen werden können. Die Größe des benötigten Speicherbereiches hängt *nicht* wie bei F(N)STENV/FLDENV oder F(N)SAVE/FRSTOR von der Betriebssystemumgebung ab, sie beträgt immer 512 (!) Bytes. FXSAVE/FXRSTOR werden somit wie folgt benutzt:

```
FXSAVE  Mem512
FXRSTOR Mem512
```



Die Art und Anordnung der durch FXSAVE gespeicherten und von FXRSTOR geladenen Daten entnehmen Sie bitte FXSAVE FXRSTOR auf Seite 868. Beachten Sie bitte, dass die Art der Daten sowie ihre Abfolge im Speicher *nicht* abhängig vom gewählten Betriebsmodus und von der aktuellen Betriebsumgebung sind.



Bitte beachten Sie auch, dass die Speicherstrukturen, die von FXSAVE/FXRSTOR verwendet werden, nicht kompatibel sind zu denen von F(N)SAVE/FRSTOR oder von FSTENV/FLDENV! Das Laden einer mit F(N)SAVE oder F(N)STENV gesicherten Umgebung wird nicht nur aufgrund der unterschiedlichen Größe des Speicherabbildes zu Problemen führen, sondern auf jeden Fall deshalb, da die Inhalte der verschiedenen Register an unterschiedlichen Offsets liegen (vgl. hierzu »FPU-, MMX- und XMM-Umgebung« auf Seite 868).

Condition Code

FXRSTOR lädt die FPU- bzw. NPX-Umgebung, somit auch den Zustand der Flags C0, C2, C2 und C3 des condition codes.

Nach FXSAVE sind die Flags C0, C1, C2 und C3 des condition codes unverändert.

Top of Stack

Durch das Rückladen der FPU-Umgebung verändert FXRSTOR den Wert des im TOS-Feld des status register der FPU und ist damit nicht Stack-neutral. Allerdings wird eine gesamte FPU-Umgebung geladen, sodass zu vorangehenden FPU-Umgebungen keine Beziehung mehr hergestellt werden kann und nicht von einem Stack-PUSH oder -POP gesprochen werden kann.

FXSAVE ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FLDENV
FSTENV
FNSTENV

Unter der »Umgebung« der FPU versteht man die Inhalte des control registers, des status registers, des tag registers sowie der Register für den last instruction pointer (LIP), den last data pointer (LDP) und den Opcode (Op). Mittels FLDENV, *load FPU environment*, können mit einem Befehl alle diese Register mit Werten aus dem Speicher belegt werden. Umgekehrt sichert FSTENV, *store FPU environment*, die Inhalte der genannten Register mit einem Befehl in den Speicher. In »F(N)STENV FLDE« auf Seite 873 im Kapitel »FPU-, MMX- und XMM-Umgebung« ist ein Speicherabbild einer durch F(N)STENV/FLDENV verwalteten FPU-Umgebung dargestellt.



F(N)STENV/FLDENV sind Befehle einer Befehlsfamilie, die zur Sicherung bzw. Restauration der FPU-Umgebung dienen. In diese Familie gehören auch die Befehle F(N)SAVE und FRSTOR (vgl. Seite 259) bzw. FXSAVE und FXRSTOR (vgl. Seite 261). Zu Details der Unterschiede zwischen den Befehlen siehe »FPU-, MMX- und XMM-Umgebung« auf Seite 868.



Der eigentlich in den hier zu betrachtenden Opcode übersetzte Befehl FNSTENV, *store FPU environment without checking for pending unmasked FPU exceptions (»no wait«)*, führt genau die angegebenen Aktionen durch. FSTENV dagegen übersetzt der Assembler in eine Befehlsfolge aus zwei Befehlen: FWAIT und FNSTENV, in dieser Reihenfolge, die auch separat und nacheinander ausgeführt werden. FWAIT dient hierbei dazu, eventuell anhängige, unmaskierte Exceptions auszulösen und damit zu handeln, bevor FNSTENV ausgeführt wird (vgl. »FWAIT« auf Seite 267).

F(N)STENV wird üblicherweise im Rahmen von interrupt handler ausgeführt. So stellt die Sicherung der FPU-Umgebung mit anschließen-

dem Auslesen des gesicherten Speicherabbildes die einzige Möglichkeit dar, die Informationen aus dem last instruction pointer, dem last data pointer und/oder dem last opcode register zu erhalten, die im Rahmen der Exception-Verarbeitung erforderlich sind. Außerdem kann durch Löschen der Exception-Flags z.B. mittels F(N)CLEX (vgl. Seite 256) verhindert werden, dass der exception handler von weiteren Interrupts unterbrochen wird, ohne dass die Information über die gesetzten exception flags verloren geht. FLDENV dient dann vor der Rückkehr in die unterbrochene Routine dazu, wieder den originalen Zustand herzustellen.



Das durch F(N)STENV gesicherte Speicherabbild entspricht dem FPU-Zustand nach der Abarbeitung aller ggf. anhängigen Exceptions. Werden bei der Wiederherstellung einer FPU-Umgebung durch FLDENV exception bits im status word gesetzt (weil anhängige Exceptions vor dem Sichern mit FNSAVE nicht bearbeitet wurden), wird unmittelbar nach der Wiederherstellung eine floating point exception ausgelöst. Um dies zu vermeiden, sollten im Speicherabbild, das mit FLDENV geladen werden soll, alle exception bits gelöscht werden!



Im Gegensatz zu F(N)SAVE führt F(N)STENV nach der Sicherung der Prozessorumgebung *keine* Initialisierung der FPU durch. Das bedeutet, dass nach F(N)STENV die FPU-Umgebung weiterhin bestehen bleibt. Änderungen, die das Speichern der Umgebung erforderlich machen, müssen deshalb im Rahmen der Befehle, die nach F(N)STENV bearbeitet werden, durchgeführt werden.

Operanden

F(N)STENV und FLDENV erwarten einen expliziten Operanden, der eine Speicherstelle angibt, in die die FPU-Umgebung abgespeichert wird bzw. aus der sie geladen werden kann. Die Größe des benötigten Speicherbereiches hängt von der Betriebssystemumgebung ab: In 16-Bit-Systemen werden 14 Bytes Platz benötigt, in 32-Bit-Systemen 28 Bytes. F(N)STENV/FLDENV werden somit wie folgt benutzt:

```
FSTENV  Mem14; FSTENV  Mem28
FNSTENV Mem14; FNSTENV Mem28
FLDENV  Mem14; FLDENV  Mem28
```



Die Art und Anordnung der durch F(N)STENV gespeicherten und von FLDENV geladenen Daten und damit auch die Größe der als Operanden übergebenen Speicherstruktur entnehmen Sie bitte Abbildung 5.52 auf Seite 873. Beachten Sie bitte, dass die Art der Daten sowie ihre Ab-

folge im Speicher abhängig vom gewählten Betriebsmodus und von der aktuellen Betriebsumgebung sind.



Bitte beachten Sie auch, dass die Speicherstrukturen, die von F(N)STENV/FLDENV verwendet werden, nicht kompatibel sind zu denen von FXSAVE/FXRSTOR und nur eingeschränkt kompatibel zu denen von F(N)SAVE/FRSTOR!

Aus diesen Gründen ist wichtig, sicherzustellen, dass von FLDENV nur Speicherabbilder geladen werden, die im gleichen Betriebsmodus (protected mode, virtual 8086 mode, real mode), in der gleichen Betriebsumgebung (16-Bit- oder 32-Bit-Betriebssysteme) und durch den korrespondierenden Speicherbefehl (F(N)STENV) hergestellt wurden.

Condition Code FLDENV lädt die FPU- bzw. NPX-Umgebung, somit auch den Zustand der Flags C0, C2, C2 und C3 des condition codes.

Nach F(N)STENV sind die Flags C0, C1, C2 und C3 des condition codes undefiniert.

Top of Stack Durch das Rückladen der FPU-Umgebung verändert FLDENV den Wert des im TOS-Feld des status register der FPU und ist damit nicht Stack-neutral. Allerdings wird eine gesamte FPU-Umgebung geladen, sodass keine Beziehung zu vorangehende FPU-Umgebungen mehr hergestellt werden kann und nicht von einem Stack-PUSH oder -POP gesprochen werden kann.

FSTENV und FNSTENV sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FNOP Analog zu NOP und der CPU veranlasst FNOP, *no floating point operation*, die FPU, nichts zu tun! Somit dient FNOP eigentlich nur dazu, Platzhalter im Befehlsstrom der FPU zu spielen.

Operanden FNOP besitzt keinen Operanden und wird daher wie folgt aufgerufen:

```
FNOP
```

Condition Code Die Flags C0, C1, C2 und C3 des condition codes sind undefiniert.

Top of Stack FNOP ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FWAIT

FWAIT veranlasst den Prozessor (nicht die FPU!), zu prüfen, ob eine nicht behandelte, unmaskierte FPU-Exception vorliegt und diese, so vorhanden, auszulösen. FWAIT *vor* einem anderen FPU-Befehl eingestreut stellt sicher, dass der folgende Befehl nicht durch anhängige Exceptions »beeinträchtigt« wird (siehe z.B. die Befehle FINIT, FSAVE, FSTENV, FXSAVE etc.). Nach einem FPU-Befehl bewirkt FWAIT, dass alle Exceptions, die durch den vorangehenden Befehl erzeugt wurden und nun ihrer Bearbeitung harren, auch tatsächlich bearbeitet werden.

Operanden

FWAIT besitzt keinen Operanden und wird daher wie folgt aufgerufen:

```
WAIT
```

Condition Code

Die Flags C0, C1, C2 und C3 des condition codes sind undefiniert.

Top of Stack

FWAIT ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

1.2.8 Obsolete Operationen

FENI
FNENI
FDISI
FNDISI

FENI, *enable interrupts*, FNENI, *no wait enable interrupts*, und FDISI, *disable interrupts*, bzw. FNDISI, *no wait disable interrupts*, haben nur beim 8087 eine Bedeutung gehabt und sind damit schon lange obsolet. Sie dienten dazu, den Mechanismus ein- und auszuschalten, mit dem die FPU auf FPU-Exceptions aufmerksam gemacht hat (vgl. »Historie« auf Seite 874).

Operanden

Keiner der genannten Befehle besitzt einen Operanden.

Condition Code

Die Flags C0, C1, C2 und C3 des condition codes sind nach F(N)ENI und FDISI undefiniert.

Top of Stack

F(N)ENI und F(N)DISI sind Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

FSETPM

Der 8086 kannte noch keinen protected mode, weshalb es bei der Kombination 8086/8087 nicht erforderlich war, in diesen Betriebsmodus schalten zu können. Mit dem 80286 jedoch begann die Möglichkeit, den protected mode nutzen zu können. Dies hatte jedoch weit reichende Folgen, auch für den Co-Prozessor 80287, da nun eine vollkommen neue Art der Adressierung des Speichers möglich war und im protected mode auch erfolgte. Genauso, wie der Prozessor in den neuen Be-

triebsmodus umgeschaltet werden musste, musste das der 80287 nun auch. Hierzu diente der Befehl FSETPM. Dieser war jedoch schnell wieder obsolet: Ab dem 80386/80387 war jeweils der Prozessor für die Adressierung des Datenbus' zuständig, der Co-Prozessor schaute ihm dabei lediglich über die Schulter und bediente sich dann an der eingestellten Adresse. Folgerichtig musste der 80387 nicht mehr die »neue« Adressierungsart beherrschen, weshalb ein Umschalten in einen anderen Betriebsmodus und damit FSETPM überflüssig wurden. FSETPM ist somit eine »Eintagsfliege« für den 80287 (vgl. »Historie« auf Seite 874).

Operanden FSETPM hat keinen Operanden.

Condition Code Die Flags C0, C1, C2 und C3 des condition codes sind nach FSETPM undefiniert.

Top of Stack FSETPM ist Stack-neutral: Der Inhalt des TOS-Feldes im status register der FPU und somit der Zustand des FPU-Stacks ändert sich durch die Operation nicht. Der Quelloperand im TOS wird durch das Ergebnis überschrieben.

1.2.9 FPU-Exceptions

Natürlich kann es auch beim Bearbeiten von FPU-Befehlen zu Ausnahmesituationen kommen. Es gibt zwei verschiedene Gründe, weshalb die FPU Ausnahmen generiert. Demgemäß gibt es zwei Arten von Exceptions:

- numerische Ausnahmesituationen
- nicht-numerische Ausnahmesituationen

Non-numeric exceptions Nicht-numerische (nicht-arithmetische) Ausnahmesituationen werden durch FPU-Befehle erzeugt, die nicht oder nur in sehr geringem Umfang arithmetische Operationen durchführen. Dazu gehören alle Befehle, die Daten in die/aus den FPU-Register(n) laden, zwischen Registern austauschen oder Verwaltungsaufgaben übernehmen, aber auch FABS und FCHS, da sie lediglich das Vorzeichen löschen oder ändern, was nicht ernsthaft als arithmetische Instruktion angesehen werden kann. Solche nicht-arithmetischen Befehle können keine FPU-Exceptions auslösen – lässt man einmal #IS (s.u.) außen vor. Vielmehr können diese Befehle CPU-Befehle auslösen, wie sie auch im Rahmen der CPU-Arithmetik auftreten können: #GP, #PF, #AC, #SS, #NM und ggf. #UD.

Arithmetic exceptions

Daneben kann die FPU allerdings auch Ausnahmezustände bei oder nach »echten« FPU-Operationen signalisieren. Die Behandlung dieser numerischen (arithmetischen) Exceptions erfolgt aber etwas anders als bei der CPU. Dies hat zwei Gründe:

- Numerische Exceptions der FPU sind maskierbar, was bedeutet, dass die FPU das Auftreten von Ausnahmezuständen in zwei Arten handhaben kann: Durch Auslösen einer Exception oder durch eigenständige »Korrektur« des Fehlers.
- Die FPU ist der CPU untergeordnet. Sie kann also nicht, wie die CPU, selbst Exceptions auslösen und damit Exception-Handler einbinden, sondern ist darauf angewiesen, dass die CPU das für sie tut.

In jedem Fall stellt die FPU zunächst einmal fest, dass ein FPU-Ausnahmezustand aufgetreten ist. Je nach Grund der Ausnahme setzt sie daraufhin in ihrem Statusregister das der Exception zugeordnete Flag.

unmaskierte Exceptions

Als Nächstes prüft sie, ob im Kontrollregister das korrespondierende Maskenbit gesetzt ist. Ist es nicht gesetzt, so bedeutet das, dass die dazugehörige Exception unmaskiert ist und tatsächlich ausgelöst werden soll. In diesem Fall setzt sie ebenfalls das ES-Flag im Statusregister und signalisiert somit eine unbehandelte Exception. (Bei NPXen wird darüber hinaus oder stattdessen ein Signal auf eine direkt mit der CPU verbundene Leitung gelegt.)

maskierte Exceptions

Ist dagegen das Maskenbit gesetzt, die Exception also maskiert, unterbleibt das Setzen von ES. In diesem Fall behandelt die FPU die Exception selbst, indem sie z.B. einen Codewert (z.B. eine NaN) in das den Fehler verursachende Register schreibt. Dadurch wird erreicht, dass der Programmfluss von Exceptions ungestört bleibt. Sinnvoll ist dies, wenn die FPU mit ihren Maßnahmen eine ausreichende Reaktion auf den Ausnahmezustand zeigt. (So ist es sicherlich nicht gerechtfertigt, »echte« Exceptions auszulösen, nur weil die FPU mit #P z.B. anzeigt, dass der berechnete Wert 1/3 nicht vollständig in das Register eingetragen werden konnte. Das ist logisch, weil 1/3 eine unendliche Anzahl von Nachkommastellen hat und niemand ernsthaft glauben würde, die FPU könne den Wert »exakt« laden. Mathematische Auswirkungen hat das keine!)

Hybride

Da es für jede mögliche Exceptionquelle ein Maskenbit gibt, ist es möglich, »unwichtige« Exceptions wie #P zu maskieren und damit von der Auslösung einer numerischen Exception mit dem Involvieren eines Exception-Handlers auszuschließen, während »wichtige« Exceptions wie

#I, das z.B. eine Stack-Verletzung anzeigt, durch Demaskierung die Auslösung einer »echten« Exception veranlassen.



Bitte beachten Sie, dass die Flags im Statusregister »klebrig« (»sticky«) sind. Das bedeutet, sie bleiben, einmal gesetzt, solange gesetzt, bis sie explizit gelöscht werden. Das bedeutet, dass Exception-Handler diese Flags explizit löschen müssen, falls sie die entsprechenden Exceptions behandelt haben. Wirklich wichtig ist das allerdings nur bei unmaskierten Exceptions, da maskierte ja in jedem Fall beim Auftreten behandelt werden. Bei unmaskierten Exceptions aber ist es wirklich wichtig, die Flags zu löschen, da die CPU so lange einen #MF auslöst, wie ES gesetzt ist.

CPU-Exception #MF

Die CPU ihrerseits prüft nun bei jedem WAIT/FWAIT sowie beim Auftreten der meisten ESC-Sequenzen (= FPU-Befehle) oder MMX-Befehle im Befehlsstrom, ob das ES-Flag im StatusWord gesetzt ist (und/oder ggf. am FPU-Pin der CPU ein Signal anliegt) und damit das Vorliegen einer FPU-Exception signalisiert wird. Ist dies der Fall, so liegt mindestens eine unmaskierte Exception vor. Die CPU unterbricht dann die Programmausführung, löst die #MF (*math fault exception*) aus und übergibt damit die Programmausführung an einen Exception-Handler, der für die Behandlung von FPU-Exceptions und das Löschen der Flags verantwortlich ist. Dieser prüft dann anhand des StatusWords, welche FPU-Exception zu behandeln ist, und tut dies.

Exceptiontypen

Analog zu den CPU-Exceptions gibt es auch bei der FPU verschiedene Typen von Exceptions. Wie bei der Einteilung bei der CPU lassen sie sich anhand des Zeitpunktes, wann sie festgestellt werden, in zwei Gruppen einteilen:

faults

Exceptions, die ausgelöst werden, bevor die FPU-Operation stattfindet, gehören in die Gruppe der »Fehler« (*faults*).

traps

Demgegenüber stehen Exceptions, die erst nach der Operation ausgelöst werden können, da die Ursachen, die zur Auslösung führen, erst durch die Operation geschaffen werden. Solche Exceptions gehören in die Gruppe »Fallen« (*traps*).

Exceptions

Die FPU kennt sechs arithmetische Exceptions:

#I

Invalid Operation; diese Exception vom Type *trap* wird von der FPU ausgelöst, wenn einer oder beide Operanden des Befehls »ungültig« (nicht für die Operation erlaubt) sind. Dies trifft z.B. bei der Division von ∞

durch ∞ zu. #I lässt sich einteilen in zwei Arten der ungültigen Operation:

- #IA, invalid arithmetic operand;
- #IS, stack overflow or underflow;

Detailangaben hierzu finden Sie in Kapitel »FPU-Exceptions« ab Seite 529.

#D Falls die FPU einen Befehl ausführen soll, bei dem ein Operand eine denormalisierte Zahl ist, so wird *denormalized operand* Exception vom Typ *fault* ausgelöst. Zur Definition von denormalisierten Zahlen siehe Abschnitt »Codierung von Fließkommazahlen« auf Seite 788.

#Z Analog der entsprechenden CPU-Exception löst die FPU eine *Divide-by-Zero* Exception vom Typ *fault* aus, sollte der Versuch unternommen werden, eine Division durch »0« durchzuführen.

#O *Numeric Overflow*; diese Exception vom Typ *trap* wird von der FPU ausgelöst, wenn nach einer Operation das Ergebnis den Wertebereich des zugrunde liegenden Datentyps überschreitet.

#U Analog wird eine *Numeric Underflow* Exception vom Typ *trap* ausgelöst, sollte das Ergebnis den darstellbaren Wertebereich unterschreiten.

#P Eine *Inexact-Result (Precision)* Exception löst die FPU immer dann aus, wenn das Ergebnis nicht exakt in dem zugrunde liegenden Datenformat darstellbar ist. Diese Exception vom Typ *trap* wird somit beispielsweise ausgelöst bei der Darstellung des Wertes 1/3, da die Anzahl der zur Repräsentation zur Verfügung stehender Bits zu klein ist, um das Ergebnis (das auch binär nicht mit einer endlichen Zahl von Stellen darstellbar ist) exakt darstellen zu können.

Weitergehende Informationen zu Exceptions und ihrer Bearbeitung finden Sie im Kapitel »Exceptions und Interrupts« auf Seite 486.

1.2.10 FPU-Emulation

Falls das EM-Flag in Kontrollregister #0 der CPU gesetzt ist, verfügt das System weder über eine FPU, noch – bei älteren Systemen – über einen NPX (= mathematischer Co-Prozessor). In diesem Fall müssen FPU-Befehle, so sie ausgeführt werden sollen, emuliert werden. Hierzu löst die CPU bei jedem FPU-Befehl im Befehlsstrom eine #NM (*device not present exception*) aus, wodurch ein Exception-Handler aufgerufen wird, der

(hoffentlich) die FPU-Befehle emulieren kann. Solch ein Exception-Handler wird üblicherweise vom Betriebssystem zur Verfügung gestellt, sodass sich der Anwender weniger Gedanken darum machen muss.

1.3 SIMD-Operationen

Single Instruction – Multiple Data

In den heute nicht mehr aus der Welt wegzudenkenden Multimedia-Anwendungen sind zwei Datentypen besonders wichtig: Bytes, die im Bereich Video/Grafik eine dominante Rolle spielen, und Words, die im Audiobereich dominieren, aber auch bei Video/Grafik zum Einsatz kommen. Auch die moderne Kommunikation über Internet oder andere Wege arbeitet mit solchen Daten. Und nicht vergessen werden dürfen solche Gebiete wie Sprach- oder Mustererkennung und sogar die Chiffrierung/Dechiffrierung.

Die wachsende Bedeutung von Multimedia, Internet und Kommunikation hat es nötig gemacht, die Prozessoren an die speziellen Bedürfnisse der Programmierer für diese Bereiche anzupassen. Zwar können alle Prozessoren ab dem ehrwürdigen 8086 mit diesen Datentypen umgehen, sie sind als grundlegende Daten anzusehen. Doch erfordern die heutigen Anwendungen Datendurchsätze, die mit den ebenfalls grundlegenden Befehlssätzen nicht mehr zu gewährleisten sind. Zwar könnte man die Prozessor-Taktraten bis ins Gigantische steigern (was sowieso erfolgt: heutige Prozessoren sind 400 mal schneller als mein erster mit damals unglaublichen 4 MHz!) und auf diese Weise für den erforderlichen Durchsatz sorgen. Doch stehen dem eine Reihe von sehr guten Gründen entgegen: Erstens kann man das Ganze nicht bis ins Unendliche treiben, irgendwann setzt einem die Physik ein grundsätzliches Ende. Und es ist die Frage, ob der zur Realisierung notwendige Aufwand in einem vernünftigen Verhältnis zum Ergebnis steht.

Zweitens sind die Daten, die verarbeitet werden müssen, alle gleich, sodass sie eigentlich hochgradig parallelisierbar wären: Ein Bild besteht z.B. aus 1024 x 768 Bildpunkten. Ein Überblenden von einem Bild zum nächsten, z.B. beim Videoschnitt, erfordert nun 786.432 gleiche Operationen: Veränderung der Intensität/Farbe der 786.432 Bildpunkte. Könnte man diese 786.432 Operationen parallel ausführen, so könnte der gesamte Bildschirm in einem Taktzyklus verändert werden! Dazu brauchte man jedoch 786.432 konventionelle Prozessoren, da pro Prozessor pro Takt nur ein Befehl mit einem Datum abgearbeitet werden kann.

Drittens: Die Operationen, die im Bereich Multimedia/Kommunikation erforderlich sind, sind nicht einzelne, einfache Operationen wie Addition oder Multiplikation. Vielmehr sind sie vielfach recht komplex aus einfachen Operationen aufgebaut: So ist z.B. die Mischung zweier Bilder (z.B. das Einblenden eines Nachrichtensprechers vor einem im Hintergrund ablaufenden Korrespondentenfilm) ein komplexer Vorgang aus Maskenbildung (XOR), Vorbereitung des Hintergrundes (AND, NOT) und Überlagerung der beiden Teilbilder (OR). Wir werden weiter unten auf dieses Beispiel zurückkommen.

Sinnvoll wäre daher, genau diese Randbedingungen in neue Fähigkeiten der Prozessoren einzubauen. Dabei hat natürlich die Kirche im Dorf zu bleiben: Der Wunsch, 786.432 Bildpunkte gleichzeitig verarbeiten zu können, ist sicherlich nicht zu realisieren – zumindest in den Rechnern von Otto Normalverbraucher. Es würde auch kaum Sinn machen, da sich die Auflösung der Darstellung ähnlich inflationär verhält wie andere Bereiche des Computers: RAM und Taktfrequenz. Und es gibt ja schon seit langem Auflösungen jenseits der 1024 x 768 Pixel, die man durchaus als Standard ansehen kann. Wo wäre die Grenze? Es läuft also, wie praktisch immer, auf einen Kompromiss heraus.

Das Ergebnis dieser Überlegungen und Bemühungen: der so genannte SIMD-Befehlssatz der modernen Prozessoren. SIMD steht für *single instruction multiple data*. Der Name ist Programm: Diese Befehle verarbeiten mehrere gleichartige Daten in einer mehr oder weniger komplexen, speziell für Multimedia und Kommunikation entwickelten Instruktion!



Bei der Besprechung der Multimedia-Extensions werden wir chronologisch vorgehen, also mit MMX beginnend uns zu SSE2 durcharbeiten – obwohl der Pentium 4, der Grundlage dieser Auflage des Assembler-Buches ist, die Erweiterungen nach SSE2 bereits implementiert hat. Hierbei spielt weniger die evolutionäre Entwicklung der SIMD-Philosophie aufgrund der Anwendung der Erweiterungen und der daraus resultierenden Forderung nach weiteren Erweiterungen eine Rolle, die ja letztlich zu den konkurrierenden Systemen von Intel (SSE) bzw. AMD (3DNow!) geführt hat. Vielmehr gibt es einen Hauptgrund: MMX beschäftigt sich mit »einfachen« Integers, kann somit als Erweiterung des »arithmetischen« CPU-Befehlssatzes betrachtet werden, auch wenn dies, wie wir noch sehen werden, eine sehr oberflächliche Verallgemeinerung ist. SSE dagegen »erweitert« die Multimedia-Fähigkeiten des Prozessors auf Fließkommazahlen und stellt somit eine gewisse Konkurrenz zur FPU her – auch wenn durch SSE die FPU sicherlich nicht

überflüssig wird! Und dies bedeutet, dass es ähnliche Unterschiede zwischen den beiden Befehlssätzen gibt wie zwischen CPU und FPU, die ja auch jeweils in eigenen Kapiteln abgehandelt wurden. SSE2 schließlich ist lediglich ein Sahnehäubchen, das neben zusätzlichen, sinnvollen Befehlen auch eine »Vereinheitlichung« der beiden Multimedia-Welten (»Integer-Welt« und »Fließkomma-Welt«) einführt.

1.3.1 SIMD, die Erste: MMX

MMX

MMX war historisch gesehen die erste Erweiterung speziell für Multimedia-Anwendungen: *M*ulti *M*edia e*X*tensions. Es handelt sich hier um eine Klasse von Instruktionen, die mit den oben besprochenen Daten wie Bytes und Words besonders gut umgehen kann und aus den geforderten speziellen »komplexen« Instruktionen besteht. Eine Zusammenfassung der unter MMX verfügbaren Instruktionen zeigt Tabelle 5.26 auf Seite 846.

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 8-Byte-Integer, wie sie die FPU-Befehle schon längst kennen.



Wir werden jedoch von dieser Definition ein wenig abweichen. Spätestens aus der Einleitung zu diesem Kapitel wissen Sie bereits, dass es neben MMX auch noch andere Erweiterungen gibt: SSE, SSE2 und 3DNow!. Und Sie können sich denken, dass diese auch ihre eigenen Datendefinitionen besitzen. Daher nehme ich an dieser Stelle vorweg, was ich auf Seite 844 tabellarisch zusammenfassen werde, und rede hier über *ShortPacked*Bytes, *ShortPacked*Words und *ShortPacked*DWords, sofern vorzeichenlose Daten gemeint sind, sowie von *ShortPackedShortInts*, *ShortPackedSmallInts* und *ShortPackedLongInts* im Falle der vorzeichenbehafteten Pendants.

Was alle diese neuen Datentypen gemeinsam haben, sind: 64 Bits (= 8 Bytes) zur Codierung des Datums. Zu Einzelheiten über die Darstellung der neuen Datentypen siehe das Kapitel »Gepackte Daten« auf Seite 811.

Was ist nun das Besondere an diesen neuen Datentypen? Die Antwort lautet: eigentlich gar nichts! Lassen wir für den Moment das QuadWord als »ungepacktes« Datum außen vor, so verhält sich jedes Element eines »gepackten Feldes« gleich und wie die Basistypen: Alle acht Bytes eines ShortPackedByte sind echte Bytes, alle vier Worte des ShortPacked-Words sind Worte und die beiden DoubleWords des ShortPacked-DWord sind zwei echte Doppelworte. Das heißt: Sie können wie die Datenelemente, auf denen sie basieren, vorzeichenbehaftet sein oder vorzeichenlos!

Warum also »ShortPackedBytes« & Co.? Unter MMX wird, um Parallelität in der Datenverarbeitung zu erreichen, mit Datenstrukturen gearbeitet! Die Parameter der MMX-Befehle stellen nicht einzelne Daten dar, sondern Felder von Daten, eben die ShortPackedIntegers. 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 *DoubleLongInt* 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! Die unter MMX verfügbaren Datenformate sind auf Seite 844 in Tabelle 5.23 zusammengestellt.

MMX-Register

Da selbst die neuesten Prozessoren »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 den meisten Grafikprogrammen, ja selbst in FIBU- oder anderen kaufmännischen Programmen nicht oder nicht häufig der Fall – die Nutzung der FPU ist eher eine Domäne medizinischer (bildgebende Diagnoseverfahren wie MRI, *magnetic resonance imaging*, oder CT, Computer-Tomographie), wissenschaftlicher (3D-Moleküldarstellungen, Vektorrechnungen) oder ausgesuchter Graphik- (CAD, *computer aided design*) oder Spezialprogramme (z.B.

CADD, *computer aided drug design*), bei denen es auf höchste Genauigkeit ankommt. (Im kaufmännischen Bereich wird wegen der Genauigkeit mit Integers gearbeitet, die so konstruiert werden, dass keine Nachkommastellen notwendig sind. So wird z.B. jede Währung in Einheiten eines 100.000stel angegeben, eine DM also als Vielfaches von 1/1000 Pfennig. Dann reicht, einen ausreichenden Wertebereich vorausgesetzt, einfache Integer-Arithmetik vollkommen aus, ja ist in vielerlei Hinsicht sogar der Fließkommarechnung überlegen!) Auch Spiele nutzen die FPU nicht, da das Laden und Speichern der Daten und deren Verarbeitung »viel« zu lange dauern und, vom eingesetzten Datentyp aus betrachtet, nicht unbedingt die Erfordernisse von Spieleprogrammierern erfüllen würde. Summa: Auch heute noch wird die FPU sehr wenig eingesetzt. (Machen Sie sich einmal den Spaß und laden Sie ein beliebiges Anwendungs- oder Spielprogramm in einen Debugger. Suchen Sie dann einmal nach FPU-Befehlen!)

Daher hat nun Intel die (bei Mittelung über alle möglichen Anwendungsgebiete) mehr oder weniger 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 Alias-Namen der FPU-Register R0 bis R7, wie Abbildung 1.35 zeigt.

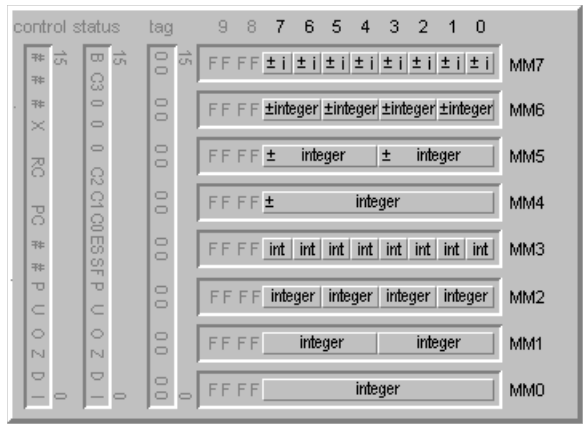


Abbildung 1.35: Die MMX-Register des Prozessors

Die Register fassen jeweils vorzeichenbehaftete oder vorzeichenlose Ganzzahlen (Integers) in den neu definierten Datenstrukturen. Unter MMX sind nur die Bytes 0 bis 7 der Register in Funktion, die beiden verbliebenen Bytes der FPU-Hardware werden auf $FF gesetzt. In der Abbildung beherbergt MM0 ein QuadWord, MM1 ein ShortPackedDoubleWord, MM2 ein ShortPackedWord und MM3 ein ShortPackedByte. Die QuadInt (MM4), ShortPackedLongInts (MM5), ShortPackedSmallInts (MM6) und ShortPackedShortInts (MM7) unterscheiden sich von den analogen Datenstrukturen vorzeichenloser Zahlen lediglich durch das im jeweiligen MSB stehende Vorzeichen.



Die MMX-Register 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 direkt ansprechen, nicht etwa um die indirekt adressierten Stackregister der FPU! 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 (z.B. `POR MM0, MM2`) statisch immer das angegebene Hardwareregister angesprochen.

Die Technik einer indirekten, dynamischen Rechenstack-Verwaltung 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, wie wir weiter oben gesehen haben, 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 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, wie schon gesagt, 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!) sowie das *status* und *control register* haben unter MMX keine Funktion.

Bleibt also festzuhalten: Unter MMX übernimmt der Prozessor die Kontrolle über die Register der FPU und nutzt sie für die parallele Bearbeitung von »gepackten« Integerzahlen. Der FPU wird so lange vorgegaukelt, dass die Inhalte in ihren Registern alles NaNs sind, *Not a Number* – keine Zahl, mit denen sie gar nichts anfangen kann. Dies erfolgt, indem die (wie gesagt als reserviert geltenden) Bits 79 bis 64 gesetzt werden. Dies ist aber die Darstellung für eine »negative qNaN« (siehe den Abschnitt »Codierung von Fließkommazahlen« auf Seite 788) und in dieser Hinsicht »geschickt eingefädelt«, da die CPU auf diese Weise etwa die Ausführung von durch den Programmierer bewusst oder unbewusst eingestreuten FPU-Befehlen verhindert bzw. entschärft: FPU-Instruktionen mit qNaNs führen selten zu Exceptions und wenn überhaupt dann nur zu »erwünschten« Veränderungen in den Registern!

Saturation Wrap-around

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 der gültige Wertebereich für das Datum über- bzw. unterschritten werden kann. 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 warp-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 *warp-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 ange-

zeigt, 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 der FPU 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 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.

MMX-Befehle

Wenn diese Technik wirklich hält, was sie verspricht, hat das Konsequenzen für die Datenbehandlung. Denn dann ist ein ShortPacked-Byte-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 der FPU erfolgt, werden acht Bytes *gleichzeitig* bearbeitet. Analoges gilt für ShortPackedWords und ShortPackedDoubleWords. 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 das SIMD Execution Model.

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 Bildschirm-

ausgaben, 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 aufwändig 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 ShortPackedByte, ShortPackedWord und ShortPackedDWord bzw. ihre vorzeichenbehafteten Zwillinge 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, sodass bei den logischen Manipulationen 64 Bits betrachtet werden, keine ShortPackedIntegers. Diese 64 Bits nennt Intel QuadWord. (Wenn Sie mich fragen, hätte man auf diese Definition auch verzichten können! Aber sei es drum!)

Arithmetische Befehle

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; *Addition of Packed Bytes*), Worte (PADDW; *Addition of Packed Words*) und Doppelworte (PADDD; *Addition of Packed DoubleWords*) unterstützt. Im *Sättigungsmodus* können vorzeichenbehaftete *Bytes* (PADDSB; *Addition and Saturation of Packed Bytes*) und *Worte* (PADDSW; *Addition and Saturation of Packed Words*) sowie deren vorzeichenlose Pendants (PADDUSB; *Addition and Saturation of Packed Unsigned Bytes* und PADDUSW; *Addition and Saturation of Packed Unsigned 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!

Operanden

Als Ziel für die Summe und Quelle des ersten Additionspartners der gepackten Addition kommt nur ein MMX-Register in Frage, während der zweite Additionspartner in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PADDB, PADDW, PADDD, PADDSB, PADDSW, PADDUSB und PADDUSW):

- Addition einer ShortPackedInteger aus einem MMX-Register zu einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, MMX`
- Addition einer ShortPackedInteger aus einer Speicherstelle zu einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, Mem64`

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; *Subtraction of Packed Bytes*), Worten (PSUBW; *Subtraction of Packed Words*) und Doppelworten (PSUBD; *Subtraction of Packed DoubleWords*). Im Sättigungsmodus können vorzeichenbehaftete Bytes (PSUBSB; *Subtraction and Saturation of Packed Bytes*) und Worte (PSUBSW; *Subtraction and Saturation of Packed Words*) sowie deren vorzeichenlose Pendants (PSUBUSB; *Subtraction and Saturation of Packed Unsigned Bytes* und PSUBUSW; *Subtraction and Saturation of Packed Unsigned 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 der FPU zugegriffen wird! Es erfolgt auch im Wrap-Around-Modus keinerlei Hinweis auf einen Über- oder Unterlauf!

Operanden

Als Ziel für die Differenz und Quelle des Subtrahenden der gepackten Subtraktion kommt nur ein MMX-Register in Frage, während der zweite Subtraktionspartner in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PSUBB, PSUBW, PSUBD, PSUBSB, PSUBSW, PSUBUSB und PSUBUSW):

- Subtraktion einer ShortPackedInteger aus einem MMX-Register von einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, MMX`
- Subtraktion einer ShortPackedInteger aus einer Speicherstelle von einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, Mem64`

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 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 enthalten dann Worte, die anderen Doppelworte. Zweitbeste Möglichkeit! Möglichkeit 2: Die beiden Zielregister fassen jeweils ein Wort des Ergebnis-DoubleWords: Das eine die niedrigerwertigen 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 niedrigerwertigen Anteile des Ergebnisses interessieren oder für die höherwertigen, verwenden Sie einen der beiden Befehle PMULLW (*Multiply Packed Word and Store Low*)

oder PMULHW (*Multiply Packed Word and Store High*), hier im Beispiel sei es *PMULLW MMx, MMy*:

```
MMx[15..00] := Temp[031..016]
MMx[31..16] := Temp[063..048]
MMx[47..32] := Temp[095..080]
MMx[63..48] := Temp[127..112]
```

Analoges gilt natürlich für PMULHW. Denkt man ein wenig genauer nach, so wird man feststellen, dass PMULHW eigentlich eine aus zwei Instruktionen zusammengesetzte Operation ist: Multiplikation zweier Worte zu einem Doppelwort mit anschließender (Integer-)Division durch $10000. Ganz analog ist dann PMULLW eine Multiplikation mit anschließender Modulo-Bildung mit $10000.

Und noch eins: Multiplikationen lassen sich nur mit Worten, nicht aber mit Bytes durchführen. Warum nicht? Ich weiß es nicht ...

Operanden

Als Ziel für das Produkt und Quelle des Multiplikanden der gepackten Multiplikation kommt nur ein MMX-Register in Frage, während der Multiplikator in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PMULLW bzw. PMULHW):

- Multiplikation einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, MMX`
- Multiplikation einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger aus einer Speicherstelle
  `XXX MMX, Mem64`

PMADDWD

PMADDWD 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 aufeinander folgende Doppelworte werden dann addiert, sodass 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]);
```

Operanden

Als Ziel für die Operation und Quelle des Multiplikanden der primären gepackten Multiplikation kommt nur ein MMX-Register in Frage, während der Multiplikator in einem MMX-Register oder an einer Speicherstelle stehen kann:

- Multiplikation mit folgender Addition einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger in einem MMX-Register
  `PMADDWD MMX, MMX`
- Multiplikation mit folgender Addition einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger aus einer Speicherstelle
  `PMADDWD MMX, Mem64`

Vergleichsbefehle

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:

- 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).

PCMPEQB
PCMPEQW
PCMPEQD
PCMPGTB
PCMPGTW
PCMPGTD

In beiden Fällen können nicht, wie im Falle der Allzweckregister-Befehle, Flags bemüht werden, um das Ergebnis des Resultates anzuzeigen! Auch lässt sich das nicht über das Statuswort der FPU realisieren. Daher muss das Ergebnis im Zieloperanden codiert werden. Führt also ein Vergleich zu einem wahren Ergebnis, sind demnach 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;
```

Operanden

Als Ziel für das Masken-Datum als Vergleichsergebnis und Quelle des Vergleichsdatums der gepackten Multiplikation kommt nur ein MMX-Register in Frage, während das zweite Vergleichsdatum in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PCMPEQB, PCMPEQW, PCMPEQD, PCMPGTB, PCMPGTW bzw. PCMPGTD):

- Vergleich einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, MMX`
- Vergleich einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger aus einer Speicherstelle
  `XXX MMX, Mem64`

Konvertierungsbefehle

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 Wort 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 Word, vorzeichenbehaftet. Aber es erfolgt eine Sättigung auf ein vorzeichenloses Byte. Warum es kein analoges *Pack with Unsigned Saturation DoubleWord to Word* gibt, entzieht sich meinem 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]);
```

Operanden

Als Ziel für das konvertierte Datum und Quelle des einen Satzes zu konvertierender Daten kommt nur ein MMX-Register in Frage, während der zweite Satz zu konvertierender Daten in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PACKSSWB, PACKSSDW bzw. PACKUSWB):

- Konvertierung einer ShortPackedInteger aus einem MMX-Register in eine »kleinere« ShortPackedInteger in einem MMX-Register
  `XXX MMX, MMX`
- Konvertierung einer ShortPackedInteger aus einer Speicherstelle in eine »kleinere« ShortPackedInteger in einem MMX-Register
  `XXX MMX, Mem64`

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 QuadWord! Es ist vielmehr das »Mischen« von zwei Doppelworten zu einem QuadWord. Der Buchstabe »L« im Mnemonic signalisiert hierbei, dass die niedrigerwertigen 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 niedrigerwertigen 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!):

```
        MV      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 ausgibt, 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
        OV      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?

Operanden Als Ziel für das konvertierte Datum und Quelle des einen Satzes zu konvertierender Daten kommt nur ein MMX-Register in Frage, während der zweite Satz zu konvertierender Daten in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKLBW, PUNPCKLWD bzw. PUNPCKLDQ):

- Konvertierung einer ShortPackedInteger aus einem MMX-Register in eine »größere« ShortPackedInteger in einem MMX-Register:
  `XXX MMX, MMX`
- Konvertierung einer ShortPackedInteger aus einer Speicherstelle in eine »größere« ShortPackedInteger in einem MMX-Register:
  `XXX MMX, Mem64`

Bit-Schiebung

Analog der Bit-Schiebebefehle der CPU in den Allzweckregistern gibt es auch entsprechende Befehle, die mit gepackten Daten arbeiten.

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 involviert, 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 genauso wenig wie im Falle der Allzweckregister, da das mit dem logischen Verschieben nach links, zumindest, was das Ergebnis betrifft, identisch ist. Insoweit nichts Neues!

Operanden

Als Ziel und Quelle des Bitfeldes kommt nur ein MMX-Register in Frage, während die Anzahl zu verschiebender Bits in einer Konstante, in einem MMX-Register oder einer Speicherstelle stehen kann (XXX steht hier für PSLLW, PSLLD, PSRLW, PSRLD, PSRAW bzw. PSRAD):

- Verschiebung der Bits einer ShortPackedInteger aus einem MMX-Register um eine als 8-Bit-Konstante übergebene Zahl Positionen
  `XXX MMX, Const8`
- Verschiebung der Bits einer ShortPackedInteger aus einem MMX-Register um eine in einem MMX-Register stehende Zahl Positionen
  `XXX MMX, MMX`
- Verschiebung der Bits einer ShortPackedInteger in einem Register um eine an einer Speicherstelle stehende Anzahl Positionen
  `XXX MMX, Mem64`



Bitte beachten Sie, dass bei diesen Bit-Schiebereien nicht das ganze Register als ein Bit-Feld betrachtet wird, sondern als acht (PackedWords) bzw. vier (PackedDoubleWords) voneinander unabhängige (Teil-)Register. Die Verschiebungen erfolgen somit für jedes skalare Datum in diesem gepackten Feld gesondert!



Falls der im zweiten Operanden übergebene Wert für die Anzahl zu verschiebender Positionen größer als 15 (PackedWords) bzw. 31 (PackedDoubleWords) ist, werden die gepackten Felder auf Null gesetzt.

PSLLQ PSRLQ

Allerdings kann man mit den Shift-Befehlen nur Worte und Doppelworte verschieben. Bytes sind ebenso wenig manipulierbar wie – QuadWords, wollte ich fast sagen. Aber Letzteres stimmt nur teilweise. Denn QuadWords können zumindest logisch nach links und rechts verschoben werden (PSLLQ und PSRLQ), was den Bit-Charakter dieses Datums unterstreicht – QuadWords sind, zumindest unter MMX, 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 QuadWord gleichzeitig verschieben. Die freigewordenen Bits werden, wie bei den anderen Befehlen auch, mit »0« aufgefüllt.

Operanden

Als Ziel und Quelle des Bitfeldes kommt nur ein MMX-Register in Frage, während die Anzahl zu verschiebender Bits in einer Konstante, in einem MMX-Register oder einer Speicherstelle stehen kann (XXX steht hier für PSLLQ bzw. PSRLQ):

- Verschiebung der Bits eines QuadWord aus einem MMX-Register um eine als 8-Bit-Konstante übergebene Zahl Positionen
  `XXX MMX, Const8`
- Verschiebung der Bits eines QuadWord aus einem MMX-Register um eine in einem MMX-Register stehende Zahl Positionen
  `XXX MMX, MMX`
- Verschiebung der Bits eines Quadword in einem Register um eine an einer Speicherstelle stehende Anzahl Positionen
  `XXX MMX, Mem64`



Bei PSLLQ und PSRLQ allerdings werden im Gegensatz zu den vorangehenden Befehlen alle 64 Bits als ein gemeinsames Feld aufgefasst, sodass diese beiden Befehle als »Ausdehnung« der Befehle SHL und SHR auf QuadWords aufgefasst werden können.

Logische Operationen

Mit den Shift-Befehlen haben wir aber den Übergang von den arithmetischen Befehlen zu den Bit-orientierten 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 Bit-orientierten 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 QuadWords arbeiten – ShortPackedIntegers spielen keine Rolle:

PAND
PANDN
POR
PXOR

Diese Bit-Manipulationsbefehle 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 QuadWord, 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	0
1	1	0
POR		
0	0	1
1	1	1
PXOR		
0	0	1
1	1	0

Tabelle 1.39: Darstellung der Bitstellungen nach den logischen Operationen PAND, PANDN, POR und PXOR

Operanden

Als Ziel und Quelle des ersten Datums kommt nur ein MMX-Register in Frage, während das zweite zu verknüpfende Datum in einem MMX-Register oder einer Speicherstelle stehen kann (XXX steht hier für PAND, PANDN, POR bzw. PXOR):

- Logische Verknüpfung eines Datums aus einem MMX-Register mit einem in einem MMX-Register stehenden Datum
  `XXX MMX, MMX`
- Logische Verknüpfung eines Datums aus einem MMX-Register mit einem in einer Speicherstelle stehenden Datum
  `XXX MMX, Mem64`



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 Operanden 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.

Ladebefehle

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 die FPU nur ExtendedReals 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 niedrigerwertige DoubleWord (ScalarDouble), 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 ist bei dem Befehl MOVQ nicht möglich, da die Allzweckregister lediglich über 32 Bits Breite verfügen und somit eine Kombination aus zwei 32-Bit-Allzweckregistern herangezogen werden müsste. Dies wird jedoch nicht unterstützt.

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 FPU-Registern, 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 Integer-Arithmetik 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. Genauso wenig erfolgt beim Kopieren eines Registerinhaltes in einen Operanden ein »Poppen«, mit dem das Register automatisch »geleert« wird, da es ja keinen Stack gibt.

Operanden

Mit MOVD ist der Datenaustausch zwischen einem MMX-Register und einer Speicherstelle bzw. einem Allzweckregister in beiden Richtungen möglich, wobei jeweils lediglich auf das niedrigerwertige DoubleWord des MMX-Registers zugegriffen wird (ScalarDouble). Beim Kopieren in ein MMX-Register wird das höherwertige DoubleWord gelöscht:

- Kopieren eines Datums aus einer Speicherstelle in ein MMX-Register

```
MOVD MMX, Mem32
```

- Kopieren eines Datums aus einem MMX-Register in eine Speicherstelle
  `MOVD Mem32, MMX`
- Kopieren eines Datums aus einem Allzweckregister in ein MMX-Register
  `MOVD MMX, Reg32`
- Kopieren eines Datums aus einem MMX-Register in ein Allzweckregister
  `MOVD Reg32, MMX`

MOVQ ist als genereller Datenaustauscher zwischen einem MMX-Register und einer Speicherstelle in beiden Richtungen möglich:

- Kopieren eines Datums aus einem MMX-Register in ein anderes MMX-Register
  `MOVQ MMX, MMX`
- Kopieren eines Datums aus einer Speicherstelle in ein MMX-Register
  `MOVQ MMX, Mem64`
- Kopieren eines Datums aus einem MMX-Register in eine Speicherstelle
  `MOVQ Mem64, MMX`

Verschiedenes 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 zeitaufwändig – 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 leerfegt. 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!

Operanden

Der Befehl EMMS besitzt keine Operanden.

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 abgeholt 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 (vgl. Seite 264) noch FSAVE/FRSTOR (vgl. Seite 259) 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: Fließkomma-Berechnungen 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 die FPU nicht noch ihre Register sichern, während die CPU 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 offen stehen – 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, sind ihre Aufgaben die gleichen! Eine weitere Besprechung ist an dieser Stelle damit nicht erforderlich.

Operanden

Beide Befehle, FXSAVE und FXRSTOR, benötigen einen 512-Byte-Operanden, in den die Umgebung eingetragen werden kann bzw. aus dem sie ausgelesen wird:

```
FXSAVE Mem512
FXRESTOR Mem512
```



Auf Seite 868 zeigt Abbildung 5.47 das Speicherabbild der Umgebung der FPU-, MMX- und XMM-Register, wie es von den Befehlen FXSAVE und FXRSTOR verwendet wird.

Beispiele

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 AND-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 AND-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 zeitaufwändiges 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 zeitaufwändige 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, 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 ausgerechnet $00 oder $FF als Ergebnis hat: Das erzeugt so schöne Masken.)

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 255-mal (das letzte Bild muss nicht berechnet werden: Es ist das Endbild). Das bedeutet: 470.016.000-mal Laden und Multiplizieren sowie 235.008.000-mal 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 allgemeiner Ba-

sis 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 Signal-verarbeitenden 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 Matrix-Berechnungen um über zwei Drittel beschleunigen.

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-, FPU- 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 erfolgen, 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, sodass 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 offen legen, 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 selbst definierte 64-Bit-Strukturen (die Sie ja immer noch ShortPackedBytes 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 Tipp: 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*!

1.3.2 MMX-Exceptions

Non-numeric exceptions

MMX-Befehle unterscheiden sich nicht grundlegend von arithmetischen CPU-Befehlen, selbst wenn sie in den FPU-Registern ablaufen, da auch sie mit Integers arbeiten und die CPU für sie zuständig ist. Daher können alle MMX-Befehle grundsätzlich auch CPU-Exceptions auslösen, wenn sie die entsprechenden Ursachen haben. Diese »non-numeric exceptions« betreffen

- Ausnahmesituationen, die beim Zugriff auf den Speicher auftreten können (#GP, #SS, #PF, #AC)
- System-Exceptions (#UD, #NM)
- anhängige FPU-Exceptions (#MF)

Weitere CPU-Exceptions können, je nach Situation, ebenfalls ausgelöst werden. Details hierzu finden Sie im Kapitel »Exceptions und Interrupts« auf Seite 486.

Numeric Exceptions

Exceptions, die die FPU betreffen, können durch MMX-Befehle nicht ausgelöst werden, obwohl sie in den FPU-Registern ausgeführt werden, da keine Fließkommazahlen eingesetzt werden. Daher können fließkommaspezifische Ausnahmesituationen wie Denormalisierung, nicht exakte Ergebnisse oder numerischer Über- oder Unterlauf (aufgrund warp-around oder saturation) nicht auftreten. Und falls einmal eine Division durch Null erfolgen sollte, gibt es ja die #DE. Doch diese Exception wird wohl sehr selten auftreten, da kein MMX-Befehl eine Division beinhaltet ...

1.3.3 MMX-Emulation

Leider ist keine MMX-Emulation analog der FPU-Emulation vorgesehen. Falls das EM-Flag in CR0 gesetzt sein sollte, werden die FPU-Befehle aufgrund nicht vorhandener FPU emuliert. Wenn aber keine FPU vorhanden ist, gibt es auch keine FPU- und damit keine MMX-Register, was die Nutzung der MMX-Befehle unmöglich macht. Daher wird in diesem Fall bei dem Versuch, eine MMX-Operation auszuführen, eine #UD (*invalid opcode exception*) ausgelöst. Falls Sie also MMX-Befehle nutzen möchten, müssen Sie darauf achten, dass das EM-Flag gelöscht ist – was nur dann der Fall ist, wenn das System über eine FPU verfügt.

1.3.4 SIMD, die Zweite: SSE

MMX ist schon etwas. Aber wie so häufig merkt man schnell, dass das, was man heute beklatscht, schnell nicht mehr ausreicht. So erging es auch der MMX-Erweiterung. Vor allem Anwendungen mit anspruchsvoller Graphik in hoher Auflösung, wie sie in Spielen anzufinden sind, stellen höhere Anforderungen, als MMX sie befriedigen kann. Dies ist nicht verwunderlich: MMX setzt auf einfache Daten wie Bytes und Words, maximal QuadWords. Dies sind aber Integers, mit denen man manches, aber eben nicht alles machen kann!

Wer kennt sie nicht, die in wahnwitziger Geschwindigkeit zwischen steilen Felswänden dahinjagenden Kampfflugzeuge bestimmter Spielprogramme. Wer hat nicht schon die wahnsinnigen Loopings und Turns bestaunt, die die digitalen Piloten, eventuell gesteuert vom Spieler, auf den Bildschirm legen. Alles dreidimensional und gerendert, im hochglanzpolierten Flügel spiegelt sich die Abendsonne und auf dem Visier des Helmes des Piloten die anfliegende Luft-Luft-Rakete, der auszuweichen ist! Mit Bytes und Words, seien sie auch vorzeichenbehaftet, nicht mehr zu realisieren. Denn um solche Bewegungsabläufe wie z.B. das Drehen um eine Achse realisieren zu können, muss man in die Trickkiste greifen. Wir erinnern uns, wenn wir ein wenig nachdenken, an unseren Mathematikunterricht und ein Thema, das viele von uns gar nicht so gerne hatten: Vektorrechnung. Fallen Ihnen hierbei auch spontan solche Begriffe wie Matrix, Kreuzprodukt und Eigenwerte ein? Dann wissen Sie ja auch noch, dass z.B. eine Drehung eines Körpers um eine Achse ein Klacks ist – wenn man die geeignete Matrix hat, die die Drehung beschreibt, und ein bisschen Vektorrechnung beherrscht. Ja, auch bloße Verschiebungen, eine geeignete Matrix vorausgesetzt, sind Kinderkram. Und auch eine komplexe Bewegung wie eine Bewegung um einen gewissen Betrag in eine bestimmte Richtung mit gleichzeitiger Drehung um einen bestimmten Betrag in einer bestimmten Ebene verlieren den Schrecken – hat man die geeignete Matrix.

Diese zu erstellen ist nicht so schwer, wie wir uns ebenfalls erinnern werden. So gibt es für alle Verschiebungen und Drehungen in alle beliebigen Richtungen »Grundmatrizen«, die einfach durch Multiplikation zu einer »Arbeitsmatrix« zusammengesetzt werden können, die dann die komplexe Bewegung beschreibt. Und die lässt man nun auf alle Punkte (Vektoren) des Objektes los, was dieser Bewegung folgen soll. Wo liegt eigentlich das Problem? Ach ja, da war ja was: Matrizen und die Ergebnisse der Berechnung mit ihnen lassen sich nur in vernachlässigbar wenigen Fällen mit Integers ermöglichen ...

SSE ... was uns sofort zu der zweiten Multimediaerweiterung im SIMD-Konzept führt, den *S*treaming *S*IMD *E*xtensions. [Kann mir mal jemand erklären, warum Extensions im Amerikanischen einmal mit X (MMX) und einmal mit E (SSE) abgekürzt werden?] Diesen Erweiterungen des Prozessor-Befehlssatzes liegen zwei Forderungen zugrunde, die mit dem Beispiel von eben leicht nachzuvollziehen sind: Erstens die Forderung nach der Verarbeitung von Fließkommazahlen und zweitens die der Verarbeitung großer (»strömender« = streaming) Datenmengen durch Verbesserungen in den Lade-/Speichermechanismen (MPEG- und ähnliche De-/Encoder lassen grüßen!).

Die erste Forderung aber hat weit reichende Folgen: Die kleinste der FPU (die ja auf Realzahlen spezialisiert ist) bekannte Fließkommazahl ist eine SingleReal mit vier Bytes Umfang. Noch kleiner hätte es auch wenig Sinn: Der Wertebereich ist mit 10^{-38} bis 10^{38} zwar nicht schlecht und für die meisten Anwendungen im genannten Bereich wahrscheinlich durchaus ausreichend, aber auch nicht besonders üppig! Wichtiger aber ist die Genauigkeit, mit der man mit SingleReals rechnen kann: Schon eine LongInt mit dem gleichen Platzbedarf kann im Format SingleReal nicht mehr exakt dargestellt werden. Denn soll der maximal darstellbare Wert mit $2^{32} = 4.294.967.296 = 4{,}294967296 \cdot 10^{10}$ noch exakt als Realzahl darstellbar sein, benötigt man 9 dezimale bzw. 32 binäre signifikante Stellen, die eine SingleReal mit 7 dezimalen bzw. 23 binären bereits nicht mehr hat.

Heißt also im Umkehrschluss: Werden »nur« Integers mit 7 dezimalen signifikanten Stellen im Rahmen von Fließkomma-Berechnungen verwendet oder werden Reals mit einem größeren Wertebereich als LongInts und »nur« 7 dezimalen Stellen Genauigkeit benötigt, ist man mit einer SingleReal bestens bedient. Für größere Genauigkeiten und/oder größere Wertebereiche müssten dann mindestens DoubleReals gefordert werden. Oder anders ausgedrückt: Eine noch kleinere Real für Datenstrukturen analog der ShortPackedWords macht keinen Sinn. Dann sollte lieber die Integer-Arithmetik mit einer entsprechenden Skalierung verwendet werden (wie z.B. in FIBU-Programmen, in denen mit Integers in Einheiten von 1/1000stel Pfennig gerechnet wird).

Um einigermaßen sinnvoll arbeiten zu können, müssen, bleiben wir bei den oben genannten Anwendungsbereichen, mindestens drei, besser vier solcher SingleReals gleichzeitig verarbeitet werden können (Vektorrechnung!). Konsequenz: Die SSE-Befehle müssen mit bis zu 128 Bits (= 16 Bytes) umgehen können. Und am Horizont macht sich bereits ein Silberstreifen in Form des Wunsches nach drei bis vier LongReals im

»gepackten« Format breit. (Sollte das zu einer Erweiterung des SSE führen? Na klar!)

SSE-Datenformat

Analog zu MMX wurde daher ein neues Datenformat definiert, das »Packed Single Precision Floating Point Value«. Es besteht analog der ShortPackedIntegers der MMX-Erweiterung aus vier SingleReals, die zusammen in einem Register behandelt werden.

An dieser Stelle eine kleine Zäsur! Intel hat mit MMX und SSE neue Datenformate definiert, die – und ich greife an dieser Stelle ein wenig voraus – unter SSE2 noch um weitere ergänzt werden. Alles in allem ein für viele nicht ganz leicht zu durchschauendes Dickicht aus Wortungetümen einerseits (»128-Bit packed double-precision floating-point values«) und leicht verwechselbaren Definitionen andererseits (ist eine »packed byte integer« nun 64 oder 128 Bits breit? Antwort: beides! Es kommt darauf an, in welcher Umgebung. Wir werden das noch sehen.) Verschlimmert wird das Ganze noch durch konkurrierende Chiphersteller, die die Fließkomma-Erweiterungen der Multimedia-Extensions anders als Intel realisieren und dadurch andere Datenstrukturen einführen, die sie aber nicht anders nennen!

Zum einen hätte mich der Verlag erschlagen, wenn ich die Produktionskosten aufgrund der Verwendung solcher Namen in die Höhe getrieben hätte. Zum anderen muss ich zugeben: Ich bin äußerst faul und möchte vermeiden, an jeder Stelle erneut nachdenken zu müssen, in welchem Kontext nun die *packed byte integer* zu interpretieren ist. Daher habe ich mir die Sache einfacher gemacht, indem ich die in Tabelle 5.23 im Anhang auf Seite 844 aufgeführten Begriffe verwenden werde. Ich stütze mich dabei auf die Elementformate, die für die CPU-Allzweck- und FPU-Register bereits definiert wurden.

Doch zurück zu den vier SingleReals, die in einem Register zusammen gepackt werden sollen. Dieses Vorhaben sprengt den hardwareseitig vorgegebenen Rahmen: Die FPU-Register sind mit 80 Bit Breite bislang die größten gewesen und wurden daher für MMX zweckentfremdet, was auch für Daten bis zu ShortPackedDWords (= 64 Bit) durchaus ausreichend war. Für vier SingleReals aber reicht das nicht!

XMM-Register

Daher hat Intel dem Prozessor acht neue Datenregister spendiert: die XMM-Register. Nein, das ist kein Druckfehler! Sie heißen tatsächlich E*X*tended *M*ulti *M*edia Register und machen einem damit das Lesen von Quellcode nicht gerade einfacher! Angesprochen werden sie, wen wird's wundern, mit XMM0 bis XMM7. Sie umfassen jeweils 128 Bit = 16 Byte, gerade ausreichend für vier PackedSingleReals. Und ebenfalls

nein, kein Druckfehler! Sie sind wirklich und physikalisch vorhanden, vollständig unabhängig von CPU- und FPU-Registern und daher ein echter Zugewinn! Abbildung 1.36 zeigt den Registersatz. In der Abbildung wird in XMM0 gerade ein SSE-Befehl auf eine ScalarSingle angewendet. Die drei verbleibenden SingleReals der PackedSingle sind damit zwar vorhanden, aber »maskiert« und inaktiv. XMM2 bis XMM7 sind derzeit leer.

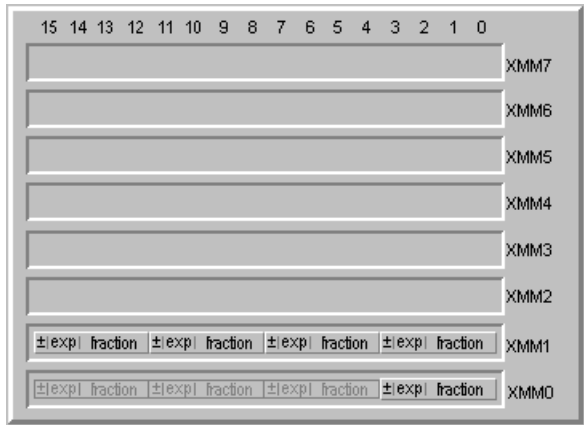


Abbildung 1.36: Die XMM-Register des Prozessors

Der Umgang mit Realzahlen in dieser Umgebung hat aber eine weitere Konsequenz, wie wir bereits bei der Besprechung der FPU-Befehle gesehen haben: Es können Exceptions auftreten, da anders als bei Integers, die man sinnvollerweise entweder sättigen oder bei denen zumindest eine Modulo-Bildung mit dem größtmöglichen Wert erfolgen kann (wrap-around), dies alles bei Realzahlen keinen Sinn macht. Hier zählt vielmehr, wie wird ggf. gerundet, was passiert mit denormalisierten Zahlen, was hat bei Über- oder Unterlauf zu erfolgen – kurz all die netten Ausnahmen, die auch in der FPU eine Rolle spielen. Und dementsprechend gibt es ein weiteres Register, das MXCSR, in dem Flags gesetzt und abgefragt werden können, um diesen Randbedingungen Rechnung zu tragen. Wir werden es im Rahmen der Exception-Besprechung unter SSE/SSE2 besprechen (vgl. Abbildung 1.39 auf Seite 361).

SSE-Befehle

Die Befehle des SSE-Befehlssatzes lassen sich zunächst in zwei große Gruppen aufteilen (vgl. Tabelle 5.26 auf Seite 846):

- Erweiterungen des MMX-Befehlssatzes, also Befehle, die mit Short-PackedIntegers umgehen, und

- Neu eingeführte Befehle, die die eben beschriebenen Datenstrukturen der PackedSingles und die Register der neu eingeführten XMM-Register betreffen.

MMX-Erweiterungen unter SSE

Die Erweiterungen des MMX-Befehlssatzes betreffen, wie gesagt, wie die originalen MMX-Befehle nur die bislang schon bekannten ShortPackedIntegers und werden in den FPU-Registern im MMX-Modus bearbeitet.

PAVGB
PAVGW

Die beiden Befehle verhalten sich absolut gleich: Im Falle von PAVGB, *packed average byte*, werden allerdings nur ShortPackedBytes, im Falle von PAVGW, *packed average word*, ShortPackedWords in die Berechnung einbezogen. Die Befehle sind nur für vorzeichenlose Daten verfügbar! Die Aktion des Befehls selbst ist simpel zu erklären: Jeweils ein Datum aus dem Quell- und Zielregister wird entnommen, addiert und durch zwei geteilt: Fertig ist der Mittelwert. Da es sich jedoch um Integers handelt, dürfen keine Nachkommateile auftreten. Daher erfolgt die Mittelwertbildung, indem zur Summe 1 addiert und das Ergebnis um eine Bitposition nach rechts verschoben wird (hier gezeigt mit PAVGW):

```
MMx[15..00] := (MMx[15..00] + MMy[15..00] + 1) SHR 1;
MMx[31..16] := (MMx[31..16] + MMy[31..16] + 1) SHR 1;
MMx[47..32] := (MMx[47..32] + MMy[47..32] + 1) SHR 1;
MMx[63..48] := (MMx[63..48] + MMy[63..48] + 1) SHR 1;
```

Auf diese Weise ist das Ergebnis grundsätzlich **auf**gerundet : (1 + 1 + 1) Div 2 = 3 Div 2 = 1; (2 + 1 + 1) Div 2 = 4 Div 2 = 2; (2 + 2 + 1) Div 2 = 5 Div 2 = 2; (3 + 1+ 1) Div 2 = 5 Div 2 = 2.

Analoges erfolgt natürlich auch byteweise mit PAVGB.

Operanden

Als Ziel für die Summe und Quelle des ersten Operanden kommt nur ein MMX-Register in Frage, während der zweite Additionspartner in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PAVGB, PAVGW):

- Mittelwertbildung einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, MMX`
- Mittelwertbildung einer ShortPackedInteger aus einer Speicherstelle mit einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, Mem64`

PEXTRW PINSRW

PEXTRW, *packed extract word*, nimmt ein spezifiziertes Word aus dem ShortPackedWord des Operanden und transferiert es in die unteren 16 Bits (= Word) eines Allzweckregisters der CPU. Ein Beispiel: *PEXTRW EAX, MM3, 2* kopiert die Bits 47 bis 32 von MM3, also das dritte Word (die Indizierung beginnt mit 0, daher bezeichnet der dritte Operand, »2«, das dritte word!) aus dem PackedWord in MM3, in das untere Word (Bits 15..0) des EAX-Registers.

Den umgekehrten Weg geht PINSRW, *packed insert word*: Es transferiert aus dem unteren Wort des Allzweckregisters ein Wort an die angegebene Stelle in einem MMX-Register. Auch hier ein erklärendes Beispiel: *PINSRW MM0, ECX, 0* kopiert Bit 15 bis 0 aus ECX in die Bitpositionen 15 bis 0 von MM0, das erste Word des ShortPackedWords in MM0.

PINSRW hat jedoch gegenüber PEXTRW noch eine weitere Möglichkeit: Quelle des Wortes muss nicht ein Allzweckregister sein, es kann auch ein Wort aus dem Speicher direkt in das entsprechende Feld des MMX-Registers kopiert werden: *PINSRW MM7, WordVar, 3*.

Operanden

Als Ziel für das aus der Quelle durch PEXTRW extrahierte Word kommt nur ein Allzweckregister in Frage, während die Quelle in einem MMX-Register als zweitem Operanden stehen muss. Ziel des einzufügenden Words bei PINSRW kann nur ein MMX-Register sein, als Quelle und damit zweitem Operanden kommt entweder ein Allzweckregister in Frage oder eine Speicherstelle. In allen Fällen muss als dritter Operand die Position des zu extrahierenden/einzufügenden Words als Konstante angegeben werden.

- Extraktion eines Words aus einem MMX-Register
  `PEXTRW Reg32, MMX, Const8`
- Insertion eines Words aus einem Allzweckregister
  `INSRW MMX, Reg32, Const8`
- Insertion eines Words aus einer Speicherstelle
  `INSRW MMX, Mem16, Const8`

PMAXSW PMAXUB PMINSW PMINUB

Diese Befehlsgruppe berechnet Minima und Maxima von zwei ShortPackedIntegers. Es können entweder vorzeichenbehaftete Worte (SW; *signed word*) oder vorzeichenlose Bytes (UB; *unsigned byte*) verwendet werden. Der Zieloperand muss immer ein MMX-Register sein, als Quelle können entweder ein MMX-Register oder eine entsprechende Datenstruktur im Speicher sein, wie an den folgenden Beispielen gezeigt wird. Zunächst *PMAXSW MM0, MM3*

```
MM0[15..00] := MAX(MM0[15..00], MM3[15..00])
MM0[31..16] := MAX(MM0[31..16], MM3[31..16])
MM0[47..32] := MAX(MM0[47..32], MM3[47..32])
MM0[63..48] := MAX(MM0[63..48], MM3[63..48])
```

Analoges gilt für *PMINUB MM7, EightByteVar*:

```
MM7[07..00] := MAX(MM7[07..00], EightByteVar[Adr + 0])
MM7[15..08] := MAX(MM7[15..08], EightByteVar[Adr + 8])
MM7[23..16] := MAX(MM7[23..16], EightByteVar[Adr + 16])
MM7[31..24] := MAX(MM7[31..24], EightByteVar[Adr + 24])
MM7[39..32] := MAX(MM7[39..32], EightByteVar[Adr + 32])
MM7[47..40] := MAX(MM7[47..40], EightByteVar[Adr + 40])
MM7[55..48] := MAX(MM7[55..48], EightByteVar[Adr + 48])
MM7[63..56] := MAX(MM7[63..56], EightByteVar[Adr + 56])
```

Operanden

Als Ziel für den Extremwert und Quelle des ersten Operanden kommt nur ein MMX-Register in Frage, während der zweite Additionspartner in einem MMX-Register oder an einer Speicherstelle stehen kann (XXX steht hier für PMAXSW, PMAXUB, PMINSW oder PMINUB):

- Extremwertbildung einer ShortPackedInteger aus einem MMX-Register und einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, MMX`
- Extremwertbildung einer ShortPackedInteger aus einer Speicherstelle und einer ShortPackedInteger in einem MMX-Register
  `XXX MMX, Mem64`

PMOVMSKB

Packed move byte mask, PMOVMSKB, erzeugt aus den *Most Significant Bits* (MSB) der Bytes eines ShortPackedBytes eine Maske und legt diese in einem Allzweckregister der CPU ab. Das Beispiel mit *PMOVMSKB EAX, MM5*:

```
Temp[0] := MM5[07]     // MSB des Bytes 0 in MM5
Temp[1} := MM5[15]     // MSB des Bytes 1 in MM5
Temp[2] := MM5[23]     // MSB des Bytes 2 in MM5
Temp[3] := MM5[31]     // MSB des Bytes 3 in MM5
Temp[4] := MM5[39]     // MSB des Bytes 4 in MM5
Temp[5] := MM5[47]     // MSB des Bytes 5 in MM5
Temp[6] := MM5[55]     // MSB des Bytes 6 in MM5
Temp[7] := MM5[63]     // MSB des Bytes 7 in MM5
EAX[07..00] := Temp
EAX[31..08] := 0
```

Operanden Als Ziel für die Maske kommt nur ein Allzweckregister in Frage. Quelle und Grundlage für die Maskenberechnung ist der Inhalt eines MMX-Registers:

```
PMOVMSKB Reg32, MMX
```

PMULHUW PMULHUW, *multiply packed unsigned words and store high result*, ist eine Ergänzung der bereits im Kapitel »SIMD, die Erste: MMX« auf Seite 274 besprochenen, auf den ersten Blick recht merkwürdigen Multiplikationen mit ShortPackedIntegers. Dieser Befehl reiht sich in die Reihe PMULLW, PMULHW (vgl. Seite 284) ein, indem er wie PMULHW zwei Words mit einander multipliziert und deren höherwertiges Wort dann in den Zieloperanden einträgt. Allerdings verwendet dieser Befehl im Unterschied zu PMULHW zwei vorzeichenlose Words:

```
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])
```

Anschließend werden dann die höherwertigen Wortanteile der berechneten Doppelworte extrahiert und in das Zielregister kopiert:

```
MMx[15..00] := Temp[031..016]
MMx[31..16] := Temp[063..048]
MMx[47..32] := Temp[095..080]
MMx[63..48] := Temp[127..112]
```

Warum gibt es PMULLUW nicht? Ganz einfach! Weil es identisch wäre mit PMULLW und damit absolut überflüssig. Beweis: Zunächst wird ja die Multiplikation durchgeführt, indem temporär das vollständige Doppelwort des Produktes aus zwei Worten gebildet wird. Die Absolutbeträge der aus der Multiplikation entstandenen Produkte einer vorzeichenlosen oder vorzeichenbehafteten Multiplikation sind aber gleich, da man ja eine Multiplikation wie folgt zerlegen kann:

```
Value1 = Signum1 * AbsValue1;
Value2 = Signum2 * AbsValue2;
Product = Value1 * Value2 = Signum1 * AbsValue1 * Signum2 * AbsValue2;
Product = (Signum1 * Signum2) * (AbsValue1 * AbsValue2)
Product = Signum * AbsValue
```

Unterschiede bei den Ergebnissen einer vorzeichenlosen und vorzeichenbehafteten Multiplikation liegen daher ausschließlich im MSB des Ergebnisses: dem Vorzeichen. Daher unterscheiden sich auch nur die höherwertigen Wortanteile des Doppelwortes der entsprechenden Produkte. Sie sehen: PMULLUW ist absolut überflüssig!

(Wenn man es ganz konsequent durchdenkt, stimmt diese Aussage nicht ganz. Sie ist nur richtig, wenn man die beiden Worte des durch Multiplikation entstandenen Doppelwortes tatsächlich als Teile eines Doppelwortes auffasst. Fasst man die Befehle dagegen als Kombination einer Multiplikation mit anschließender Integer-Division mit dem Divisor $10000 auf, wie wir das weiter oben auch getan haben, so müsste es ein PMULLUW geben, da sich die Integer-Division vorzeichenbehafteter und vorzeichenloser Zahlen durch ein Vorzeichen unterscheiden und PMULLW müsste aus dem gleichen Grund Daten mit Vorzeichen erzeugen, was es nicht tut! Aber das sind nun wirklich akademische Spitzfindigkeiten.)

Operanden

Als Ziel für das Produkt und Quelle des Multiplikanden der gepackten Multiplikation kommt nur ein MMX-Register in Frage, während der Multiplikator in einem MMX-Register oder an einer Speicherstelle stehen kann:

- Multiplikation einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger in einem MMX-Register
  `PMULHUW MMX, MMX`
- Multiplikation einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger aus einer Speicherstelle
  `PMULHUW MMX, Mem64`

PSADBW

PSADBW, *compute sum of absolute differences of packed bytes as word*, berechnet zunächst für jedes Byte in einem ShortPackedByte den Absolutwert der Differenz der beiden Operanden:

```
Temp[07..00] := ABS(SUB(MMx[07..00], MMy[07..00]))
Temp[15..08] := ABS(SUB(MMx[15..08], MMy[15..08]))
Temp[23..16] := ABS(SUB(MMx[23..16], MMy[23..16]))
Temp[31..24] := ABS(SUB(MMx[31..24], MMy[31..24]))
Temp[39..32] := ABS(SUB(MMx[39..32], MMy[39..32]))
Temp[47..40] := ABS(SUB(MMx[47..40], MMy[47..40]))
Temp[55..48] := ABS(SUB(MMx[55..48], MMy[55..48]))
Temp[63..56] := ABS(SUB(MMx[63..56], MMy[63..56]))
```

In einem zweiten Schritt wird nun die Summe dieser Differenzen addiert, wobei die Bytes auf Wortgröße expandiert werden (denn die Gesamtsumme kann ja locker die Bytegrenze überschreiten!):

```
Temp2[15..00] := EXPAND(Temp[07..00])
Temp2[15..00] := Temp2[15..00] + Temp[15..08]
Temp2[15..00] := Temp2[15..00] + Temp[23..16]
Temp2[15..00] := Temp2[15..00] + Temp[31..24]
```

```
Temp2[15..00] := Temp2[15..00] + Temp[39..32]
Temp2[15..00] := Temp2[15..00] + Temp[47..40]
Temp2[15..00] := Temp2[15..00] + Temp[55..48]
Temp2[15..00] := Temp2[15..00] + Temp[63..56]
```

Schließlich wird das Word in die Bits 15 bis 0 des Zieloperanden kopiert. Alle weiteren Bits werden gelöscht:

```
MMx[15..00] := Temp2[15..00]
MMx[63..16] := 0
```

Operanden Als Ziel für die Berechnung und Quelle des ersten Partners kommt nur ein MMX-Register in Frage, während der zweite Partner in einem MMX-Register oder an einer Speicherstelle stehen kann:

- SAD-Bildung einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger in einem MMX-Register
  `PSADBW MMX, MMX`
- SAD-Bildung einer ShortPackedInteger aus einem MMX-Register mit einer ShortPackedInteger aus einer Speicherstelle
  `PSADBW MMX, Mem64`

PSHUFW Dieser Befehl ist der »Gambler« unter den SSE-Befehlen, da er Daten mischt wie ein Kartenspieler seine Karten – wenn auch nicht auf Zufall basierend, sondern sehr akkurat und zuverlässig nach vorzugebenden Regeln. Gemischt werden immer Worte eines Quelloperanden, die dann in einem Zieloperanden neu zusammengesetzt werden. Die Regeln werden in einem dritten Parameter, einer Konstanten, übergeben, wie z.B. in *PSHUFW MM0, MM2, 37*.

Zu den Regeln: Die Konstante, ein Byte, wird interpretiert als Feld mit vier Einträgen à 2 Bit:

```
Rule0 = Const[1..0]
Rule1 = Const[3..2]
Rule2 = Const[5..4]
Rule3 = Const[7..6]
```

Diese Bits werden als Ziffer interpretiert, die dadurch maximal Werte zwischen 0 und 3 annehmen kann. Im Beispiel, die Konstante 37 wird hexadezimal als $1B (= 00011011b = 00_01_10_11) dargestellt, hätte Rule0 die Bitfolge »11«, was »3« bedeutet. Rule1 (»10«) hätte den Wert 2, Rule2 (»01«) den Wert 1 und Rule3 (»00«) den Wert 0.

Diese Regeln sind in Wirklichkeit die Indizes in das ShortPackedWord in der Quelle, die an die festgelegten Stellen im Ziel-ShortPackedWord kopiert werden sollen, und zwar:

```
MMx[Word(0)] := MMy[Word(Rule0)]
MMx[Word(1)] := MMy[Word(Rule1)]
MMx[Word(2)] := MMy[Word(Rule2)]
MMx[Word(3)] := MMy[Word(Rule3)]
```

Unser Beispiel würde also folgende Kopierarbeit leisten:

```
MM0[15..00] := MM2[63..48]
MM0[31..16] := MM2[47..32]
MM0[47..32] := MM2[31..16]
MM0[63..48] := MM2[15..00]
```

Das ist gleichbedeutend mit einer Umorientierung der Wortfolge von hinten nach vorne! Wie Sie sehen, ist der Befehl nicht uninteressant. Denn er verbietet nicht, gleiche Quellindices für unterschiedliche Ziele zu benutzen, wie in *PSHUFW MM3, MM4, 149*. Die Analyse der Konstanten (= $95 = 10_01_01_01b) zeigt: Die Zielindices 0 bis 2 werden mit Quellindex = 1 belegt, Zielindex 3 mit Quellindex = 2, also:

```
MM3[15..00] := MM4[31..16]
MM3[31..16] := MM4[31..16]
MM3[47..32] := MM4[31..16]
MM3[63..48] := MM4[47..32]
```

Genial, oder?

Operanden

Als Ziel des Mischens kommt nur ein MMX-Register in Frage, während die Quelle in einem MMX-Register oder an einer Speicherstelle stehen kann:

- Mischen der Komponenten einer ShortPackedInteger aus einem MMX-Register in einer ShortPackedInteger in einem MMX-Register
  `PSHUFW MMX, MMX, Const8`
- Mischen der Komponenten einer ShortPackedInteger aus einer Speicherstelle in einer ShortPackedInteger in einem MMX-Register
  `PSHUFW MMX, Mem64, Const8`

XMM-Befehle

Bis hierher wurden lediglich die Erweiterungen besprochen, die unter SSE den MMX-Befehlssatz betreffen. Kommen wir nun zu den neuen Möglichkeiten unter SSE, die mit der Nutzung der neuen Register und

des neuen Datentyps PackedSingleReal realisiert werden können. Die implementierten Befehle umfassen Möglichkeiten zur

- Bearbeitung von PackedSingleReals
- Verwaltung der XMM-Register und
- Optimierung der Datenströme

XMM-Arithmetik

Die Befehle zur Verarbeitung von PackedSingleReals lassen sich wiederum einteilen in Befehle zum

- arithmetischen Manipulieren der Daten
- logischen Manipulieren der Daten
- Datenvergleich
- SAD-Bildung
- Datenkonversion

Skalare XMM-Daten

Der XMM-Befehlssatz zeichnet sich durch eine Besonderheit aus. Analog der Instruktionen mit ShortPackedIntegers und MMX werden auch PackedSingleReals unter XMM parallel mit einer Instruktion bearbeitet, sodass tatsächlich vier Daten auf einmal verändert werden. Darüber hinaus jedoch besteht auch die Möglichkeit, nur jeweils die SingleReals an der »niedrigsten« Position der Operation zu verarbeiten, während die drei an den »höheren Positionen« befindlichen unverändert bleiben:

```
XMMx[031..000] := Operation(XMMx[031..000], XMMy[031..000])
XMMx[063..032] := XMMx[063..032]             // unverändert
XMMx[095..064] := XMMx[095..064]             // unverändert
XMMx[127..096] := XMMx[127..096]             // unverändert
```

Diese Art der Berechnungen nennt Intel »skalar« und vergleicht sie mit den Berechnungen, die in den FPU-Registern ablaufen. (Bitte beachten Sie hierbei, dass das, was Intel in seinen Dokumentationen »erster« Quelloperand nennt, auch gleichzeitig der Zieloperand ist: In ADD EAX, ECX beispielsweise ist EAX sowohl erster Quelloperand als auch Zieloperand, während ECX der zweite Quelloperand ist. Die Operation läuft also ab nach EAX + ECX ➞ EAX. Dies ist wichtig, da häufig genug, auch von Intel, davon gesprochen wird, dass bei Operationen auf skalare SingleReals die drei »höheren« SingleReals im XMM-Register vom Quelloperanden in den Zieloperanden »durchgereicht« werden. Das mag formal ja auch stimmen. De facto jedoch passiert nichts! Denn nach dem eben gesagten sind ja erster Quelloperand und Zieloperand identisch, sodass die Inhalte der »höherwertigen« drei skalaren SingleReals

bei skalaren Operationen schlichtweg unverändert bleiben. Daher ist auch richtig, wenn Intel skalare XMM-Operationen mit FPU-Operationen vergleicht: Die entsprechende Operation könnte genauso gut auch in den FPU-Registern mit den »least significant« PackedSingleReals als Operanden ablaufen, sofern der entsprechende Befehl im FPU-Befehlssatz überhaupt implementiert ist.)

Arithmetische Befehle

Die arithmetischen Befehle umfassen erheblich mehr Möglichkeiten als die analogen Befehle unter MMX auf die gepackten Integers. So können die gepackten und skalaren SingleReals addiert werden, subtrahiert, multipliziert und dividiert; es können die reziproken Werte berechnet werden, die Quadratwurzeln und die reziproken Quadratwurzeln; schließlich können wie bei den MMX-Erweiterungen unter SSE auch die Minima und Maxima berechnet werden.

ADDPS ADDSS

Jeweils für den »gepackten« und »skalaren« Fall gibt es einen Additionsbefehl. So addiert ADDPS, *add packed single-precision floating-point values*, zwei PackedSingleReals, während ADDSS, *add scalar single-precision floating-point values*, das Gleiche scalar erledigt. Zumindest was die gepackte Version betrifft, erwartet uns hierbei nichts Überraschendes:

```
XMMx[031..000] := ADD(XMMx[031..000], XMMy[031..000])
XMMx[063..032] := ADD(XMMx[063..032], XMMy[063..032])
XMMx[095..064] := ADD(XMMx[095..064], XMMy[095..064])
XMMx[127..096] := ADD(XMMx[127..096], XMMy[127..096])
```

Auch der skalare Fall läuft wie erwartet ab:

```
XMMx[031..000] := ADD(XMMx[031..000], XMMy[031..00])
XMMx[127..000] := XMMx[127..032]      // unverändert.
```

Operanden

Als Ziel für die Summe und Quelle des ersten Operanden kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während der zweite Additionspartner bei gepackten Strukturen in einem XMM-Register oder an einer Speicherstelle stehen kann, bei skalaren in einem XMM-Register oder einem Allzweckregister:

- Addition einer gepackten oder skalaren SingleReal aus einem XMM-Register mit einer solchen SingleReal aus einem XMM-Register:

  ```
  ADDPS XMM, XMM
  ADDSS XMM, XMM
  ```

- Addition einer gepackten oder skalaren SingleReal aus einem XMM-Register mit einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister:
  ```
  ADDPS XMM, Mem128
  ADDSS XMM, Reg32
  ```

DIVPS
DIVSS
MULPS
MULSS
SUBPS
SUBSS

Für *divide packed single-precision floating-point value*, DIVPS, *divide scalar single-precision floating-point value*, DIVSS, *multiply packed single-precision floating-point value*, MULPS, m*ultiply scalar single-precision floating-point value*, MULSS, *subtract packed single-precision floating-point value*, SUBPS, und *subtract scalar single-precision floating-point value*, SUBSS, gilt das Analoge zu den Additionen, weshalb ich auf eine weitere Darstellung und Besprechung verzichte.

Operanden

Als Ziel der Operation und Quelle des ersten Operanden kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während der zweite Operationspartner bei gepackten Strukturen in einem XMM-Register oder an einer Speicherstelle stehen kann, bei skalaren in einem XMM-Register oder einem Allzweckregister (XXX steht für DIVPS, MULPS und SUBPS, YYY für DIVSS, MULSS und SUBSS):

- Arithmetische Verknüpfung einer gepackten oder skalaren SingleReal aus einem XMM-Register mit einer solchen SingleReal aus einem XMM-Register
  ```
  XXX XMM, XMM
  YYY XMM, XMM
  ```
- Arithmetische Verknüpfung einer gepackten oder skalaren SingleReal aus einem XMM-Register mit einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister:
  ```
  XXX XMM, Mem128
  YYY XMM, Reg32
  ```

SQRTPS
SQRTSS

Diese beiden Befehle berechnen die Quadratwurzeln der im Quelloperanden verzeichneten skalaren oder gepackten SingleReals und legen sie im Zieloperanden ab. Dabei spielt sich absolut nichts Geheimnisvolles ab:

```
XMMx[031..000] := SQRT(XMMy[031..000])
XMMx[063..032] := SQRT(XMMy[063..032])
XMMx[095..064] := SQRT(XMMy[095..064])
XMMx[127..096] := SQRT(XMMy[127..096])
```

bzw.

```
XMMx[031..000] := SQRT(XMMy[031..00])
XMMx[127..000] := XMMx[127..032]      // unverändert.
```

Operanden

Als Ziel für die Berechnung der Quadratwurzel kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während die Quelle und somit das Argument der Wurzelbildung bei gepackten Strukturen in einem XMM-Register oder an einer Speicherstelle stehen kann, bei skalaren in einem XMM-Register oder einem Allzweckregister:

- Quadratwurzelbildung einer gepackten oder skalaren SingleReal aus einem XMM-Register
  ```
  SQRTPS XMM, XMM
  SQRTSS XMM, XMM
  ```
- Quadratwurzelbildung einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister
  ```
  SQRTPS XMM, Mem128
  SQRTSS XMM, Reg32
  ```

RCPPS
RCPSS

Diese beiden Befehle berechnen die Kehrwerte der SingleReals des Quelloperanden und legen sie im Zieloperanden ab. Auch dies kann entweder skalar oder mit allen vier Realzahlen einer PackedSingleReal erfolgen:

```
XMMx[031..000] := APPROXIMATE(1.0 / XMMy[031..000])
XMMx[063..032] := APPROXIMATE(1.0 / XMMy[063..032])
XMMx[095..064] := APPROXIMATE(1.0 / XMMy[095..064])
XMMx[127..096] := APPROXIMATE(1.0 / XMMy[127..096])
```

bzw.:

```
XMMx[031..000] := APPROXIMATE(1.0 / XMMy[031..000])
XMMx[127..000] := XMMx[127..032]      // unverändert.
```

Wichtig zu wissen ist hierbei, dass die Reziprokwerte Annäherungen sind, was bedeutet, dass tatsächlich Unterschiede zwischen der Reziprokwertberechnung RCPPS XMM1, XMM0 und der Division von DIVPS XMM1, XMM0 mit der Vorbelegung von jeweils 1.0 für die SingleReals in XMM1 auftreten können. So liegen die Unterschiede vor allem in den Ergebnissen und Reaktionen, wenn NaNs oder Unendlichkeiten involviert sind oder ein Über- oder Unterlauf auftritt.

Operanden Als Ziel für die Berechnung des Reziprokwertes kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während die Quelle und somit das Argument der Berechnung bei gepackten Strukturen in einem XMM-Register oder an einer Speicherstelle stehen kann, bei skalaren in einem XMM-Register oder einem Allzweckregister:

- Reziprokwertbildung einer gepackten oder skalaren SingleReal aus einem XMM-Register:
  ```
  RCPPS XMM, XMM
  RCPSS XMM, XMM
  ```
- Reziprokwertbildung einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister:
  ```
  RCPPS XMM, Mem128
  RCPSS XMM, Reg32
  ```

RSQRTPS RSQRTSS Die reziproken Quadratwurzeln der skalaren oder gepackten SingleReals sind die Bildung der Quadratwurzeln mit anschließender Kehrwertbildung. Auch bei diesen Berechnungen werden die Ergebnisse angenähert, weshalb die Einzelaktionen nach dem Motto SQRTPS XMM1, XMM2 – DIVPS XMM0, XMM1 mit jeweils 1.0 als Vorbelegung für die SingleReals in XMM0 zu unterschiedlichen Ergebnissen führen können wie die Ausführung von RSQRTPS XMM0, XMM2.

Operanden Als Ziel für die Berechnung der reziproken Quadratwurzel kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während die Quelle und somit das Argument der Wurzelbildung bei gepackten Strukturen in einem XMM-Register oder an einer Speicherstelle stehen kann, bei skalaren in einem XMM-Register oder einem Allzweckregister:

- Reziproke Quadratwurzelbildung einer gepackten oder skalaren SingleReal aus einem XMM-Register
  ```
  RSQRTPS XMM, XMM
  RSQRTSS XMM, XMM
  ```
- Reziproke Quadratwurzelbildung einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister
  ```
  RSQRTPS XMM, Mem128
  RSQRTSS XMM, Reg32
  ```

MAXPS
MAXSS
MINPS
MINSS

MAXPS, MAXSS, MINPS und MINSS machen das, was man erwartet: Sie geben entweder den größeren oder den kleineren der beiden Operanden in den Zieloperanden zurück. Dies kann entweder für alle vier SingleReals eines PackedSingleReal erfolgen (MAXPS, MINPS) oder aber skalar nur mit der niedrigstwertigen SingleReal der PackedSingleReals (MAXSS, MINSS). In diesem Falle bleiben, wie bei allen skalaren Operationen, die Inhalte der höherwertigen SingleReals im Zieloperanden unverändert.

Operanden

Als Ziel für den Extremwert und Quelle des ersten Operanden kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während der zweite Partner der Operation bei gepackten Strukturen in einem XMM-Register oder an einer Speicherstelle stehen kann, bei skalaren in einem XMM-Register oder einem Allzweckregister (XXX steht für MAXPS oder MINPS, YYY für MAXSS oder MINSS):

- Extremwertbildung einer gepackten oder skalaren SingleReal aus einem XMM-Register und einer solchen SingleReal aus einem XMM-Register
  ```
  XXX XMM, XMM
  YYY XMM, XMM
  ```
- Extremwertbildung einer gepackten oder skalaren SingleReal aus einem XMM-Register und einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister
  ```
  XXX XMM, Mem128
  YYY XMM, Reg32
  ```

Logische Operationen

Die logischen Operationen in den XMM-Registern entsprechen weitestgehend denen, die auch in den MMX-Registern ablaufen. So gibt es hier wie dort die AND-, AND-NOT-, OR- und XOR-Verknüpfung, während man eine NOT-Verknüpfung vergeblich sucht:

ANDPS
ANDNPS
ORPS
XORPS

Allerdings erfolgen diese Operationen nur mit PackedSingleReals – die entsprechenden Zwillinge (ANDSS, ANDNSS, ORSS, XORSS) für skalare SingleReals sind **nicht** implementiert. Die Befehle führen tatsächlich eine bitweise Verknüpfung der beiden Operanden durch und geben sie im Zieloperanden zurück, wie am Beispiel von ANDPS XMM0, XMM1 dargestellt:

```
XMMx[000] := XMMx[000] AND XMMy[000]
XMMx[001] := XMMx[001] AND XMMy[001]
```

```
:
:
:
XMMx[126] := XMMx[126] AND XMMy[126]
XMMx[127] := XMMx[127] AND XMMy[127]
```

Ich muss zugeben, nicht so ganz den Sinn dieser Befehle verstanden zu haben: Wozu AND, OR & Co bei *Realzahlen* – denn darum handelt es sich ja bei den XMM-Daten? So machen logische Operationen eigentlich nur in zwei Fällen so richtig Sinn: wenn man die Daten als Bit-Felder interpretiert und entsprechend manipulieren möchte oder bei trickreichen arithmetischen Integer-Operationen, bei denen die logischen Befehle für Berechnungen »zweckentfremdet« werden, die anders auch erfolgen könnten, so aber einfacher, schneller und effektiver realisiert werden können. Beispiel:

```
Result := Integer AND Maske
```

Wählt man nun z.B. für Maske $0000FFFF, so entspricht die Operation einer Modulo-Berechnung, also der Restbildung nach Division mit (Maske + 1). Konventionell müsste dies mit folgenden Prozessorbefehlen realisiert werden:

```
Temp := DIV(Integer, Wert)    // Divisionsrest abgeschnitten
Temp := MUL(Temp, Wert)       // um Divisionsrest bereinigte Integer
Result := SUB(Integer, Temp)  // Divisionsrest
```

(Unnötig zu sagen, dass man Division und Multiplikation auch durch die effektiveren Shift-Befehle – und damit auch bitorientiert! – ersetzen kann, wenn *Wert* ein ganzzahliges Vielfaches von 2 ist!) Allerdings ist eine solche Modulo-Bildung mit Realzahlen auf diese Weise aus einsichtigen Gründen nicht möglich!

Bleibt als mögliche Erklärung für die Existenz der genannten XMM-Befehle nur Folgendes:

- Ganz so Fließkomma-orientiert wie bisher angenommen sind die XMM-Register nicht. So könnten immerhin Integers und Bit-Felder mit 128 Bits verarbeitet werden (das wären zwei »PackedQuadWords« oder vier PackedDoubleWords). Dies aber würde dann zumindest in vielen Fällen die MMX-Erweiterungen überflüssig machen.
- Man möchte auch bei Realzahlen bestimmte »einfache« Berechnungen machen können. Denkbar wäre die »Absolutierung« von Real-

zahlen durch ein ANDPS mit der Maske $7FFFFFFF, die alle Bits der Realzahl unverändert lässt außer dem Vorzeichen, das explizit gelöscht wird. Oder das Gegenteil: die explizite »Negativierung« der Realzahl durch eine OR-Verknüpfung mit $80000000. Auch das »Extrahieren« des Exponenten wäre dann genauso einfach realisierbar durch AND-Verknüpfung mit $7F800000 wie das Pendant zur Gewinnung der Mantisse durch AND-Verknüpfung mit $807FFFFF.

Operanden

Als Ziel für die Operation und Quelle des ersten Operanden kommt nur ein XMM-Register in Frage, während der zweite Operationspartner in einem XMM-Register oder an einer Speicherstelle stehen kann (XXX steht für ANDPS, ANDNPS, ORPS, XORPS):

- Logische Verknüpfung einer gepackten SingleReal aus einem XMM-Register mit einer gepackten SingleReal aus einem XMM-Register
  `XXX XMM, XMM`
- Logische Verknüpfung einer gepackten SingleReal aus einem XMM-Register mit einer gepackten SingleReal aus einer Speicherstelle
  `XXX XMM, Reg32`

Datenvergleich

Wir haben bereits bei der Besprechung der MMX-Befehle gesehen, dass Vergleiche bei den Multimedia-Extensions etwas andere Resultate haben als bei »normalen« CPU- oder FPU-Vergleichen. Während bei diesen irgendwelche Flags oder *condition codes* gesetzt werden, werden durch die Multimedia-Befehle Masken als Resultat des Vergleiches gesetzt. Dies macht ja, wie wir am Beispiel der Wetterkarte gesehen haben, auch durchaus Sinn!

Die Vergleichsbefehle unter SSE bilden hiervon keine Ausnahme. Auch bei diesen Befehlen wird, je nach Ergebnis des Vergleichs, eine Maske bestehend aus lauter »1« (Bedingung erfüllt) oder »0« (Bedingung nicht erfüllt) generiert.

CMPPS
CMPSS

Für beide Fälle, die Verwendung skalarer oder gepackter SingleReals, gibt es genau jeweils einen Vergleichsbefehl: CMPPS, *compare packed single-precision floating-point values*, und CMPSS, *compare scalar single-precision floating-point values*. Das erscheint einem zunächst ein bisschen wenig, verfügt doch bereits der MMX-Befehlssatz über zwei grundlegende Arten des Vergleiches: auf Gleichheit oder größeren Wert. Und mit den Realzahlen in den FPU-Registern sind noch erheblich mehr Vergleiche möglich (realisiert über die *condition codes*!).

In der Tat jedoch ermöglichen CMPPS und CMPSS erheblich mehr Vergleiche als MMX. Realisiert wird das durch einen dritten Parameter, der den Befehlen zusätzlich zu den beiden zu vergleichenden Operanden übergeben wird. Im Intel-Jargon heißt dieser Parameter Vergleichsprädikat (»comparison predicate«) und ist eine Bytekonstante. Die unteren drei Bits codieren die Art des anzustellenden Vergleichs, die restlichen Bits gelten (wie immer, wenn etwas auf Zuwachs ausgelegt ist) als reserviert. Damit sind die in Tabelle 1.40 dargestellten »Prädikate« möglich:

pred.	comparison type	pred.	comparison type
0	equal	4	not equal
1	less than = not (greater than or equal)	5	not less than = greater than or equal
2	less than or equal = not greater than	6	not less than or equal = greater than
3	unordered	7	ordered

Tabelle 1.40: »Prädikate« der Befehle CMPPS und CMPSS und ihre Bedeutung

Falls Ihnen die Arbeit mit den Prädikaten zu ungewohnt oder nicht komfortabel genug erscheint, helfen Sie sich doch mit der Definition von Makros aus. Die hierzu notwendigen Informationen entnehmen Sie bitte Tabelle 1.41.

Makro	Instruktion	Vergleich
CMPEQPS op1, op2	CMPPS op1, op2, 0	gleich
CMPLTPS op1, op2	CMPPS op1, op2, 1	kleiner
CMPLEPS op1, op2	CMPPS op1, op2, 2	kleiner oder gleich
CMPGTPS op1, op2	CMPPS op1, op2, 6	größer
CMPGEPS op1, op2	CMPPS op1, op2, 5	größer oder gleich
CMPOPS op1, op2	CMPPS op1, op2, 7	geordnet
CMPUOPS op1, op2	CMPPS op1, op2, 3	ungeordnet
CMPNEQPS op1, op2	CMPPS op1, op2, 4	nicht gleich
CMPNLTPS op1, op2	CMPPS op1, op2, 5	nicht kleiner
CMPNLEPS op1, op2	CMPPS op1, op2, 6	nicht kleiner oder gleich
CMPNGTPS op1, op2	CMPPS op1, op2, 2	nicht größer
CMPNGEPS op1, op2	CMPPS op1, op2, 1	nicht größer oder gleich
CMPNUOPS op1, op2	CMPPS op1, op2, 7	nicht ungeordnet
CMPNOPS op1, op2	CMPPS op1, op2, 3	nicht geordnet

Tabelle 1.41: Mögliche Makronamen für die Realisierung Prädikat-unabhängiger Vergleichsbefehle unter SSE

Es sind also praktisch die gleichen Vergleiche möglich, wie sie auch mit Realzahlen in der FPU und ihren Registern realisiert werden. Ein Unterschied aber bleibt: Während man mit der FPU zwei Realzahlen vergleichen kann und dann nach dem Vergleich die Art der Beziehung feststellen kann (retrospektiv!), muss bei den XMM-Befehlen die Art des Vergleiches vor dem Ausführen der Instruktion feststehen (prospektiv!).

Da analog der Vergleichsbefehle für gepackte Integer (MMX) nicht, wie im Falle der Allzweckregister-Befehle, Flags bemüht werden können, um das Ergebnis des Resultates anzuzeigen, muss das Ergebnis im Zieloperanden codiert werden. Führt also ein Vergleich zu einem wahren Ergebnis, so wird in das Ziel an die betreffende Position $FFFFFFFF geschrieben, andernfalls wird »0« eingetragen:

```
IF XMMx[031..000] · XMMy[031..000] = TRUE
  THEN XMMx[031..000] := $FFFFFFFF
  ELSE XMMx[031..000] := $00000000;
IF XMMx[063..032] · XMMy[063..032] = TRUE
  THEN XMMx[063..032] := $FFFFFFFF
  ELSE XMMx[063..032] := $00000000;
IF XMMx[095..064] · XMMy[095..064] = TRUE
  THEN XMMx[095..064] := $FFFFFFFF
  ELSE XMMx[095..064] := $00000000;
IF XMMx[127..096] · XMMy[127..096] = TRUE
  THEN XMMx[127..096] := $FFFFFFFF
  ELSE XMMx[127..096] := $00000000;
```

wobei »○« für die betreffende Operation (siehe Tabelle 1.40) steht.

Operanden

Als Ziel für die Ergebnismaske und Quelle des ersten Vergleichsoperanden kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während der zweite Vergleichspartner bei gepackten Strukturen in einem XMM-Register oder an einer Speicherstelle stehen kann, bei skalaren in einem XMM-Register oder einem Allzweckregister. In jedem Fall gibt der dritte Operand den prediction code an und ist eine Konstante:

- Vergleich einer gepackten oder skalaren SingleReal aus einem XMM-Register mit einer solchen SingleReal aus einem XMM-Register

  ```
  CMPPS XMM, XMM, Const8
  CMPSS XMM, XMM, Const8
  ```

- Vergleich einer gepackten oder skalaren SingleReal aus einem XMM-Register mit einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister
  ```
  CMPPS XMM, Mem128, Const8
  CMPSS XMM, Reg32, Const8
  ```

COMISS
UCOMISS

Nicht immer möchte man das Ergebnis eines Vergleiches in Form einer Maske vorliegen haben, die dann weiterverarbeitet werden muss. Gut wäre es, auch einen Befehl zu haben, mit Hilfe dessen man analog der FPU-Befehle FCOM/FUCOM die Flags des condition code oder noch besser analog FCOMI/FUCOMI die Flags des Allzweckregisters anhand des Vergleichsergebnisses setzen lassen und somit Programmverzweigungen realisieren kann. Dies haben auch die Intel-Ingenieure gesehen und den Befehl COMISS und seinen Zwillingsbruder UCOMISS kreiert. Diese Befehle vergleichen erst zwei SingleReals und setzen dann, analog zur der Situation bei der FPU, Flags, anhand derer retrospektiv festgestellt werden kann, welche Beziehung zwischen den Werten der Operanden besteht.

Dabei ergibt sich jedoch ein kleines Problem: Es gibt nur das EFlags-Register des Prozessors, in dem Flags gesetzt werden können, da die XMM-Register ja mit den FPU-Registern nichts zu tun haben. Und auch das Setzen eines *condition codes*, wie bei der FPU, ist nicht möglich, da das zum FPU-StatusWord analoge MXCSR der XMM-Register einen solchen nicht kennt. Bliebe die Möglichkeit, anstelle der Masken von CMPPS eben die *condition codes* in das Zielregister einzutragen. Dies ist aber nicht sehr effektiv, da die Vergleichsbefehle ja in Verbindung mit Programmverzweigungen eingesetzt werden sollen, also erst einmal von einem XMM-Register in ein CPU-Register gelangen müssten – vorzugsweise in das EFlags-Register. Ferner müsste man dies mit vier SingleReals gleichzeitig machen. Im Hinblick auf eine möglichst schnelle Verarbeitung großer Datenmengen, wie sie unter SSE ja gefordert wird, nicht gerade das, was wir brauchen.

Daher haben beide Befehle eine Einschränkung: Sie wirken nur auf skalare SingleReals. Allerdings wird dieser »Nachteil« mit einem großen Vorteil eingekauft: Es werden in Abhängigkeit des Vergleiches Flags im EFlags-Register gesetzt, sodass unmittelbar in Form von Verzweigungen reagiert werden kann!

COMISS, *compare ordered scalar single-precision floating-point values*, und UCOMISS, *compare unordered scalar single-precision floating-point values*, vergleichen also die beiden Operanden und setzen in Abhängigkeit des

Vergleichsresultats folgende Flags im EFlags-Register der CPU, wie Tabelle 1.42 zeigt.

ZF	PF	CF	Ergebnis	Bemerkungen
0	0	0	größer	OF, AF und SF werden explizit gelöscht.
0	0	1	kleiner	
1	0	0	gleich	
1	1	1	ungeordnet	

Tabelle 1.42: Stellung einiger Condition Code im EFlags-Register der CPU und ihre Bedeutung bei den Befehlen COMISS und UCOMISS

Die Flagstellungen entsprechen denen nach einem Vergleich mittels FCOMI/FUCOMI bzw. CMP. Daher kann, wie dort, unmittelbar durch Auswertung der gesetzten Flags im Programm verzweigt werden, z.B. mit den Jxx-Befehlen.

Übrigens: Der einzige Unterschied zwischen COMISS und UCOMISS besteht darin, wie auf NaNs reagiert wird. So wird bei UCOMISS nur dann eine Exception ausgelöst, wenn einer der beiden Operanden eine sNaN ist. COMISS löst bei jeder NaN eine Exception aus. Somit reagieren sie auch in diesem Falle wie FCOMI/FUCOMI.

Operanden

Als Quelle des ersten Vergleichsoperanden kommt nur ein XMM-Register in Frage, während der zweite Vergleichspartner in einem XMM-Register oder einem Allzweckregister stehen kann (XXX steht für COMISS oder UCOMISS):

- Vergleich einer skalaren SingleReal aus einem XMM-Register mit einer solchen SingleReal aus einem XMM-Register
  `XXX XMM, XMM`
- Vergleich einer skalaren SingleReal aus einem XMM-Register mit einer skalaren SingleReal aus einem Allzweckregister
  `XXX XMM, Reg32`

Datenaustausch

Der Datenaustausch der XMM-Register mit dem Rest der (Prozessor-) Welt lässt sich auf verschiedene Weise vorstellen. Zum einen ist es existentiell, analog der FPU-Befehle über Instruktionen zu verfügen, die Daten aus dem Speicher in das XMM-Register schaufeln und umgekehrt. Diese Befehle könnten auch, wie im Falle der FPU-Befehle, Daten zwischen XMM-Registern verschieben. Solche Befehle gibt es auch: MOVAPS und MOVUPS sowie, lasst uns die skalaren SingleReals nicht vergessen, MOVSS.

Weiterhin könnte es interessant werden, Daten innerhalb eines XMM-Registers vertauschen zu können, so wie es z.B. der CPU-Befehl BSWAP macht. Auch dies wird mit zwei Befehlen realisiert: MOVLPS und MOVHPS. Wie man an den Endungen »-PS« bereits sieht, sind hiervon nur die PackedSingleReals betroffen. (Alles andere machte auch keinen Sinn: skalare SingleReals können ja per definitionem nur im niedrigstwertigen Teil des XMM-Registers residieren!)

Bleiben noch zwei Befehle, die wir oben als Erweiterung des MMX-Befehlssatzes kennen gelernt haben und die auch bei den XMM-Befehlen Sinn machen: die Extraktion des MSB der PackedSingleReals unter Bildung einer Maske und das »Mischen« SingleReals. Und auch diese Befehle gibt es: MOVMSKPS und SHUFPS. (Auch hier unnötig zu sagen: Das macht nur bei PackedSingleReals Sinn und nicht bei skalaren.)

Doch nun im Einzelnen:

MOVAPS
MOVUPS
MOVSS

Die Daten, die in die und aus den XMM-Registern geschaufelt werden sollen, können im Speicher an beliebigen Speicherstellen zum Liegen kommen. Schön, wenn der Programmierer immer auf alles achtet und sauber programmiert. Dann nämlich liegen alle PackedSingleReal-Datenstrukturen sauber an 16-Byte-Grenzen. Das sind Adressen, die ohne Restbildung durch 16, der Größe der Datenstruktur, teilbar sind. Diese höchst willkommene Anordnung der Datenstrukturen nennt man »ausgerichtet« oder angelsächsisch »aligned«.

Liegt diese Traumausgangssituation vor, kann MOVAPS, *move aligned packed single-precision floating-point value*, zum Einsatz kommen. Dieser Befehl ermöglicht den Datenaustausch mit dem Speicher oder innerhalb der XMM-Register, weshalb als Operanden genau diese Ziele und Quellen angegeben werden können. Einzige Einschränkung: Ein Operand muss ein XMM-Register sein!

Hat der Programmierer dagegen einmal wieder auf solche »Nebensächlichkeiten« nicht geachtet oder liegen andere widrige Gründe vor, kann MOVAPS nicht eingesetzt werden. Dann schlägt die Stunde von MOVUPS, *move unaligned packed single-precision floating-point* value. Es ist, wie gesagt, der absolute Zwilling von MOVAPS, nur dass eben nicht auf die Ausrichtung Wert gelegt wird. Dieser Befehl ist damit langsamer als sein Pendant, aber sicherer.

Auch skalare SingleReals können geladen, gespeichert und mit anderen XMM-Registern ausgetauscht werden. Verantwortlich hierfür ist MOVSS. Weiter gibt es nichts Besonderes zu sagen.

Operanden

Als Ziel des Kopiervorgangs kommt im Falle der gepackten wie skalaren Strukturen nur ein XMM-Register in Frage, während Quelle bei gepackten Strukturen ein XMM-Register oder eine Speicherstelle sein kann, bei skalaren ein XMM-Register oder ein Allzweckregister (XXX steht für MOVAPS oder MOVUPS):

- Kopieren einer gepackten oder skalaren SingleReal aus einem XMM-Register in ein XMM-Register
  ```
  XXX   XMM, XMM
  MOVSS XMM, XMM
  ```
- Kopieren einer gepackten SingleReal aus einer Speicherstelle oder einer skalaren SingleReal aus einem Allzweckregister in eine gepackte oder skalare SingleReal in ein XMM-Register
  ```
  XXX   XMM, Mem128
  MOVSS XMM, Reg32
  ```

MOVLPS
MOVHPS

Sollen lediglich zwei der vier möglichen SingleReals einer PackedSingleReal bewegt werden, ist auch dies möglich. Dabei ist zu unterscheiden, ob die beiden niedrigerwertigen SingleReals benutzt werden sollen oder die beiden höherwertigen. Dementsprechend gibt es zwei Befehle hierfür: MOVLPS, *move low packed single-precision floating-point values*, und MOVHPS, *move high packed single-precision floating-point values*. Sie machen genau das, was man ihrem Namen entsprechend erwartet: Laden von zwei SingleReals aus dem Speicher in den niedrigerwertigen Teil des XMM-Registers und zurück bzw. das Gleiche in den höherwertigen Teil.

```
MOVLPS
XMMx[031..000] := Mem[031..000])
XMMx[063..032] := Mem[063..032])
XMMx[095..064] := XMMx[095..064]; // unverändert
XMMx[127..096] := XMMx[127..096]; // unverändert

MOVHPS
XMMx[031..000] := XMMx[031..000]; // unverändert
XMMx[063..032] := XMMx[063..032]; // unverändert
XMMx[095..064] := Mem[031..000])
XMMx[127..096] := Mem[063..032])
```



Datenaustausch zwischen zwei XMM-Registern oder gar zwischen dem höherwertigen und niedrigerwertigen Teil eines oder verschiedener XMM-Register ist mit diesem Befehlen **nicht** möglich!

Operanden Als Ziel und Quelle des Kopiervorgangs kommt entweder ein XMM-Register oder eine Speicherstelle in Frage, wobei jeweils ein Operand ein XMM-Register und einer die Speicherstelle sein muss (XXX steht für MOVHPS oder MOVLPS):

- Kopieren einer »halben« gepackten SingleReal aus einem XMM-Register in eine Speicherstelle:
  `XXX Mem64, XMM`
- Kopieren einer »halben« gepackten SingleReal aus einer Speicherstelle in ein XMM-Register:
  `XXX XMM, Mem64`

MOVLHPS MOVHLPS Der Fall des Datenaustauschs einer halben PackedSingle zwischen zwei Registern ist den beiden Befehlen MOVLHPS, *move low to high packed single-precision floating-point value*, und MOVHLPS, *move high to low packed single-precision floating-point values*, vorbehalten. Sie kopieren entweder die niedrigerwertigen beiden SingleReals einer PackedSingleReal in die höherwertigen (MOVLHPS) oder umgekehrt (MOVHLPS). Hierbei ist der intraindividuelle wie auch der interindividuelle Austausch möglich (also entweder innerhalb eines XMM-Registers oder zwischen zwei):

```
MOVHLPS
XMMx[063..000] := XMMy[127..064]
XMMx[127..000] := XMMx[127..04]       // unverändert

MOVLHPS
XMMx[063..000] := XMMx[063..000]      // unverändert
XMMx[127..064] := XMMy[063..000])
```

Überflüssig darauf hinzuweisen, dass Austausch mit dem Speicher mit diesen Befehlen nicht möglich ist.

Operanden Als Ziel und Quelle des Kopiervorgangs kommt nur ein XMM-Register in Frage (XXX steht für MOVHLPS oder MOVLHPS):

```
XXX XMM, XMM
```

MOVMSKPS MOVMSKPS ist absolut identisch zu PMOVSKB, dem »Masken-Befehl«, der bei den MMX-Erweiterungen unter SSE bereits für PackedBytes besprochen wurde. Auch hier entnimmt der Prozessor jeder ge-

packten SingleReal das MSB, bei dem es sich ja um das Vorzeichen handelt, und baut daraus eine Maske, die in einem Allzweckregister der CPU abgelegt wird. Da es jedoch nur vier SingleReals in einer PackedSingleReal gibt, werden auch nur vier Bits codiert. Alle anderen werden auf 0 gesetzt:

```
Temp[0] := MMx[07]     // Signum der ShortReal mit Index 0
Temp[1] := MMx[15]     // Signum der ShortReal mit Index 1
Temp[2] := MMx[23]     // Signum der ShortReal mit Index 2
Temp[3] := MMx[31]     // Signum der ShortReal mit Index 3
Temp[4] := 0
Temp[5] := 0
Temp[6] := 0
Temp[7] := 0
Reg32[07..00] := Temp
Reg32[31..08] := 0
```

Operanden

Als Ziel für die Maske kommt nur ein Allzweck-Register in Frage, Quelle ist immer ein XMM-Register:

```
MOVMSKPS Reg32, XMM
```

SHUFPS

Auch diesen Befehl, *shuffle packed single-precision floating-point value*, kennen wir in Form seiner Integer-Variante aus der Besprechung der MMX-Erweiterungen unter SSE. Dort hieß er PSHUFW. Das Prinzip ist hier wie dort das gleiche:

Die Konstante, die dem Befehl zusätzlich zu den beiden Operanden als dritter Parameter mitgegeben wird, wird interpretiert als Feld mit vier Einträgen à 2 Bit:

```
Rule0 = Const[1..0]
Rule1 = Const[3..2]
Rule2 = Const[5..4]
Rule3 = Const[7..6]
```

Diese Bits werden als Ziffer interpretiert, die dadurch maximal Werte zwischen 0 und 3 annehmen kann. Sie sind die Indices in die PackedSingleReal in der Quelle, die an die festgelegten Stellen in der ZielPackedSingleReal kopiert werden sollen:

```
MMx[SingleReal(0)] := MMy[SingleReal(Rule0)]
MMx[SingleReal(1)] := MMx[SingleReal(Rule1)]
MMx[SingleReal(2)] := MMy[SingleReal(Rule2)]
MMx[SingleReal(3)] := MMx[SingleReal(Rule3)]
```

Auch hier ein Beispiel zur Verdeutlichung. Der Befehl SHUFPS XMM0, XMM5, 114 (114 = $72 = 01110010b = 01_11_00_10b = 1_3_0_2) würde folgende Registerbelegung bewirken:

```
XMM0[031..000] := XMM5[095..064]
XMM0[063..032] := XMM5[031..000]
XMM0[095..064] := XMM5[127..096]
XMM0[127..096] := XMM5[063..032]
```

Operanden Als Ziel für das Ergebnis und Quelle des ersten Misch-Partners kommt nur ein XMM-Register in Frage, während der zweite Misch-Partner in einem XMM-Register oder an einer Speicherstelle stehen kann. In jedem Fall gibt der dritte Operand den shuffle code an und ist eine Konstante:

- Mischen einer gepackten SingleReal aus einem XMM-Register mit einer solchen SingleReal aus einem XMM-Register
  `SHUFPS XMM, XMM, Const8`
- Mischen einer gepackten SingleReal aus einem XMM-Register mit einer solchen SingleReal aus einer Speicherstelle
  `SHUFPS XMM, Mem128, Const8`

UNPCKHPS UNPCKLPS UNPCKHPS und UNPCKLPS sind die PackedSingleReal-Pendants zu den ShortPackedInteger-Befehlen PUNPCKHBW, PUNPCKHWD und PUNPCKHDQ bzw. PUNPCKLBW, PUNPCKLWD und PUNPCKLDQ. Wie diese Befehle, die wir bereits bei den MMX-Befehlen kennen gelernt haben, »entpacken« auch UNPCKHPS und UNPCKLPS: In diesem Fall PackedSingleReals aus zwei Operanden in einen. Und das erfolgt ganz analog zu den MMX-Pendants, also auch mit unterschiedlichen Befehlen für die jeweiligen höherwertigen oder niedrigerwertigen Anteile der SingleReal:

```
UNPCKHPS:
XMMx[031..000] := XMMx[095..064]
XMMx[063..032] := XMMy[095..064]
XMMx[095..064] := XMMx[127..096]
XMMx[127..096] := XMMy[127..096]

UNPCKLPS:
XMMx[031..000] := XMMx[031..000]
XMMx[063..032] := XMMy[031..000]
XMMx[095..064] := XMMx[063..032]
XMMx[127..096] := XMMy[063..031]
```

Mehr ist eigentlich nicht zu sagen ...

Operanden

Als Ziel für das Ergebnis und Quelle des ersten Entpackungspartners kommt nur ein XMM-Register in Frage, während der zweite Entpackungspartner in einem XMM-Register oder an einer Speicherstelle stehen kann (XXX steht für UNPCKHPS oder UNPCKLPS):

- Entpacken einer gepackten SingleReal aus einem XMM-Register und einer solchen SingleReal aus einem XMM-Register:
  `XXX XMM, XMM`
- Entpacken einer gepackten SingleReal aus einem XMM-Register und einer solchen SingleReal aus einer Speicherstelle:
  `XXX XMM, Mem128`

Datenkonversion

Mit den Befehlen der Datenkonversion ist es möglich, skalare oder gepackte SingleReals in Integers oder gepackte Integers vom Typ LongInt (beide umfassen vier Bytes pro Element!) umzuwandeln und umgekehrt.

CVTPI2PS
CVTSI2SS
CVTPS2PI
CVTSS2SI

Hierzu gibt es jeweils zwei Befehlspaare: CVTPI2PS und CVTSI2SS konvertieren gepackte oder skalare LongInts in gepackte oder skalare SingleReals, während CVTPS2PI und CVTSS2SI den umgekehrten Vorgang ermöglichen. Nachdem für die SingleReals die XMM-Register heranzuziehen sind, ist die Frage, wo die konvertierten LongInts hergeholt oder hingebracht werden sollen. Aber diese Frage können Sie sich selbst beantworten! Neben der trivialen Lösung »Speicher« gibt es noch Register, deren Spezialität gepackte LongInts sind ...

Ja, wir haben hier die bislang einzigen Befehle, die einen Datenaustausch zwischen XMM- und MMX-Registern ermöglichen. (Und an dieser Stelle der Hinweis: Achten Sie nun im Folgenden sehr exakt auf das Vorhandensein des »X« im Mnemonic! Auch ich habe beim Schreiben geschwitzt.) Dies ist auch der Grund, warum mit diesen Befehlen jeweils »nur« zwei Daten konvertiert werden können: das MMX-Register umfasst nur 64 Bits und kann daher maximal zwei LongInts à vier Bytes aufnehmen.

```
CVTPI2PS:
XMMx[031..000] := SingleReal(MMy[031..000])
XMMx[063..032] := SingleReal(MMy[063..032])
XMMx[127..064] := XMMx[127..064]      // unverändert

CVTPS2PI
MMx[031..000] := LongInt(XMMy[031..000])
MMx[063..032] := LongInt(XMMy[063..032])
```

Interessant an diesen Befehlen ist, dass neben MMX-Registern auch Speicherstellen als Quelloperand in Frage kommen. Dies ist nicht bei jedem dieser Befehle so trivial, wie es zunächst vielleicht aussieht: *CVTPS2PI MM3, Var64Byte* konvertiert zwei SingleReals aus dem Speicher in zwei LongInts und legt sie im MMX-Register ab. Hierbei ist, obschon durch einen XMM-Befehl verursacht, kein XMM-Register involviert! Andererseits konvertiert *CVTPI2PS XMM5, Var64Byte* die in der Variable stehenden beiden LongInts und legt sie unter Umgehung des MMX-Registers gleich im spezifizierten XMM-Register ab.

Und noch eine Besonderheit: Da nach Intels Auffassung »skalare« ShortReals nicht viel mit »gepackten« zu tun haben und eher den FPU-Reals zuzuordnen sind als den XMM-Reals, sind die Quell- und Zielregister bei den »skalaren« Zwillingen der Konvertierungsbefehle nicht die MMX-Register, die für gepackte Integers zuständig sind, sondern Allzweckregister der CPU:

```
CVTSI2SS:
XMMx[031..000] := SingleReal(REG32[031..000])
XMMx[127..032] := XMMx[127..032]      // unverändert

CVTSS2SI
REG32[031..000] := LongInt(XMMy[031..000])
```

Und auch in diesem Fall gibt es den Sonderfall, dass das XMM-Register bei dieser Instruktion überhaupt nicht involviert ist. Dann nämlich, wenn eine skalare SingleReal direkt aus dem Speicher genommen und konvertiert werden soll. Dann wird sie direkt im Allzweckregister abgelegt: *CVTSS2SI EBX, Var32Byte*.

Bleibt noch eine »Kleinigkeit« zu klären! LongInts haben einen Wertebereich von $\pm 4.29 \cdot 10^{10}$ (oder exakt: 4,294,967,295), SingleReals von $\pm 3{,}675252 \cdot 10^{38}$. Die Konvertierung einer LongInt in eine SingleReal ist damit von der Größenordnung her kein Problem: Der Wertebereich der LongInt ist vollständig im Wertebereich der SingleReal enthalten. Probleme aber gibt es im umgekehrten Fall: Ist der absolute Wert der SingleReal größer als die absolut maximal darstellbare LongInt, so ist die SingleReal nicht mehr konvertierbar!

Das nächste Problem ist, was man mit den Nachkommaanteilen tut, die Realzahlen ja nun einmal aufgrund ihrer Definition haben können (und in der Regel auch haben, sonst könnte man ja gleich mit Integers rechnen). Werden die einfach abgeschnitten? Wird gerundet? Und, wenn ja: abwärts oder aufwärts?

Und um die Problematik nicht zu klein bleiben zu lassen, ein drittes Problem! Da SingleReals größere Wertebereiche haben als LongInts, aber wie diese nur 32 Bits zur Darstellung benötigen, muss noch ein Pferdefuß existieren, der bei der Konvertierung eine Rolle spielen könnte. Und den gibt es auch tatsächlich, er wird leider meistens vergessen oder verdrängt, zumindest aber nicht berücksichtigt: Genauigkeit. Wenn man einer Zahl 8 Bits klaut, um ihr einen Exponenten zu spendieren, mit der der Wertebereich ausgedehnt werden kann, so kann das nur auf Kosten der Genauigkeit gehen, die damit um 8 Bits kleiner wird. Und so ist es auch: Während LongInts zehn signifikante Stellen besitzen (bei dezimaler Betrachtung, bei binärer natürlich 32!) haben SingleReals »nur noch« acht.

Wer nun »Na und?« ruft und glaubt, dass das mehr als genug sei, glaubt auch, dass die Quadratwurzel aus dem Quadrat des Natürlichen Logarithmus von e hoch 2 = 1.999999999 ist und sollte ein wenig nachdenken! Denn die größte LongInt heißt 4,294,967,295 oder $4.294967295 \cdot 10^{10}$ in Realzahldarstellung. Und nun zählen wir acht signifikante Stellen ab: $4.2949672 \cdot 10^{10}$. Weitere Nachkommastellen *kann* der Rechner *nicht* darstellen! Das aber bedeutet: jede LongInt mit mehr als acht Stellen ist als SingleReal nicht mehr exakt darstellbar! So ist jeder Wert zwischen 4,294,967,200 und 4,294,967,299 der gleiche: 4,294,967,200. Auch 198,235,742 ist »nur« 198,235,740, genauso wie 198,235,749. Summa: Nur Integers, die innerhalb der Genauigkeitsgrenze liegen, die durch die Anzahl der möglichen Stellen der Realzahl vorgegeben ist, können auch exakt konvertiert werden. Und das sind bei SingleReals eben 24 binäre bzw. 8 dezimale Stellen.

Was folgt nun aus diesen drei Problemen? Der Prozessor muss irgendwie mit diesen Möglichkeiten umgehen können. Er muss also wissen, wie er sich beim Auftreten von Über- bzw. Unterschreitungen zu verhalten hat. Und wie Sie gesehen haben, ist das nicht nur beim offensichtlichen Überschreiten der maximal darstellbaren LongInt bei der Konvertierung SingleReal ➞ LongInt der Fall, sondern auch beim subtileren Fall der Konvertierung einer LongInt mit mehr als acht signifikanten Stellen in eine SingleReal.

Ihm dies klarzumachen, besitzt der XMM-Registersatz das Feld »Rounding Control«, also die Bits 14 und 13 des MXCS-Registers. Sie codieren ein Kontrollfeld analog der FPU, mit dem die Art der Rundung vorgegeben wird (vgl. Seite 361).

Fehlt noch zu nennen, was der Prozessor in dem Falle tut, wenn der Wert der zu konvertierenden Zahl den Wertebereich des Zielformates überschreitet (was ja nur bei der Konvertierung einer SingleReal in eine LongInt möglich ist). In diesem Fall wird einfach eine »undefinierte« (»indefinite«) Integer zurückgegeben. Nachdem es ja bei Integers nicht so sehr viele Möglichkeiten gibt, diese darzustellen, benutzt man dazu $80000000, also die »Null« mit negativem Vorzeichen.

Operanden Als Ziel bei der Konvertierung einer gepackten SingleReal kommt nur eine ShortPackedInteger und somit ein MMX-Register in Frage, bei skalaren SingleReals eine LongInt und daher ein Allzweckregister. Quelle kann eine »halbe« gepackte SingleReal in einem XMM-Register oder an einer Speicherstelle sein (CVTPS2PI), oder eine skalare SingleReal in einem XMM-Register oder ebenfalls an einer Speicherstelle (CVTSS2SI). Umgekehrt ist bei der Konvertierung einer gepackten Integer das Ergebnis eine gepackte oder skalare SingleReal, Ziel also immer ein XMM-Register; die Integer kann hierbei in Form einer ShortPacked-Integer in einem MMX-Register oder an einer Speicherstelle vorliegen (CVTPI2PS) oder als LongInt in einem Allzweckregister oder ebenfalls an einer Speicherstelle (CVTSI2SS):

- Konvertierung einer gepackten bzw. skalaren SingleReal aus einem XMM-Register in eine gepackte Integer in einem MMX-Register bzw. eine LongInt in einem Allzweckregister
  ```
  CVTPS2PI MMX, XMM
  CVTSS2SI Reg32, XMM
  ```
- Konvertierung einer gepackten SingleReal bzw. einer skalaren SingleReal aus einer Speicherstelle in eine gepackte Integer in einem MMX-Register bzw. eine LongInt in einem Allzweckregister
  ```
  CVTPS2PI MMX, Mem64
  CVTSS2SI Reg32, Mem32
  ```
- Konvertierung einer gepackten Integer aus einem MMX-Register oder einer LongInt in einem Allzweckregister in eine gepackte oder skalare SingleReal in einem XMM-Register
  ```
  CVTPI2PS XMM, MMX
  CVTSI2SS XMM, Reg32
  ```
- Konvertierung einer gepackten Integer oder einer LongInt aus einer Speicherstelle in eine gepackte oder skalare SingleReal in einem XMM-Register
  ```
  CVTPI2PS XMM, Mem64
  CVTSI2SS XMM, Mem32
  ```

XMM-Verwaltung

Die »Verwaltung« der XMM-Erweiterungen unter SSE beschränkt sich auf das MXCS-Register, das ja für die Rundung bei Datenkonvertierung, aber auch für das Maskieren von Exceptions und deren Verwaltung zuständig ist. Dazu gibt es zwei Instruktionen:

LDMXCSR STMXCSR

LDMXSR lädt, wie der Name *load MXCSR* vermuten lässt, ein Doppelwort aus dem Speicher in das MXCSR (vgl. Seite 361), während *store MXCSR* genau das Gegenteil tut. Diese Instruktionen sind daher für eine weitere Besprechung ebenso spannend, wie es FLDCW/FSTCW bei der Besprechung der FPU-Befehle war.

Operanden

LDMXCSR und STMXCSR haben je einen impliziten und expliziten Operanden. Der implizite Operand ist in beiden Fällen das MXCS-Register, der explizite eine Speicherstelle. Bei LDMXCSR ist der implizite Operand Ziel der Operation, die Quelle die explizit anzugebende Speicherstelle. Bei STMXCSR dagegen wird der Wert aus dem implizit angegebenen Quelloperanden an die explizit bezeichnete Speicherstelle übertragen. Die Befehle werden daher wie folgt aufgerufen:

```
LDMXCSR Mem32
STMXCSR Mem32
```

Optimierung der Datenströme

Um die folgenden Befehle besser einordnen zu können, sollte zunächst einmal ein kleiner Ausflug in die Welt der Datenströme unternommen werden. Generell kann man Daten in zwei große Gruppen aufteilen: Daten, die lediglich einmal benötigt werden, wie z.B. Daten in Multimedia-Anwendungen, wo es nur darauf ankommt, die Videosequenz auf den Bildschirm und die Geräusche in die Lautsprecher zu bekommen; und Daten, die man häufiger benötigt, wie z.B. Programmcode (der für den Prozessor ja auch lediglich aus Daten für die eigenen Instruktionen besteht). Letztere nennt man »temporale« Daten (»temporal«, engl. = zeitlich, soll heißen »über eine gewisse Zeit verfügbar«, nicht zu verwechseln mit »temporary«, engl. = temporär, vorübergehend), die ersteren »nicht-temporal«.

Nun wissen wir alle, dass seit vielen Prozessorgenerationen viel Schweiß in die Entwicklung von Mechanismen gesteckt wurde, temporale Daten möglichst schnell und effizient verfügbar zu machen: Die Performance eines Prozessors hängt nicht zuletzt davon erheblich ab, wie schnell der Prozessor an seine Daten kommt. Dieses Blut und dieser Schweiß endeten (vorläufig) in der Bereitstellung von mehr oder weniger aufwändigen Pufferungsmechanismen mit mehr oder weniger auf-

wändigen Strukturen. Jeder hat vermutlich von den *first level caches* und *second level caches* gehört und vielleicht bereits damit die eine oder andere unliebsame Erfahrung gemacht (so wie auch ich, der vor Jahren bei einem Pentium 166 MHz nach einer Speichererweiterung von 64 MB auf 128 MB zu seiner Verblüffung feststellen musste, dass sein Rechner nun erheblich langsamer lief als das Vorgängermodell mit 120 MHz und 48 MB RAM. Grund: Der implementierte Cache war nicht zur Zusammenarbeit mit mehr als 64 MB ausgelegt und wurde nach der in den Datenblättern ausdrücklich erlaubten Aufrüstung einfach abgeschaltet ...) Diese Caches speichern nach ausgeklügelten Mechanismen die Daten, die häufig benötigt werden, in speziellen, sehr schnellen Speichern. So kann verhindert werden, dass der Prozessor immer in den relativ trägen RAM schauen und sich dort bedienen muss.

Es ist offensichtlich, dass die Effektivität dieser Puffermechanismen sehr davon abhängt, mit welchen Daten gedealt wird. Wird durch den Cache ein Datenstrom aus nicht-temporalen Daten gejagt, so kann man seine Funktion getrost vergessen: Da gibt es nichts zu puffern. Man spricht in diesem Fall von »Cache-Verschmutzung« (»cache pollution«), da sinnvollerweise zu puffernde Daten nicht mehr gepuffert werden können. Das aber wiederum heißt, dass der Cache umso effektiver ist, je weniger Multimediadaten durch ihn geschleust werden müssen! Eine ernüchternde Erkenntnis!

Wie passt das mit der Forderung von oben zusammen, nach der doch gerade für Multimedia ein besonders schneller Datentransfer möglich sein soll? Die SSE-Erweiterungen der Prozessoren tragen dieser Problematik in mehrfacher Weise Rechnung. Ich kann und will an dieser Stelle nicht in Details gehen, da dies in erheblichem Maße den Rahmen dieses Buches sprengen würde, verweise daher alle Interessierten auf Sekundärliteratur und belasse es bei einer sehr kurzen Besprechung der Befehle, die SSE zu diesem Zweck zur Verfügung stellt, nur um Ihnen eine Idee zu geben, wie die beiden Randbedingungen unter einen Hut gebracht werden können.

MOVNTQ
MOVNTPS
MASKMOVQ

Diese Befehle veranlassen den Prozessor, QuadWords (MOVNTQ; *move a non-temporal quadword*) oder einzelne Bytes (MASKMOVQ; *move quadword by mask*) aus MMX- bzw. PackedSingleReals (MOVNTPS; *move a non-temporal packed single-precision floating-point value*) aus XMM-Registern in den Speicher zu schreiben. Dabei wird ihm nahe gelegt, möglichst nicht den Cache zu benutzen; vielmehr wird dieser, falls erforderlich, vorher zwangsweise geleert.

Während MOVNTQ und MOVNTPS lediglich Variationen des MOV-Befehls sind, die ausschließlich ganze MMX- (MOVNTQ) oder XMM-Register (MOVNTPS) in den Speicher schreiben (und nur hier macht ein non-temporal writing Sinn!), ist MASKMOVQ ein komplizierterer Befehl. Ihm wird eine Maske übergeben, in der das most significant Bit (MSB) jedes Bytes darüber entscheidet, ob das dazugehörige Byte im Quelloperanden in das Ziel kopiert wird oder das betreffende Zielbyte unverändert bleibt:

```
IF Mask[07] = 1 THEN Dest[07..00] := Source[07..00]
IF Mask[15] = 1 THEN Dest[15..08] := Source[15..08]
IF Mask[23] = 1 THEN Dest[23..16] := Source[23..16]
IF Mask[31] = 1 THEN Dest[31..24] := Source[31..24]
IF Mask[39] = 1 THEN Dest[39..32] := Source[39..32]
IF Mask[47] = 1 THEN Dest[47..40] := Source[47..40]
IF Mask[55] = 1 THEN Dest[55..48] := Source[55..48]
IF Mask[63] = 1 THEN Dest[63..56] := Source[63..56]
```

Operanden

MASKMOVQ hat, wie gesehen, drei Operanden: einen impliziten und zwei explizite. Der implizite Operand ist der Zieloperand (Dest); es handelt sich um eine Speicherstelle, die in der Adressierungsregister-Kombination DS:(E)DI angegeben ist. Diese Adresse muss auf eine Mem64 zeigen. Der zweite und somit erste explizit angegebene Operand ist die Quelle (Source); bei ihr handelt es sich immer um ein MMX-Register. Auch der dritte und damit als zweites explizit angegebene Operand muss ein MMX-Register sein, da die Maske (Mask) enthält. MASKMOVQ wird somit wie folgt aufgerufen:

```
MASKMOVQ MMX, MMX
```

MOVNTQ und MOVNTPS sind verwandte Befehle, die ein MMX- bzw. XMM-Register auslesen und an eine Speicherstelle kopieren. Daher ist der jeweils erste oder Zieloperand eine Speicherstelle, der zweite oder Quelloperand entweder ein MMX- (MOVNTQ) oder XMM-Register (MOVNTPS):

```
MOVNTQ  Mem64, MMX
MOVNTPS Mem128, XMM
```

PREFETCHTx

Mit PREFETCH werden Daten kontrolliert in die verschiedenen Ebenen der Cache-Technologie gesteckt. Dadurch ist es möglich, anhand der zu erwartenden Daten die Cache-Strukturen optimal zu nutzen und eine Cache-Verunreinigung weitgehend zu verhindern.

Den PREFETCH-Befehl gibt es in vier Versionen: drei für temporale Daten (PREFETCHTx), bei denen zusätzlich angegeben werden kann, ab welcher Hierarchiestufe des Caches die Daten verfügbar sein sollten, und einer (PREFETCHNTA) für nicht-temporale Daten. Man unterscheidet daher

- T0 (PREFETCHT0): benutze alle cache levels
- T1 (PREFETCHT1): benutze alle cache levels außer level 0
- T2 (PREFETCHT2): benutze alle cache levels außer level 0 bis 1
- NTA (PREFETCHNTA): umgehe den cache und nutze Cache-Strukturen für nicht-temporale Daten.

In der Praxis heißt das für Prozessoren ab dem Pentium III (zumindest bis zum Pentium 4):

- PREFETCHT0 transferiert die Daten je nach Bedarf in die cache levels 1 und/oder 2
- PREFETCHT1 transferiert die Daten in den niedrigsten level 2
- PREFETCHT2 transferiert ebenfalls die Daten in cache level 2 und
- PREFETCHNTA transferiert die Daten möglichst »nahe« an den Prozessor und umgeht »darunter« liegende Strukturen: cache level 1.

Operanden PREFETCHTx hat einen Operanden, der auf eine Byte-Speicherstelle zeigen muss. Daher können alle PREFTECH-Varianten nur wie folgt aufgerufen werden:

```
PREFETCHTx Mem8
```

SFENCE SFENCE, *store fence*, nimmt Einfluss auf den Datenfluss, indem es Bereiche mit unterschiedlichen Arten der Datenspeicherung sauber voneinander trennt, indem es »Zäune«, engl. »fences«, aufbaut. Dadurch wird gewährleistet, dass alle Daten, die vor einem »Zaun« geschrieben wurden, hinter dem »Zaun« tatsächlich global verfügbar sind. Zu Einzelheiten der Datenspeicherung wird auf Sekundärliteratur verwiesen.

Operanden SFENCE hat keine Operanden und wird daher wie folgt benutzt:

```
SFENCE
```

1.3.5 SIMD, die Dritte: SSE2

SSE2

SSE2 nun heißt die logische Fortentwicklung dessen, was mit MMX und SSE einmal begonnen wurde. Um es kurz zu machen: Die Veränderungen, die SSE2 einführt, laufen auf eine »Vereinheitlichung« aller Strukturen und Möglichkeiten hinaus, die MMX und SSE in unterschiedlicher Weise eingeführt haben, bei gleichzeitigem »Aufbohren« aller Datenstrukturen auf 128 Bit.

Ja, Sie haben richtig gelesen: Unter SSE2 gibt es nun nicht nur 128-Bit-Reals, sondern eben auch 128-Bit-Integers – wenn auch in beiden Fällen »nur« gepackt. Doch genauer:

SSE2-Datenformate

SSE2 definiert fünf (!) neue Datenformate. Vier davon dienen der Bezeichnung von PackedIntegers – wir werden sie gar nicht erst erwähnen, da Intel sie sehr »elegant« nur um den Präfix »128-« erweitert hat, ansonsten aber die gleichen Bezeichnungen verwendet hat wie bei den Short-Versionen. Anders die *packed double-precision floating-point values*, die logische Inflation der PackedSingleReals. Sie werden gemäß Tabelle 5.23 auf Seite 844 als PackedDoubleReals bezeichnet und bilden mit den PackedSingleReals die Familie der PackedReals.

Auch die ShortPackedIntegers erlebten, wie gesagt, unter SSE2 eine Inflation zu den PackedIntegers, die aus den gleichen Elementen wie die analogen Short-Versionen bestehen, nur dass sie eben doppelt so viele enthalten.

XMM-Register

Und da nun mit 128 Bit auch zwei QuadWords in ein Register passen, wurden die PackedIntegers noch um die PackedQuads, also zwei PackedQWords oder zwei PackedQInts, erweitert – je nach Vorhandensein des Vorzeichens. Eine Zusammenfassung zeigt Tabelle 5.23 auf Seite 844. Abbildung 1.37 zeigt die XMM-Registerbelegung unter Berücksichtigung der neuen Integer-Datenformate. In der Abbildung fasst XMM0 ein DoubleQuadWord, XMM1 ein PackedQuadWord, XMM2 ein PackedDobuleWord, XMM3 ein PackedWord und XMM4 ein PackedByte. Analog sind auch vorzeichenbehaftete PackedIntegers darstellbar, von denen hier jedoch nur die PackedQuadInts(XMM5), PackedLongInts (XMM6) und PackedIntegers (XMM7) gezeigt werden.

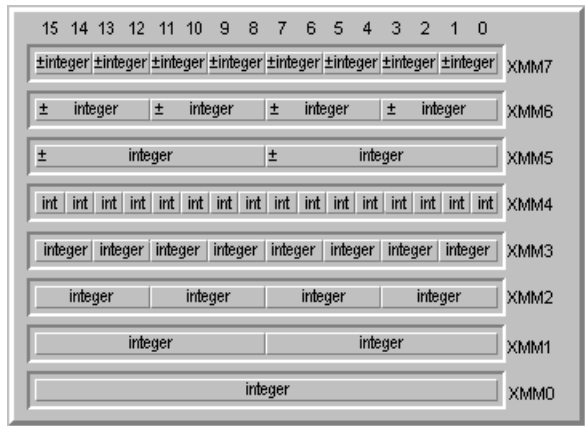


Abbildung 1.37: Speicherabbild der XMM-Register mit Integers im Rahmen Erweiterung des SIMD-Befehlssatzes unter SSE2

Abbildung 1.38 zeigt, dass im Vergleich zu SSE (siehe Abbildung 1.36) lediglich die neuen Fließkomma-Datenformate ScalarDouble und PackedDouble hinzugekommen sind.

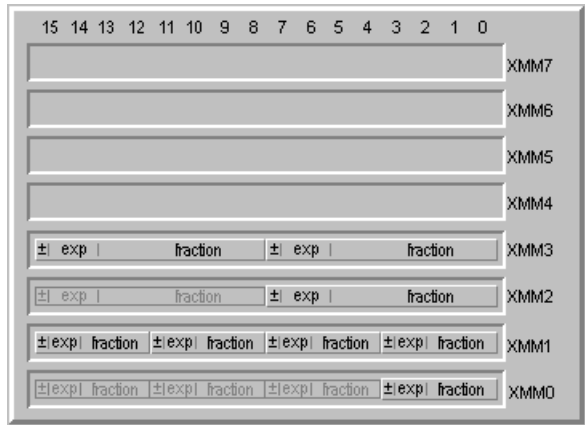


Abbildung 1.38: Speicherabbild der XMM-Register mit Realzahlen im Rahmen der Erweiterung des SIMD-Befehlssatzes unter SSE2

SSE2-Befehle

Was nun die »neuen« Befehle unter SSE2 betrifft, so kann man folgende Vermutungen anstellen. Zunächst werden alle SSE-Befehle, die mit PackedSingles oder ScalarSingles arbeiten, auf PackedDoubles und ScalarDoubles ausgedehnt werden. Das betrifft dann alle arithmetischen, logischen, vergleichenden, mischenden und konvertierenden Befehle sowie die Datenaustausch-Instruktionen. Damit wäre dann ein

»globaler« Befehlssatz für alle gepackten und skalaren Reals verwendbar, die in den 128-Bit-XMM-Registern verwaltet werden können. Eine Liste der unter SSE2 verfügbaren Instruktionen mit Fließkommazahlen zeigt Tabelle 5.26 auf Seite 846.

Dann dürfte eine »Vereinheitlichung« aller unter SSE erfolgten Anpassungen der Integer-Instruktionen erfolgen, sodass auch hier ein »globaler« Integer-Befehlssatz resultiert, der zum einen die 64-Bit-MMX-, zum anderen die 128-Bit-XMM-Register benutzt. Auch diese Befehle sind in Tabelle 5.26 auf Seite 846 zusammengestellt.

Schließlich wird es eine Erweiterung der »Verwaltungsbefehle« geben, die den 128-Bit-Datenstrukturen mit DoubleReals Rechnung tragen.

XMM-Befehlssatz

Und so ist es auch: Analog der Einteilung unter SSE im vorherigen Abschnitt lassen sich die Befehle, die mit den neuen gepackten Fließkommazahlen vom Typ PackedReal arbeiten, in die folgenden Klassen einteilen:

- arithmetisches Manipulieren der Daten
- logisches Manipulieren der Daten
- Datenvergleich
- Datenaustausch
- Datenkonversion

Arithmetische Befehle

Wie mit den PackedSingles und den ScalarSingles können auch mit den PackedDoubles und den ScalarDoubles Additionen (ADDPD, ADDSD), Subtraktionen (SUBPD, SUBSD) Multiplikationen (MULPD, MULSD) und Divisionen (DIVPD, DIVSD) durchgeführt sowie Quadratwurzeln (SQRTPD, SQRTSD) gebildet und Maxima (MAXPD, MAXSD) und Minima (MINPD, MINSD) bestimmt werden. Die Befehle arbeiten absolut analog zu den bereits unter SSE besprochenen PackedSingle/ScalarSingle-Befehlen, sodass auf eine weitere Besprechung verzichtet werden kann.

Leider hat die SSE2-Erweiterung einen Mangel. Die Bildung der Reziprokwerte sowie der reziproken Quadratwurzeln ist mit PackedDoubles und ScalarDoubles nicht möglich, die Befehle existieren offensichtlich nicht! Intel allein weiß, warum nicht.

Logische Befehle

Auch hier gibt es nichts Neues zu berichten! Die von PackedSingles her bekannten Befehle für die AND-, AND-NOT-, OR- und XOR-Operationen gibt es auch für PackedDoubles. Hier heißen sie ANDPD, AND-

NPD, ORPD und XORPD und verhalten sich absolut identisch zu den PackedSingles-Zwillingen.

Datenvergleich

Auch die den Datenvergleichsbefehlen für PackedSingles analogen Instruktionen für PackedDoubles gibt es: CMPPD und CMPSD arbeiten hier wie dort prospektiv mit Prädikaten, was bedeutet, dass vor dem Vergleich die Art des Vergleiches bekannt sein muss und über das Prädikat dem jeweiligen Befehl mitgeteilt werden muss. Das Ergebnis ist auch hier eine Maske, in der alle 64 Bits der DoubleReals der Felder gesetzt sind oder, falls die Bedingung nicht zutrifft, gelöscht sind. COMISD und UCOMISD arbeiten analog zu COMISS und UCOMISS retrospektiv, was bedeutet, dass durch den Vergleich Flags im EFlags-Register gesetzt werden, die für Programmverzweigungen benutzt werden können.



Bitte beachten Sie, dass es einen »Namenskonflikt« gibt! CMPSD ist das Mnemonic für einen Befehl, der zwei skalare DoubleWords vergleicht, jedoch wird dieses Mnemonic seit dem 80386 auch für eine Erweiterung des Stringbefehls CMPS auf DoubleWords als Operanden verwendet (vgl. Seite 132). Ein echter Konflikt ist das jedoch nicht, da der Assembler anhand der übergebenen Operanden feststellen kann, ob nun die String- oder die XMM-Variante benutzt werden soll. Und auch für den Programmierer dürfte dies anhand des Programmkontextes ziemlich eindeutig sein.

Datenaustausch

Natürlich verhalten sich die Datenaustausch-Instruktionen ebenfalls absolut in line! Mit MOVAPD, MOVUPD, MOVSD, MOVHPD, MOVLPD und MOVNTPD haben wir die analogen Datenschaufeln für PackedDoubles und ScalarDoubles. Einzig die »high-low«-Austauscher MOVLHPS und MOVHLPS besitzen kein PackedDouble-Pendant – es hätte auch wenig Sinn!



Bitte beachten Sie, dass es einen »Namenskonflikt« gibt! MOVSD ist das Mnemonic für einen Befehl, der den Transfer von skalaren DoubleWords in ein und aus einem XMM-Register bewerkstelligt, jedoch wird dieses Mnemonic seit dem 80386 auch für eine Erweiterung des Stringbefehls MOVS auf DoubleWords als Operanden verwendet (vgl. Seite 132). Ein echter Konflikt ist das jedoch nicht, da der Assembler anhand der übergebenen Operanden feststellen kann, ob nun die String- oder

die XMM-Variante benutzt werden soll. Und auch für den Programmierer dürfte dies anhand des Programmkontextes ziemlich eindeutig sein.

Mit MOVMSKPD, UNPCKHPD, UNPCKLPD und SHUFPD haben wir die Analoga der verbleibenden Datenaustausch-Befehle für gepackte Realzahlen vom Typ DoubleReal. Auch bei diesen Instruktionen ist nichts Neues hinzugekommen.

Datenkonvertierung

Lediglich bei der Datenkonvertierung hat sich einiges getan. So gibt es nun insgesamt 22 Instruktionen, die Daten von einem Format in ein anderes überführen können. Vier davon wurden bereits unter SSE besprochen. Sie ermöglichen die Konvertierung von skalaren oder gepackten LongInts in skalare oder gepackte SingleReals.

CVTSD2SI
CVTSI2SD
CVTPD2PI
CVTPI2PD

Diese vier Befehle sind nun die Analoga der SingleReal-Konvertierungs-Befehle CVTSS2SI, CVTSI2SS, CVTPS2PI und CVTPI2PS für DoubleReals und ermöglichen somit ebenfalls die Konvertierung von skalaren und gepackten Realzahlen in skalare oder gepackte LongInts und umgekehrt. Bitte beachten Sie, dass bei diesem Vorgang nicht nur die Zahlenart gewechselt wird (Real ↔ Integer) sondern auch die verwendete Datengröße (8-Byte-Real ↔ 4-Byte-Integer):

```
CVTSI2SD:
XMMx[063..000] := DoubleReal(REG32[031..000])
XMMx[127..064] := XMMx[127..032]      // unverändert

CVTSD2SI
REG32[031..000] := LongInt(XMMy[063..000])
```

Auch bei diesen Befehlen erfolgt der Datenaustausch zwischen dem XMM-Register und einem Allzweckregister (oder einer Speichervariablen), sofern skalare Daten betroffen sind, bzw. zwischen XMM- und MMX-Register (oder dem Speicher) im Falle von gepackten Zahlen:

```
CVTPI2PD:
XMMx[063..000] := DoubleReal(MMy[031..000])
XMMx[127..064] := DoubleReal(MMy[063..032])

CVTPD2PI
MMx[031..000] := LongInt(XMMy[063..000])
MMx[063..032] := LongInt(XMMy[127..063])
```

Alles in allem nichts Aufregendes! Vielleicht nur das: Da die DoubleReal über mehr signifikante Stellen als die SingleReal verfügt und in beiden Fällen die Konversion lediglich in und aus LongInts erfolgt, kann mit diesen Befehlen eine LongInt erstmals vollständig und exakt in eine Realzahl konvertiert werden und umgekehrt, so die Realzahl nicht noch zusätzliche Nachkommastellen hat.

Operanden Als Ziel bei der Konvertierung einer gepackten DoubleReal kommt nur eine ShortPackedInteger und somit ein MMX-Register in Frage, bei skalaren DoubleReals eine LongInt und daher ein Allzweckregister. Quelle kann eine gepackte DoubleReal in einem XMM-Register oder an einer Speicherstelle sein (CVTPD2PI), oder eine skalare DoubleReal in einem XMM-Register oder ebenfalls an einer Speicherstelle (CVTSD2SI). Umgekehrt ist bei der Konvertierung einer ShortPackedInteger das Ergebnis eine gepackte oder skalare DoubleReal, Ziel also immer ein XMM-Register; die Integer kann hierbei in Form einer gepackten Integer in einem MMX-Register oder an einer Speicherstelle vorliegen (CVTPI2PD) oder als LongInt in einem Allzweckregister oder ebenfalls an einer Speicherstelle (CVTSI2SD):

- Konvertierung einer gepackten bzw. skalaren DoubleReal aus einem XMM-Register in gepackte Integer in einem MMX-Register bzw. eine LongInt in einem Allzweckregister
  ```
  CVTPD2PI MMX, XMM
  CVTSD2SI Reg32, XMM
  ```
- Konvertierung einer gepackten bzw. einer skalaren DoubleReal aus einer Speicherstelle in eine gepackte Integer in einem MMX-Register bzw. eine LongInt in einem Allzweckregister
  ```
  CVTPD2PI MMX, Mem128
  CVTSD2SI Reg32, Mem64
  ```
- Konvertierung einer gepackten Integer aus einem MMX-Register oder einer LongInt in einem Allzweckregister in eine gepackte oder skalare DoubleReal in einem XMM-Register
  ```
  CVTPI2PD XMM, MMX
  CVTSI2SD XMM, Reg32
  ```
- Konvertierung einer gepackten Integer oder einer LongInt aus einer Speicherstelle in eine gepackte oder skalare DoubleReal in einem XMM-Register
  ```
  CVTPI2PD XMM, Mem64
  CVTSI2SD XMM, Mem32
  ```

CVTPS2DQ
CVTDQ2PS
CVTPD2DQ
CVTDQ2PD

Neu dagegen sind vier weitere Befehle, die die Konvertierung von gepackten Realzahlen im XMM-Register nicht in die ShortPackedIntegers der MMX-Register übernehmen, sondern in PackedInteger-Strukturen der XMM-Register und damit die Daten »zu Hause« lassen. Auch in diesem Fall wird eine 4-Byte-Integer in eine 8-Byte-Real überführt und umgekehrt. Beachten Sie bitte, dass im Falle der DoubleReals nur zwei von vier möglichen Integer-Plätzen in den gepackten Strukturen benutzt werden (können).

```
CVTPS2DQ:
XMMx[031..000] := LongInt(XMMy[031..000])
XMMx[063..032] := LongInt(XMMy[063..032])
XMMx[095..064] := LongInt(XMMy[095..064])
XMMx[127..096] := LongInt(XMMy[127..096])

CVTPD2DQ:
XMMx[031..000] := LongInt(XMMy[063..000])
XMMx[063..032] := LongInt(XMMy[127..064])
XMMx[127..064] := 0;

CVTDQ2PS:
XMMx[031..000] := SingleReal(XMMy[031..000])
XMMx[063..032] := SingleReal(XMMy[063..032])
XMMx[095..064] := SingleReal(XMMy[095..064])
XMMx[127..096] := SingleReal(XMMy[127..096])

CVTDQ2PD:
XMMx[063..000] := DoubleReal(XMMy[031..000])
XMMx[127..064] := DoubleReal(XMMy[063..032])
```

Die Namensgebung ist in meinen Augen nicht ganz glücklich! DQ ist ein DoubleQuadWord und soll damit anzeigen, dass die gesamten 128 Bit des XMM-Registers betroffen sind. In Wirklichkeit jedoch wird nicht aus zwei Realzahlen ein DoubleQuadWord gemacht oder umgekehrt, sondern aus zwei bzw. vier Realzahlen zwei bzw. vier Integer und umgekehrt, die Teil einer Packed-Struktur sind. Aber sei es drum!

Operanden

Bei den folgenden Befehlen kann die Quelle entweder eine Speicherstelle sein oder ein XMM-Register. Ziel ist in jedem Fall ein XMM-Register:

- Konvertierung einer gepackten SingleReal oder DoubleReal aus einem XMM-Register in eine PackedInteger in einem XMM-Register
  ```
  CVTPS2DQ XMM, XMM
  CVTPD2DQ XMM, XMM
  ```

- Konvertierung einer gepackten SingleReal oder DoubleReal aus einer Speicherstelle in eine PackedInteger in einem XMM-Register
  ```
  CVTPS2DQ XMM, Mem128
  CVTPD2DQ XMM, Mem128
  ```
- Konvertierung einer PackedInteger aus einem XMM-Register in eine gepackte SingleReal oder DoubleReal in einem XMM-Register
  ```
  CVTDQ2PS XMM, XMM
  CVTDQ2PD XMM, XMM
  ```
- Konvertierung einer PackedInteger an einer Speicherstelle in eine gepackte SingleReal oder DoubleReal in einem XMM-Register
  ```
  CVTDQ2PS XMM, Mem128
  CVTDQ2PD XMM, Mem128
  ```

CVTSS2SD
CVTSD2SS
CVTPS2PD
CVTPD2PS

Natürlich lassen sich auch SingleReals in DoubleReals überführen und umgekehrt. Zuständig hierfür sind die Befehle CVTSS2SD und CVTSD2SS, die die Konvertierung von skalaren Daten übernehmen, sowie CVTPS2PD und CVTPD2PS, die das Gleiche mit gepackten Strukturen ermöglichen. In jedem Fall sind die XMM-Register involviert:

```
CVTSS2SD:
XMMx[063..000] := DoubleReal(XMMy[031..000])
XMMx[127..064] := XMMx[127..064])     //unverändert

CVTSD2SS:
XMMx[031..000] := SingleReal(XMMy[063..000])
XMMx[127..032] := XMMx[127..032])     //unverändert

CVTDPS2PD:
XMMx[063..000] := DoubleReal(XMMy[031..000])
XMMx[127..064] := DoubleReal(XMMy[063..032])

CVTPD2PS:
XMMx[031..000] := SingleReal(XMMy[063..000])
XMMx[063..032] := SingleReal(XMMy[127..064])
XMMx[127..064] := 0;
```

Operanden

Wie bei allen SSE2-Befehlen üblich ist bei diesen Konvertierungsroutinen das Ziel ein XMM-Register. Quelle kann jedoch auch hier wieder ein XMM-Register sein oder eine Speicherstelle:

- Konvertierung einer gepackten bzw. skalaren SingleReal aus einem XMM-Register in eine gepackte bzw. skalare DoubleReal in einem MMX-Register
  ```
  CVTPS2PD XMM, XMM
  CVTSS2SD XMM, XMM
  ```

- Konvertierung einer gepackten bzw. skalaren SingleReal aus einer Speicherstelle in eine gepackte bzw. skalare DoubleReal in einem MMX-Register bzw. eine LongInt in einem Allzweckregister
  ```
  CVTPS2PD XMM, Mem64
  CVTSS2SD XMM, Mem32
  ```
- Konvertierung einer gepackten oder skalaren DoubleReal aus einem XMM-Register in eine gepackte oder skalare SingleReal in einem XMM-Register
  ```
  CVTPD2PS XMM, XMM
  CVTSD2SS XMM, XMM
  ```
- Konvertierung einer gepackten oder skalaren DoubleReal aus einer Speicherstelle in eine gepackte oder skalare SingleReal in einem XMM-Register
  ```
  CVTPD2PS XMM, Mem128
  CVTSD2SS XMM, Mem64
  ```

CVTTPD2DQ
CVTTPD2PI
CVTTPS2DQ
CVTTPS2PI
CVTTSD2SI
CVTTSS2SI

Diese sechs Instruktionen können leicht verwechselt werden, da sie lediglich ein zusätzliches »T« im Namen tragen – und das auch noch an einer Stelle, an der es nicht besonders auffällt. In der Tat jedoch ist dieses T nicht ganz unwichtig, steht es doch für »*truncation*«. Und genau dieses Abschneiden unterscheidet sie von ihren Pendants. Genauer gesagt: Die »T-Modelle« der Befehle sind Spezialfälle, die sich genauso verhalten wie die Originale, wenn man in das Feld *rounding control* des MXCS-Register den Code für »truncation« eingegeben hätte. Sie machen damit ein häufiges Wechseln dieses Codes überflüssig.

Operanden

Bei den folgenden Befehlen kann die Quelle entweder eine Speicherstelle sein oder ein XMM-Register. Ziel ist je nach Zielformat ein XMM- oder MMX-Register:

- Konvertierung einer gepackten DoubleReal aus einem XMM-Register oder einer Speicherstelle in eine PackedInteger in einem XMM-Register
  ```
  CVTTPD2DQ XMM, XMM
  CVTTPD2DQ XMM, Mem128
  ```
- Konvertierung einer gepackten DoubleReal aus einem XMM-Register oder einer Speicherstelle in eine ShortPackedInteger in einem MMX-Register
  ```
  CVTTPD2PI MMX, XMM
  CVTTPD2PI MMX, Mem128
  ```

- Konvertierung einer gepackten SingleReal aus einem XMM-Register oder einer Speicherstelle in eine PackedInteger in einem XMM-Register
  ```
  CVTTPS2DQ XMM, XMM
  CVTTPS2DQ XMM, Mem128
  ```
- Konvertierung einer gepackten SingleReal aus einem XMM-Register oder einer Speicherstelle in eine ShortPackedInteger in einem MMX-Register
  ```
  CVTTPS2PI MMX, XMM
  CVTTPS2PI MMX, Mem64
  ```
- Konvertierung einer skalaren DoubleReal aus einem XMM-Register oder einer Speicherstelle in eine LongInt in einem Allzweckregister
  ```
  CVTTSD2SI Reg32, XMM
  CVTTSD2SI Reg32, Mem64
  ```
- Konvertierung einer skalaren SingleReal aus einem XMM-Register oder einer Speicherstelle in eine LongInt in einem Allzweckregister
  ```
  CVTTSS2SI Reg32, XMM
  CVTTSS2SI Reg32, Mem32
  ```

MMX-Befehlssatz

Während sich der XMM-Befehlssatz unter SSE2 mit Realzahlen beschäftigt, stellt der MMX-Befehlssatz unter SSE2 die Erweiterungen dar, die die MMX-Befehle mit Integers unter SSE2 erfahren haben.

Und an dieser Stelle können wir es uns leicht machen! Nehmen Sie jeden beliebigen MMX-Befehl, unabhängig, ob er bereits in den MMX-Erweiterungen implementiert wurde oder »erst« mit SSE, und erweitern Sie ihn um die Möglichkeit, anstelle von ShortPackedIntegers/QuadWords in den MMX-Registern auch die PackedIntegers/ DoubleQuadWords in den XMM-Registern zu verwenden. Und schon haben Sie die Erweiterungen des MMX-Befehlssatzes unter SSE2.



An dieser Stelle mache ich

1. es mir leicht,
2. es Ihnen schwerer und

verzichte auf die Darstellung der Operanden zu den einzelnen Befehlen. In Band 2, Die Assembler-Referenz, sind sie sowieso noch einmal mit ihren Opcodes im Einzelnen dargestellt!

Doch es gibt auch ein paar neue MMX-Befehle unter SSE2, die jedoch alle Abwandlungen von bereits bestehenden sind. Diese werden im Folgenden in der Reihenfolge des Alphabets beschrieben.

MOVDQA
MOVDQU
MOVDQ2Q
MOVQ2DQ

MOVDQA und MOVDQU sind die XMM-Variante des Befehls MOVQ. Während bei MMX mit MOVQ der gesamte Inhalt des Registers (64 Bit) beladen oder ausgelesen werden kann, erfolgt dies bei XMM (128 Bit) mit diesen beiden Befehlen. Der Unterschied zwischen beiden Befehlen liegt darin, dass MOVDQA Daten nur von bzw. an »ausgerichtete« Speicherstellen lesen bzw. ablegen kann, während MOVDQU diese Einschränkung nicht hat. Ausgerichtet heißt hierbei: Die Speicherstelle muss an einer Paragraphengrenze (16 Bytes) liegen. Tut sie das nicht, wird eine general protection exception #GP ausgelöst!

MOVDQ2Q und MOVQ2DQ sind eine MOV-Variante, die den Austausch zwischen MMX- und XMM-Register ermöglicht. Sie sind somit die XMM-Varianten des MOVD-Befehls unter MMX, der ja auch ein »halbes« MMX-Register mit einem Allzweckregister austauscht. MOVDQ2Q tauscht eben ein »halbes« XMM-Register mit einem MMX-Register aus:

```
MOVDQ2Q:
MMX[63..00] := XMM[063..000]

MOVQ2DQ:
XMM[063..000] := MMX[063..00]
XMM[127..063] := 0
```

Operanden

Die MOVD-Analoga können wie folgt verwendet werden (vgl. hierzu Seite 296):

- Kopieren eines Datums aus einem MMX-Register in das »untere« DoubleWord eines XMM-Registers
  `MOVQ2DQ XMM, MMX`
- Kopieren eines Datums aus dem »unteren« DoubleWord eines XMM-Registers in ein MMX-Register
  `MOVDQ2Q MMX, XMM`

MOVDQA / MOVDQU gestatten den generellen Datenaustausch zwischen einem XMM-Register und einer Speicherstelle oder einem anderen XMM-Register in beiden Richtungen (XXX steht für MOVDQA bzw. MOVDQU):

- Kopieren eines Datums aus einem XMM-Register in ein anderes XMM-Register
  `XXX XMM, XMM`
- Kopieren eines Datums aus einer Speicherstelle in ein XMM-Register
  `XXX XMM, Mem128`
- Kopieren eines Datums aus einem XMM-Register in eine Speicherstelle
  `XXX Mem128, XMM`

Alles in allem nichts Außergewöhnliches.

PADDQ PSUBQ PADDQ und PSUBQ sind die Erweiterungen der Befehle PADDB, PADDW und PADDD bzw. PSUBB, PSUBW und PSUBD auf die Welt der QuadWords und damit absolut nichts Ungewöhnliches. Auf eine weitere Besprechung kann daher an dieser Stelle verzichtet werden.

PMULUDQ PMULUDQ multipliziert zwei DoubleWords und legt das Produkt als QuadWord im Ziel ab. Der Befehl ist sowohl auf MMX- wie auch auf XMM-Register anwendbar. Werden MMX-Register verwendet, so kann nur jeweils das niedrigerwertige DoubleWord einer ShortPackedInteger als Operand herangezogen werden. Das Ergebnis wird dann als QuadWord (und nicht etwa wie bei den anderen Multiplikationsbefehlen anteilig) im MMX-Register abgelegt:

```
MMx[63..00] := MMx[31..00] * MMy[31..00]
```

Im Falle der Verwendung von XMM-Datenstrukturen wird jeweils das erste und dritte DoubleWord einer PackedInteger zur Multiplikation herangezogen, die beiden entstehenden QuadWords werden dann zusammen in das XMM-Register gelegt:

```
XMMx[063..000] := XMMx[031..000] * XMMy[031..000]
XMMx[127..000] := XMMx[095..064] * XMMy[095..064]
```

Operanden PMULUDQ kann wie folgt eingesetzt werden:

- Multiplikation zweier DoubleWords aus zwei MMX-Registern
  `PMULUDQ MMX, MMX`

- Multiplikation zweier DoubleWords aus einem MMX-Register und einer Speicherstelle
  `PMULUDQ MMX, Mem64`
- Multiplikation zweier DoubleWords aus zwei XMM-Registern
  `PMULUDQ XMM, XMM`
- Multiplikation zweier DoubleWords aus einem XMM-Register und einer Speicherstelle
  `PMULUDQ XMM, Mem128`

PSHUFLW
PSHUFHW
PSHUFD

PSHUFD ist die logische Weiterentwicklung des PSHUFW-Befehls für MMX-Daten (vgl. Seite 316) zur Anwendung auf XMM-Daten. Wie dort Words werden auch hier DoubleWords eines Quelloperanden nach Regeln gemischt, die dann in einem Zieloperanden neu zusammengesetzt werden. Die Regeln werden im dritten Parameter, einer Konstanten, übergeben, die als Feld mit vier Einträgen à 2 Bit interpretiert wird, deren Werte somit zwischen 0 und 3 liegen können.

Wie bekannt, sind diese Regeln auch hier in Indizes, allerdings nicht, wie bei PSHUFW in ein ShortPackedWord, sondern in ein PackedDoubleWord in der Quelle, die an die festgelegten Stellen im Ziel-PackedDoubleWord kopiert werden sollen:

```
XMMx[DoubleWord(0)] := XMMy[DoubleWord(Rule0)]
XMMx[DoubleWord(1)] := XMMy[DoubleWord(Rule1)]
XMMx[DoubleWord(2)] := XMMy[DoubleWord(Rule2)]
XMMx[DoubleWord(3)] := XMMy[DoubleWord(Rule3)]
```

PSHUFLW und PSHUFHW nun sind Befehle, die ebenfalls mit XMM-Registern arbeiten. Allerdings mischen sie, wie PSHUFW, Words, nicht DoubleWords. Dies kann entweder mit dem höherwertigen Word eines DoubleWords im PackedDoubleWord durch PSHUFHW erfolgen:

```
XMMx[Word(4)] := XMMy[HighWord(DoubleWord(Rule0))]
XMMx[Word(5)] := XMMy[HighWord(DoubleWord(Rule1))]
XMMx[Word(6)] := XMMy[HighWord(DoubleWord(Rule2))]
XMMx[Word(7)] := XMMy[HighWord(DoubleWord(Rule3))]
```

oder mit dem niedrigerwertigen mit PSHUFLW:

```
XMMx[Word(0)] := XMMy[LowWord(DoubleWord(Rule0))]
XMMx[Word(1)] := XMMy[LowWord(DoubleWord(Rule1))]
XMMx[Word(2)] := XMMy[LowWord(DoubleWord(Rule2))]
XMMx[Word(3)] := XMMy[LowWord(DoubleWord(Rule3))]
```

Operanden Als Ziel des Mischens kommt für alle Befehle nur ein XMM-Register in Frage, während die Quelle in einem XMM-Register oder an einer Speicherstelle stehen kann. Die Regeln stehend im dritten Parameter, einer Konstanten der Breite 8 Bit (XXX steht für PSHUFD, PSHUFHW oder PSHUFLW):

- Mischen der Komponenten einer PackedInteger aus einem XMM-Register in einer PackedInteger in einem XMM-Register
  `XXX XMM, XMM, Const8`
- Mischen der Komponenten einer PackedInteger aus einer Speicherstelle in einer PackedInteger aus einem XMM-Register
  `XXX XMM, Mem128, Const8`

PSLLDQ PSRLDQ PSLLDQ und PSRLDQ fassen den Inhalt eines XMM-Registers wie ihre MMX-Analoga PSLLQ und PSRLQ als einzelne Integer, also hier als QuadWord, und nicht als PackedInteger-Struktur auf. Sie sind somit die logische Fortführung der Reihe SHL – PSLLQ bzw. SHR – PSRLQ.

Operanden PSLLDQ und PSRLDQ können nur auf XMM-Register angewendet werden und erlauben auch nur eine Konstante zur Angabe der zu verschiebenden Positionen:

- Logische Verschiebung des Inhalts eines XMM-Registers um Const Positionen nach links:
  `PSLLDQ XMM, Const8`
- Logische Verschiebung des Inhalts eines XMM-Registers um Const Positionen nach rechts:
  `PSRLDQ XMM, Const8`

Optimierung der Datenströme Bleibt noch die Besprechung der »allgemeinen« Instruktionen, die mit der SSE2-Erweiterung neu hinzugekommen sind. Diese Befehle kümmern sich wiederum um das »streaming« in den *streaming SIMD extensions*, also den Teil, der für einen möglichst reibungslosen Austausch von Daten mit dem Speicher zuständig ist. Analog der Erweiterungen unter SSE gibt es dazu neue MOV-Variationen:

MOVNTDQ MOVNTPD MOVNTI MASKMOVDQU MOVNTDQ tauscht zwischen einem XMM-Register und dem Speicher ein DoubleQuadWord, also 128 Bit aus und benutzt dafür einen Mechanismus, der den Cache umgeht (»non-temporal hint«; zur Beschreibung des Begriffes »non-temporal« siehe Seite 339). Es ist damit das XMM-Analogon zu MOVNTQ, das ja bekanntlich den cache-umgehenden Austausch mit dem MMX-Register ermöglicht. Das Gleiche macht

MOVNTPD mit gepackten DoubleReals und hat damit sein SSE-Gegenstück in MOVNTPS. Und auch der cache-schonende Austausch zwischen einem Allzweckregister der CPU und dem Speicher wurde unter SSE2 realisiert: MOVNTI arbeitet mit LongInts und damit dem maximal in einem Allzweckregister darstellbaren Datum.

MASKMOVDQU ist das SSE2-Pendant zu MASKMOVQ und schreibt selektiv Bytes aus einem DoubleQuadWord in einem XMM-Register anhand einer Maske in den oder aus dem Speicher. Auch dieser Befehl umgeht dabei den Cache. MASKMOVDQU benötigt hierzu nicht eine ausgerichtete Speicherstelle: Wie das »U« im Befehlsnamen signalisiert, funktioniert das Ganze mit nicht ausgerichteten (»unaligned«) Daten.

```
IF Mask[007] = 1 THEN Dest[007..000] := Source[007..000]
IF Mask[015] = 1 THEN Dest[015..008] := Source[015..008]
IF Mask[023] = 1 THEN Dest[023..016] := Source[023..016]
IF Mask[031] = 1 THEN Dest[031..024] := Source[031..024]
IF Mask[039] = 1 THEN Dest[039..032] := Source[039..032]
IF Mask[047] = 1 THEN Dest[047..040] := Source[047..040]
IF Mask[055] = 1 THEN Dest[055..048] := Source[055..048]
IF Mask[063] = 1 THEN Dest[063..056] := Source[063..056]
IF Mask[071] = 1 THEN Dest[071..064] := Source[071..064]
IF Mask[079] = 1 THEN Dest[079..072] := Source[079..072]
IF Mask[087] = 1 THEN Dest[087..080] := Source[087..080]
IF Mask[095] = 1 THEN Dest[095..088] := Source[095..088]
IF Mask[103] = 1 THEN Dest[103..096] := Source[103..096]
IF Mask[111] = 1 THEN Dest[111..104] := Source[111..104]
IF Mask[119] = 1 THEN Dest[119..112] := Source[119..112]
IF Mask[127] = 1 THEN Dest[127..120] := Source[127..120]
```

Operanden

MASKMOVDQU hat wie MASKMOVQ drei Operanden: einen impliziten und zwei explizite. Der implizite Operand ist der Zieloperand (Dest); es handelt sich um eine Speicherstelle, die in der Adressierungsregister-Kombination DS:(E)DI angegeben ist. Diese Adresse muss auf eine Mem128 zeigen. Der zweite und somit erste explizit angegebene Operand ist die Quelle (Source); bei ihr handelt es sich immer um ein XMM-Register. Auch der dritte und damit als zweites explizit angegebene Operand muss ein XMM-Register sein, das die Maske (Mask) enthält. MASKMOVDQU wird somit wie folgt aufgerufen:

```
MASKMOVQ XMM, XMM
```

MOVNTDQ und MOVNTPD sind mit MOVNTQ und MOVNTPs verwandte Befehle, die ein XMM-Register auslesen und an eine Speicher-

stelle kopieren. Daher ist der jeweils erste oder Zieloperand eine Speicherstelle, der zweite oder Quelloperand XMM-Register:

```
MOVNTDQ Mem128, MMX
MOVNTPD Mem128, XMM
```

MOVNTI ist eine MOV-Variante, die auch für »einfache« Integers einen nicht-temporären Pfad zur Verfügung stellt und Daten von einem Allzweckregister in den Speicher transportiert:

```
MOVNTI Mem32, Reg32
```

CLFLUSH CLFLUSH dient dazu, den Cache – genauer gesagt: die *cache line*, die mit einer bestimmten Adresse verknüpft ist, in den Speicher zurückzuschreiben und dann zu invalidieren. Dieser Befehl ist daher im Zusammenhang mit den »strömenden« Daten und den »non-temporal hints« zu sehen und dient der Optimierung der Datenflüsse. Auf seine Einsatzgebiete wird im Rahmen dieses Buches nicht weiter eingegangen.

Operanden CLFLUSH hat einen Operanden, der auf eine Byte-Speicherstelle zeigen muss:

```
CLFLUSH Mem8
```

LFENCE MFENCE LFENCE und MFENCE sind Ergänzungen zu dem mit SSE eingeführten Befehl SFENCE. Während SFENCE einen »Zaun« bei Speichervorgängen errichtet, bewerkstelligt das LFENCE für Ladevorgänge. MFENCE kombiniert die beiden Befehle zu einem »globalen« Zaun. Für Details wird auch hier auf Sekundärliteratur verwiesen.

Operanden Wie SFENCE haben LFENCE und MFENCE keine Operanden:

```
LFENCE
MFENCE
```

PAUSE Auch auf Sekundärliteratur wird beim Pause-Befehl verwiesen. Nur so viel: Er dient dazu, zwei Besonderheiten des Pentium-4-Prozessors zu entschärfen: den hohen Stromverbrauch in und den Verlust an Performance beim Verlassen einer sog. »spin wait loop«.

Operanden PAUSE hat keine Operanden und wird daher wie folgt benutzt:

```
PAUSE
```

branch taken branch not taken Es gibt zwei Präfixe, die nur auf Maschinencode-Ebene zur Verfügung stehen und für die es – wie bei den prefixes 66h und 67h (*operand size override* und *address size override*) – keine Mnemonics gibt, die im Rahmen von Assemblern eingesetzt werden könnten. Diese Präfixe heißen

2Eh (*branch not taken*) und 2Fh (*branch taken*) und sind nur in Verbindung mit einem bedingten Sprungbefehl (Jcc – *jump on condition*) erlaubt. Sie geben dem Prozessor einen Hinweis, welcher Befehl als nächster abgearbeitet werden wird. Ein wenig genauer.

Wie Sie ja wissen, verfügt der Prozessor über eine mehr oder weniger ausgeprägte *prefetch queue*, in der die jeweils auf den zurzeit abgearbeiteten Befehl folgenden Instruktionen stehen. Dies erfolgt ja, um die Performance zu steigern: Denn während der Befehl abgearbeitet wird, kann durch Auslesen des Speichers parallel der jeweils nächste Befehl in die queue geholt werden – der Prozessor verliert damit keine wertvolle Ausführungszeit mit dem relativ zeitaufwändigen Speicherzugriff. Er bedient sich aus der (hoffentlich) immer korrekt gefüllten *prefetch queue*. Und genau hier liegt das Problem. Diese Queue wird immer dann richtig mit den jeweils auf die Situation korrekt angepassten Befehlen gefüllt sein, wenn keine Programmverzweigung ansteht. Denn nur in diesem Fall ist klar, dass der nächste Befehl im Speicher auch wirklich der nächste abzuarbeitende Befehl ist. Kommt es jedoch zu einer Programmverzweigung z.B. aufgrund einer erfüllten oder nicht erfüllten Bedingung, so kann es vorkommen, dass die in der *prefetch queue* stehenden Befehle die falschen sind – dann nämlich, wenn genau die Bedingung erfüllt ist, die der Lademechanismus der *prefetch queue* eben nicht angenommen hat. Konsequenz: Die gesamte *queue* muss geleert und dann wieder neu gefüllt werden, was einen erheblichen Zeitverlust zur Folge hat.

Wie kommt nun der Lademechanismus dazu, irgendeine Situation als wahrscheinlich »annehmen« zu können? Bei Schleifen dürfte die Sache noch einigermaßen klar sein: Schleifen werden häufiger zurückverzweigt als verlassen (ansonsten könnte man sie ja auch nicht als Schleife bezeichnen!), weshalb es eine gute Idee ist, anzunehmen, dass eine Rückverzweigung an den Schleifenbeginn erfolgt und die Befehle in der Schleife erhalten bleiben sollten. Und je nachdem, wie viele Befehle in der Schleife stehen und wie groß die *prefetch queue* ist, kann das bedeuten, dass über viele Schleifendurchgänge hinweg überhaupt nicht »nachgeladen« werden muss. Auf der anderen Seite jedoch stehen die Vergleichsbefehle. Hier ist absolut offen und sehr schlecht vorhersehbar, welchen Weg in der Verzweigung man gehen muss. Denn dies hängt nicht nur davon ab, um welche Bedingung es sich handelt, sondern auch welche Daten daran gemessen werden. In solchen Fällen ist fast nie eine korrekte Vorhersage machbar – es sei denn, man ist der Programmierer und weiß, welche Daten zum Einsatz kommen und wie

häufig diese Daten zu der einen oder anderen bedingten Verzweigung führen. Dies alles so »in Silizium zu gießen«, dass tatsächlich ein Performancegewinn herauskommt, ist nicht einfach – vor allem im Rahmen der Multimedia-Anwendungen.

Es wäre also nicht schlecht, wenn der Programmierer dem Prozessor praktisch einen Tipp geben könnte, was wohl als Nächstes am ehesten zu erwarten ist. Und genau zu diesem Zweck wurden mit SSE2 die beiden Präfixe 2Eh und 2Fh eingeführt. 2Eh sagt dem Prozessor, dass die Verzweigung mit hoher Wahrscheinlichkeit nicht erfolgt, er also gut daran tut, die *prefetch queue* weiter mit den nächsten Befehlen füllen zu lassen. 2Fh dagegen sagt ihm, dass es mit sehr hoher Wahrscheinlichkeit zu der Verzweigung kommen wird und er sich entsprechend darauf einstellen sollte. Sie können sich vorstellen, wie hilfreich diese beiden Präfixe im Hinblick auf Performance sein können, wenn es darum geht, Multimedia- oder andere aufwändige Audio-/Video-Anwendungen zu realisieren, bei denen eine hohe Anzahl gleicher Daten verarbeitet werden.

1.3.6 Exceptions unter SSE/SSE2

Non-numeric exceptions

Zunächst einmal können alle SSE-/SSE2-Befehle grundsätzlich auch CPU-Exceptions auslösen, wenn sie die entsprechenden Ursachen haben. Diese »non-numeric exceptions« betreffen

- Ausnahmesituationen, die beim Zugriff auf den Speicher auftreten können (#GP, #SS, #PF, #AC)
- System-Exceptions (#UD, #NM)

Details hierzu finden Sie im Kapitel »Exceptions und Interrupts« auf Seite 486.

Numeric exceptions

SSE und SSE2 haben den Prozessor-Befehlssatz um Instruktionen erweitert, die ein einfaches Manipulieren von Datenstrukturen aus Realzahlen ermöglichen. Kern der Instruktionen ist somit das Behandeln von Fließkommazahlen. Daher ergeben sich unter SSE und SSE2 die gleichen Probleme, die wir bereits bei der Besprechung der FPU mit ihren Befehlen erörtert haben: Es kann bei Fließkomma-Operationen zu Ausnahmesituationen kommen wie Überlauf, Denormalisierung oder ein falsches Zahlenformat. Diese Exceptions nennt man »numeric exceptions«.

Die Behandlung solcher numerischen Ausnahmesituationen bei der FPU ist einerseits im Rahmen von »Exception-Handlern« möglich: Programmteilen, die von der CPU aufgerufen werden, wenn sich eine bestimmte Ausnahmesituation eingestellt hat. Auf diese Weise ist es möglich, sinnvoll im Programm weiter zu machen, wenn z.B. versucht wurde, durch 0 zu dividieren. Andererseits kann die FPU durch Maskierung solcher Exception-Quellen auch selbst zur Klärung der Ausnahmesituation beitragen, indem sie z.B. Codewerte generiert, die die Ausnahmesituation zwar signalisieren, aber eine Fortführung des Programms ohne Unterbrechung ermöglichen. Sie kennen das aus der Beschreibung der FPU-Befehle.

MXCSR

Diese Möglichkeit der Nutzung von Exception-Handlern und der Maskierung von Exceptions wurde auch unter SSE und SSE2 für die Behandlung von Fließkommazahlen realisiert. Dies bedeutet aber, dass auch die dazu notwendigen hardwareseitigen Voraussetzungen geschaffen wurden. Das sind zum einen die unterschiedlichen Exceptions, die ausgelöst werden, wenn eine bestimmte Ausnahmesituation auftritt, sowie die entsprechenden Status- und Maskenbits im MXCS-Register (*multi-media control and status register*). Dieses ist in Abbildung 1.39 dargestellt. Wie man sieht, sind nur die Bits 0 bis 15 definiert, alle anderen gelten als reserviert und sollten nicht gesetzt werden, da dies zu einer #GP (*general protection exception*) führte.

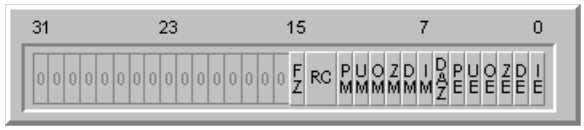


Abbildung 1.39: Das MXCS-Register

Betrachtet man dieses Register genauer, so könnte man glauben, es wäre aus dem *status register* und dem *control register* der FPU zusammengesetzt. Und so ähnlich ist es auch: Lässt man die FPU-Besonderheiten wie *busy*, *condition code*, TOS und *error summary* (*status register*) sowie *precision control* (*control register*) außer Betracht, finden sich auf engstem Raum und auf 16 Bit zusammengepfercht die gleichen Flags für Masken und Signale der gleichen Exceptions wie für die FPU. Selbst das Feld *round control* ist vorhanden. Hinzugekommen sind lediglich zwei Flags: *flush to zero* (FZ) und *denormals are zero* (DAZ).



Nach einem Prozessor-Reset ist der Defaultwert, der in das MXCS-Register eingetragen wird, $1F80. Das bedeutet: FZ ist gelöscht, rounding control erzwingt »Runden zur nächsten (geraden) Integer« (00_b; siehe Seite 193) und die Flags PM bis IM sind gesetzt, die entsprechenden Exceptions somit maskiert. DAZ ist ebenso gelöscht wie die SIMD-Exception-Statusflags PE bis IE.

LDMXCSR
STMXCSR

Das MXCSR kann durch LDMXCSR, *load multi-media control and status register*, oder FXRSTOR beschrieben und durch STMXCSR, *store multi-media control and status register*, oder FXSAVE ausgelesen werden. Damit ist das Setzen der Maskenbits möglich, ebenso wie das Prüfen, ob ein *exception flag* gesetzt ist. (Vgl. auch Seite 339).

Bei der Besprechung der Exceptions unter SSE und SSE2 können wir es uns leicht machen: Es gibt, verglichen mit den FPU-Exceptions, absolut nichts Neues. So kann eine *overflow exception* (OE) ausgelöst werden, wenn das Ergebnis das darstellbare Format überschreitet oder eine *underflow exception* (UE) bei einer Unterschreitung. Eine *precision exception* (PE) ist ebenso vorhanden wie eine *divide-by-zero exception* (ZE) und eine *denormal operand exception* (DE). Und es darf auch eine *invalid operation exception* (IE) nicht fehlen. Mit Hilfe der korrespondierenden Maskenbits können diese Interruptquellen maskiert werden. Ja selbst die Details sind absolut identisch: Auch die Statusbits im MXCSR sind »*sticky*« und müssen explizit gelöscht werden.

DAZ und FZ

Die wenigen Neuigkeiten sind schnell abgehandelt: *Flush to zero* (Bit 15) und *denormals are zero* (Bit 6) sind weder Masken noch Signale einer Exception. Mit ihnen kann gesteuert werden, welches Verhalten an den Tag gelegt werden soll, so bestimmte *exceptions* maskiert sind. Ist z.B. die *underflow exception* maskiert (UM = 1), sodass keine Exception ausgelöst wird, so wird, falls FZ gesetzt ist, der Inhalt des Registers gleich Null gesetzt und PE und UE gesetzt. Ist dagegen UM = 0, so wird eine #U ausgelöst und die Stellung von FZ ignoriert. FZ führt also zu einer Art »Sättigung« auf Null, falls der Wert zu klein würde, ohne den Programmablauf zu stören.

Analog arbeitet DAZ. Ist DAZ gesetzt, so wird im Falle des Auftretens einer Denormalen das Register auf Null gesetzt, wobei das ursprüngliche Vorzeichen erhalten bleibt (also de facto auf +0 oder -0). In diesem Falle wird weder eine Exception ausgelöst (DM = 0) noch DE gesetzt. DAZ erzwingt also das »Abrunden« auf Null, wenn sich ein Wert nur denormalisiert darstellen ließe.



DAZ und FZ erzwingen ein Verhalten, das nicht mit IEEE 754 konform ist. Diese Standardisierung verlangt nämlich eigentlich die Auslösung der exceptions bzw. das Signal, dass etwas nicht stimmt. Durch FZ und DAZ dagegen kann so getan werden, als ob nichts passiert sei. Der Hintergrund für diese Möglichkeit liegt einzig in der Verbesserung der Performance, die dadurch erreicht wird. Denn DAZ und FZ spielen ja nur dann eine Rolle, wenn der Wert einer Realzahl so klein ist, dass man sie mit Fug und Recht auf Null setzen kann, sodass dadurch kein gravierender Fehler entsteht. Auf der anderen Seite jedoch unterbrechen in diesem Fall keine exceptions den Programmablauf, was im Rahmen der Verarbeitung großer Mengen von »strömenden« Daten (SIMD!) von großem Vorteil ist.

OSXMMEXCEPT

In diesem Zusammenhang ist noch ein weiteres Bit im *control register* 4 der CPU von Bedeutung, das mit in die Exception-Problematik eingreift. Über dieses Bit 10, OSXMMEXCEPT, kann das Betriebssystem Anwendungsprogrammen mitteilen, ob es *exception handlers* für SIMD-Befehle systemseitig unterstützt oder nicht. Wenn dieses Flag gesetzt ist, heißt das, dass das Betriebssystem einen solchen Handler zur Verfügung stellt, der im Falle unmaskierter *exceptions* deren Behandlung übernimmt. In diesem Falle wird eine #XF (*SIMD floating point exception*) ausgelöst und die entsprechenden Statusbits im MXCSR gesetzt. Ist dieses Flag gelöscht, so wird beim Auftreten der nächsten SSE- oder SSE2-Instruktion eine #UD (*invalide opcode exception*) ausgelöst.

unmaskierte Exceptions

Auch wenn die SSE/SSE2-Befehle (zumindest die die Fließkommazahlen betreffenden) gleiche Exceptiongründe und -quellen haben wie die FPU-Befehle, läuft die Exceptionauslösung ein wenig anders ab als bei der FPU. Der Hintergrund ist, dass die FPU Exceptions selbst nicht auslösen kann und der CPU daher eine FPU-Exception durch ein gesetztes ES-Flag signalisieren muss. Die CPU prüft dieses ES-Flag nur bei WAIT/FWAIT und den meisten (nicht allen!) FPU-Befehlen. Daher kann es zu einer deutlichen zeitlichen Verschiebung zwischen Exceptionauslösung und -behandlung kommen, die nur dadurch verhindert werden kann, dass nach jedem FPU-Befehl, der eine Exception auslösen könnte, ein WAIT/FWAIT gesetzt wird – was nicht gerade im Sinne der Performance ist.

Die Fließkomma-SSE-/SSE2-Befehle dagegen werden durch die CPU ausgeführt – es gibt keine »MMXPU«! Daher stellt auch die CPU Exceptions, die bei diesen Befehlen auftreten können, unmittelbar fest und löst sofort einen #XF aus – so OSXMMEXCEPT das erlaubt. Dies ist auch der Grund, weshalb auf das ES-Flag verzichtet werden konnte. Al-

les weitere erfolgt wie von der FPU her gewohnt: Sind durch Löschen der entsprechenden Masken exceptions »unmaskiert«, so führen sie bei Auftreten der korrespondierenden Ausnahmesituation zur Auslösung der SIMD-Exception #XF. Die Ursache für die Auslösung wird dann durch Setzen der korrespondierenden Flags (IE, DE, ZE, UE, OE, PE) signalisiert und erfolgt in zwei Stufen:

- Zunächst wird auf Ausnahmesituationen reagiert, die vor der Operation auftreten können. Das sind das Vorliegen ungültiger Operanden, Division durch Null und ein denormalisierter Operand. Liegt eine solche Situation vor, so wird durch Setzen der Flags und ggf. der Auslösung der exception reagiert.
- Nach der Operation wird dann auf Ausnahmesituationen reagiert, die als Folge der Operation auftreten können: Überlauf, Unterlauf und Genauigkeitsprobleme. Liegt eine solche Situation vor, so wird analog reagiert.

Das bedeutet, dass ein Befehl durchaus Quelle für zwei exceptions sein kann: Wenn z.B. eine SSE- oder SSE2-Instruktion mit einem denormalisierten Operanden den kleinsten darstellbaren Wert endgültig unterschreitet. Dann wird vor der Operation eine #XF mit gesetztem DE ausgelöst und nach der Operation eine #XF mit gesetztem UE.

Der Ablauf sieht nun im Einzelnen wie folgt aus:

1. Prüfung auf »Vor-Ausführungsausnahmen«. Dies erfolgt für alle Daten der gepackten Struktur gesondert, wobei jedoch eine »Summe« gebildet wird: für jede Zahl in der gepackten Struktur gibt es einen Satz an internen Flags, die entsprechend gesetzt werden. Diese werden dann allerdings logisch ODER-verknüpft und das Ergebnis in MXCSR eingetragen. Die Summe stellt auch eine »Summe« im Hinblick auf die Exception-Quellen dar: Eine Denormale und der Versuch der Division durch Null bei einer anderen Realzahl der Datenstruktur führt nur zu einer Exception, wobei jedoch die entsprechenden Flags (in diesem Fall DE und ZE) gesetzt sind.
2. Liegt keine Ausnahmesituation vor, wird mit 6. weitergemacht.
3. Prüfung des Flags OSXMMEXCEPT in CR4. Ist dieses Bit gelöscht, stellt das Betriebssystem keinen *exception handler* für SIMD zur Verfügung. In diesem Fall wird eine #UD (*invalid opcode exception*) ausgelöst und der entsprechende Handler aufgerufen.
4. Andernfalls wird nun der Handler für #XF aufgerufen und somit eine #XF ausgelöst.

5. Der Handler hat nun dafür zu sorgen, dass die Ursache für die Exception beseitigt wird. Der Prozessor beginnt nämlich mit Schritt 1 von vorne, sodass sich Endlosschleifen ergeben könnten.
6. Prüfung auf »Nach-Ausführungsausnahmen«. Dies erfolgt wiederum für alle Daten der gepackten Struktur gesondert, wobei jedoch auch hier eine »Summe« gebildet und nur einmal die Exception ausgelöst wird. In diesem Falle bleiben alle Operatoren unverändert.
7. Liegt keine Ausnahmesituation vor, wird die Operation korrekt beendet.
8. Andernfalls wird wiederum das OSXMMEXCEPT-Flag geprüft und analog 3. verfahren. Es wird also entweder eine #UD oder eine #XF mit Ansprung des entsprechenden Handlers ausgelöst. Und auch in diesem Fall hat der Handler dafür zu sorgen, dass die Ursache beseitigt wird, da zu 6. zurückgegangen wird.

maskierte Exceptions

Auch im Falle der Maskierung von Exceptions verfährt der Prozessor wie eben geschildert. Das bedeutet, dass auch die Ausnahmesituationen für jede Zahl in der Struktur ermittelt und dann mittels ODER-Verknüpfung die »Summe« gebildet wird. Allerdings ist im Falle der Maskierung der weitere Verlauf unterschiedlich: So wird/werden abhängig von der Ursache der Ausnahme der/die Operatoren »maskiert«: Je nach Ursache und ggf. Stellung von FZ und DAZ werden Nullen, gerundete Werte, vorzeichenbehaftete Unendlichkeiten, Denormale, »Undefinierte« oder qNaNs erzeugt (vgl. hierzu »SIMD-Realzahl-Exceptions« auf Seite 542). Bis auf den Fall, dass ein Unterlauf ohne gleichzeitige Ungenauigkeit Ursache für die Exception ist, werden zusätzlich noch die Flags in MXCSR gesetzt.

1.3.7 Sind die SIMD verfügbar?

Erhebt sich die Frage, wie man feststellen kann, ob und, wenn ja, welche Erweiterungen der Prozessor unterstützt. Dies zu klären ist ein mehrstufiger Prozess. Er beruht hauptsächlich auf der Existenz des Befehls CPUID, der prozessorinterne Flagstellungen auslesen und damit über die Features des Prozessors Auskunft geben kann.

In der ersten Stufe ist daher zunächst festzustellen, ob der CPUID-Befehl überhaupt verfügbar ist. Er wurde ursprünglich mit dem Pentium eingeführt und dann z.T. nachträglich in verschiedenen Prozessoren »nachgerüstet«. Es ist also heutzutage einigermaßen sicher, dass der Rechner, auf dem ein Programm laufen soll, mit diesem Befehl gesegnet

ist. Doch es gibt auch heute noch alte Rechner auf Basis von 80386 oder 80486, die z.B. als Drucker-Server fungieren. Zwar laufen die heutigen hochgezüchteten Betriebssysteme wie Windows 2000 oder Windows ME auf diesen Rechnern nicht mehr; aber aus Kompatibilitätsgründen kann es doch Sinn machen, zu berücksichtigen, dass der vorliegende Prozessor CPUID eventuell nicht unterstützt.

Intel empfiehlt schlicht und ergreifend, CPUID einfach aufzurufen. Falls der Befehl nicht unterstützt würde, würde eine #UD (invalid opcode exception) ausgelöst. Warum dieser Rat gegeben wird, ist mir ein wenig rätselhaft, denn es gibt einen anderen Weg, der auch ohne exception und damit die Notwendigkeit zur Behandlung der Ausnahmesituation funktioniert.

- CPUID-Test; im EFlags-Register ist Bit 21, das *ID flag*, für CPUID verantwortlich. Ist dieses Bit umschaltbar, so ist CPUID verfügbar, ansonsten nicht!
- MMX-Test; das Ausführen des CPUID-Befehls mit dem Argument $00000001 in EAX liefert in EDX als Ergebnis *feature flags* (*ff*). Die *ff* sind ein Bitfeld, bei dem Bit 23 (*MMX flag*) anzeigt, dass MMX unterstützt wird. Dies alleine reicht eigentlich schon, um die Verfügbarkeit der MMX-Erweiterungen zu signalisieren. Trotzdem sollte noch geprüft werden, ob das FPU-Emulationsbit EM (Bit 2) im Kontroll-Register 0 gesetzt oder gelöscht ist. Ist es nämlich gesetzt, so ist die FPU-Emulation aktiviert und die Ausführung eines MMX-Befehls führt zum Auslösen einer #UD (*invalid opcode exception*).
- SSE-Test; in einem »Aufwasch« kann beim Testen auf MMX auch festgestellt werden, ob die SSE-Erweiterung implementiert ist. Denn Bit 25 der *feature flags* (*SSE flag*) signalisiert die Verfügbarkeit der SSE-Erweiterung, so wie ...
- SSE2-Test; ... Bit 26 der *feature flags* (*SSE2 flag*) es für die SSE2-Erweiterungen tut. Auch in diesem Fall könnte durch das einfache Prüfen dieser Bits der Test beendet sein. Es ist jedoch in beiden Fällen sinnvoll, ein wenig weiter zu testen. Denn es ist ja durchaus wichtig, zu prüfen, ob beispielsweise das aktuelle Betriebssystem bei einem *task switch* die FPU-/MMX-/XMM-Umgebung sichert oder das dem Anwendungs-Programmierer überlässt. Und in letzterem Fall ist nicht uninteressant zu wissen, ob die Befehle, die zum Sichern oder Restaurieren der Umgebungen erforderlich sind, überhaupt unterstützt werden.

- FXSR-Test; Bit 24 der *feature flags* (*FXSR flag*) signalisiert die Verfügbarkeit des FXSAVE-FXRSTOR-Paares, mit dem die Umgebungen gesichert oder geladen werden können.
- OS-Unterstützung FXSR; dieser Test dient der Klärung der Frage, ob das Betriebssystem das FXSAVE-FXRSTOR-Paar unterstützt. Hierzu wird das *control register #4* des Prozessors ausgelesen. Ist Bit 9, das OSFXSR flag, gesetzt, unterstützt das Betriebssystem die Befehle, ansonsten nicht. Auch hier kann gleichzeitig geklärt werden ...
- OS-Unterstützung SIMD exceptions; ... ob das Betriebssystem *exceptions* der SIMD-Fließkomma-Instruktionen unterstützt. Das ist der Fall, wenn das *OSXMMEXCPT flag* Bit 10 des *control registers #4* gesetzt ist. Dabei gibt es nur einen kleinen Haken: CR4 ist nur mit privileg level 0 ansprechbar, ansonsten gibt es eine #GP (*general protection exception*). Und das dürfte, so man nicht am Betriebssystem selbst herumbastelt, die Regel sein ...

Und wenn wir schon einmal beim Testen sind: Die Möglichkeit, Denormale auf Null abzurunden, wurde erst in späteren Versionen des Pentium 4 Prozessors eingeführt. Dies wird ja durch das *DAZ flag* im MXCS-Register gesteuert. Falls also diese Möglichkeiten genutzt werden sollen, so muss geprüft werden, ob der aktuelle Prozessor überhaupt über dieses *feature* verfügt. Dazu wird eine Variable erzeugt, die 512 Bytes Umfang hat und mit Nullen vorbesetzt wird. Nun wird, so der Test von eben ergeben hat, dass der FXSAVE-Befehl verfügbar ist, diesem die Variable als Argument übergeben. Das Ergebnis ist die aktuelle Umgebung, kopiert in die Variable. Die Bytes 28 bis 31 dieser Variable enthalten die MXCSR-Maske. Hat sie den Wert $00000000, liegt der Default-Zustand vor. Dieser Default-Zustand wird eigentlich durch die realen Bitstellungen $0000FFBF im MXCSR definiert, was bedeutet, dass ein FRSTOR bei Lesen des Wertes $00000000 den Wert $0000FFBF in das Register schreibt. In diesem Fall aber ist Bit 6, das *DAZ flag*, reserviert und signalisiert so, dass das Flag und damit der *denormals are zero mode* nicht unterstützt wird. Nur dann, wenn die Maske einen von $00000000 verschiedenen Wert hat, ist DAZ realisiert. Ein gesetztes Bit 6 zeigt dann, dass der Modus aktiviert wurde, bei einem gelöschten Bit wurde er deaktiviert.

Auf der beiliegenden CD-ROM befindet sich ein Windows-Programm, das die Verfügbarkeit von SIMD prüft.



1.3.8 3DNow!, die Erste: das AMD-SSE

3DNow!

Auch AMD hat die MMX-Erweiterungen, die Intel eingeführt hatte, übernommen. Es ist daher nicht notwendig, weitere Einzelheiten zu AMDs MMX-Implementierung zu nennen: Sie sind praktisch identisch. Allerdings hat AMD die Notwendigkeit zur Erweiterung der Multimediamöglichkeiten auf Fließkommazahlen anders umgesetzt als Intel.

3DNow!-Register

Während Intel mit SSE auf neue Register, die XMM-Register, setzte, hat AMD die bestehende Infrastruktur genutzt, frei nach dem Motto: Wenn schon MMX-Register für die Multimediaerweiterungen missbraucht werden, warum dann nicht in der ihnen ureigensten Art – mit der Bearbeitung von Fließkommazahlen? Dies führte dazu, dass 3DNow!, wie AMD diese Erweiterungen nannte, in den MMX-Registern angesiedelt ist.

Abbildung 1.40 zeigt die Nutzung der FPU-/MMX-Register unter dem 3DNow!-Befehlssatz. Die einzelnen Register werden wie die MMX-Register mit MM0 bis MM7 angesprochen und fassen jeweils zwei Single-Reals in Form einer »gepackten« Datenstruktur namens ShortPacked-Singles. Unter 3DNow! sind nur die Bytes 0 bis 7 der Register in Funktion, die beiden verbliebenen Bytes der FPU-Hardware werden auf $FF gesetzt. Das *tag register* sowie das *status* und *control register* haben unter 3DNow! keine Funktion.

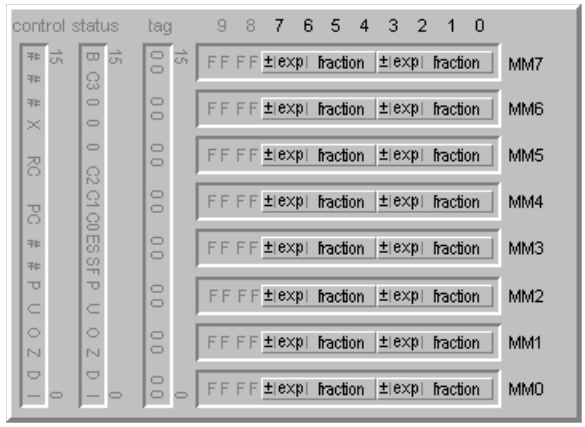


Abbildung 1.40: Die »3DNow!«-Register des Prozessors

Das führte jedoch zu erheblichen Einschränkungen und Inkompatibilitäten mit Intels SSE-Technologie. Denn die FPU- bzw. MMX-Register wurden nicht etwa aufgebohrt, sondern behielten ihre Breite von 80 bzw. 64 Bit. Da damit lediglich maximal zwei SingleReals mit insgesamt 64 Bit packbar waren, konnte und kann über diese Datenstrukturen hinaus nicht gearbeitet werden. Tabelle 5.27 auf Seite 850 stellt die unter 3DNow! verfügbaren Daten nochmals zusammen. Gemäß der Philosophie der Nomenklatur in diesem Buch wurden als neue Datenstrukturen die ShortPackedReals eingeführt.

3DNow!-Befehlssatz

Die Tatsache, dass nun die neuen gepackten Realzahlen in den ursprünglichen FPU-Registern verwaltet und bearbeitet werden, bedeutet jedoch nicht, dass auch die »normalen« FPU-Befehle auf die gepackten Strukturen angewendet werden könnten. Daher spendierte auch AMD seinen 3DNow!-Prozessoren einen neuen Befehlssatz für die Fließkomma-Erweiterungen unter Multimedia. Diese Befehle sind leider alles andere als kompatibel zu den Intel-Befehlen. Tabelle 5.30 auf Seite 855 versucht, vergleichbare Befehle beider Prozessorhersteller einander gegenüberzustellen. Grundlage hierfür ist der leider vergleichsweise geringe Umfang der AMD-Instruktionen.

Augenscheinlichster Unterschied der beiden Befehlssätze ist, dass Intel bei der Benennung der SSE-/SSE2-Instruktionen nicht solche führenden Buchstaben verwendet wie »F« für FPU-Instruktionen oder »P« für Befehle mit gepackten Strukturen unter MMX: Intels Befehle reihen sich, sinnvoll oder nicht, in die »normalen« CPU-Befehle ein. Anders AMD: Da auch die Fließkomma-Berechnungen unter 3DNow! in den MMX-Registern ablaufen und daher irgendwie zu MMX gehören, beginnen sie wie die MMX-Befehle mit »P« – gefolgt von einem »F«, weil sie ja Realzahlen betreffen.

Aber auch »unter der Haube« unterscheiden sich 3DNow! und SSE erheblich. Während Intel »seinen« Befehlssatz kräftig aufgebohrt und jeder neuen Instruktion neue Opcodes spendiert hat, hat AMD die Lösung des »double shifting« gewählt. Wenn man nämlich das Byte 0Fh als »Shift«-Taste zum Umschalten von Ein-Byte-Code-Befehlen zu Zwei-Byte-Code-Befehlen interpretiert, kann durch zweimaliges »Umschalten« mittels 0Fh, 0Fh auf eine »höhere« Ebene, eben die 3D-Now!-Ebene gewechselt werden. Einzelheiten hierzu entnehmen Sie bitte dem Kapitel »Befehls-Decodierung« ab Seite 832.

Auch AMD hat verschiedene Grundoperationen zur Verfügung gestellt:

- arithmetisches Manipulieren der Daten
- Datenvergleich
- Datenkonversion

PFADD PFSUB PFSUBR PFMUL

Addition, Subtraktion, Multiplikation – außer der Tatsache, dass es analog zu dem FPU-Befehl SUBR auch bei gepackten Zahlen des Type ShortPackedReals eine reziproke Subtraktion gibt, bei der nicht der Zieloperand vom Quelloperanden abgezogen wird, sondern umgekehrt, gibt es zu diesen Befehlen nicht viel zu sagen!

Operanden

Zieloperand und somit erster Operand der arithmetischen Berechnung ist immer ein MMX-Register, Quelloperand und zweiter Partner kann ein MMX-Register oder eine Speicherstelle sein. Damit kommen für die Befehle folgende Möglichkeiten in Frage (XXX steht für PFADD, PFSUB, PFSUBR und PFMUL):

- Arithmetische Verknüpfung einer Zahl in einem MMX-Register mit einer Zahl in einem MMX-Register
  `XXX MMX, MMX`
- Arithmetische Verknüpfung einer Zahl in einem MMX-Register mit einer Zahl aus einer Speicherstelle
  `XXX MMX, Mem64`

PFRCP PFRCPIT1 PFRCPIT2

Eine Division sucht man bei 3DNow! anders als bei SSE bzw. SSE2 vergebens. Das bedeutet, zwei ShortPackedReals können nicht durch einander dividiert werden! Es ist noch nicht einmal möglich, eine solche Division durch die Multiplikation mit dem entsprechenden Kehrwert zu erreichen. Was man jedoch tun kann, ist, eine ShortPackedReal durch eine Konstante c zu dividieren – wohlgemerkt beide SingleReals in der gepackten Struktur durch die gleiche (skalare!) Konstante. (In den folgenden Darstellungen sind die skalaren Werte mit Kleinbuchstaben, die gepackten Daten durch Großbuchstaben repräsentiert.)

q_1 := a_1/c; q_2 := a_2/c bzw. Q := A/c

Hierzu wird die Division in zwei Schritte zerlegt: Die Bildung des Kehrwertes der skalaren Konstante mittels PFRCP mit anschließender Multiplikation mit den Operanden:

q_1 := a_1 * 1/c; q_2 := a_2 * 1/c oder Q = A * 1/c

Das aber bedeutet, dass der Kehrwert der skalaren Konstante nach seiner Bildung im höher- und niedrigerwertigen Teil des Registers abgelegt werden muss, was PFRCP auch macht:

```
MMx[31..00] := Reciprocal(MMx[31..00])
MMx[63..32] := Reciprocal(MMx[31..00])
```

Auf diese Weise ist schnell die Division von Q = A / c erfolgt:

```
MOVD MM0, Divisor      ; Laden des skalaren Divisors in
                       ; MM0[31..0]
PFRCP MM0, MM0         ; Bildung des Kehrwertes in
                       ; MM0[31..00] und MM0[63..32]
MOVQ MM1, Dividend     ; Laden der SingleReals in
                       ; MM1[31..00] und MM1[63..32]
PFMUL MM0, MM1         ; Multiplikation
```

Das Ganze hat noch einen kleinen Schönheitsfehler! PFRCP bildet den Kehrwert, indem in einer ROM-basierten Tabelle nachgeschaut wird. Das Ergebnis ist damit recht ungenau: Die 24 signifikanten binären Stellen einer SingleReal führen zu einem Kehrwert mit max. 14 binären Stellen Genauigkeit. Dies ist wahrlich nicht viel, vor allem, wenn man es in uns gewohnterer dezimaler Genauigkeit ausdrückt: Reduktion von 8 dezimalen Stellen der SingleReal auf 5. Das mag zwar für viele Anwendungen ausreichend sein und damit diesen Befehl rechtfertigen. Doch es gibt auch sehr viele Fälle, in denen die an Genauigkeit sowieso schon nicht sonderlich protzenden SingleReals nicht durch Kehrwertbildung noch ungenauer werden dürfen. Kurz: Es muss ein Mechanismus her, der die Genauigkeit wieder erhöht. Dies ist möglich durch zwei weitere Befehle. Diese beiden Operationen stellen zwei Iterationsstufen der Kehrwertbildung nach dem Verfahren von Newton-Raphson dar und heißen PFRCPIT1 und PFRCPIT2. Sie erwarten folgende Operatoren in der angegebenen Reihenfolge:

1. `PFRCPIT1 [Input von PFRCP)], [Output von PFRCP]` oder
   `PFRCPIT1 [Output von PFRCP], [Input von PFRCP)]`

2. `PFRCPIT2 [Output von PFRCPIT1], [Output von PFRCP]`

Um eine auf die 24 binären Stellen einer SingleReal genaue Kehrwertbildung zu erhalten, ist also folgende Sequenz erforderlich:

X_0 := PFRCP(c)
X_1 := PFRCPIT1(c, X_0)
C^{-1} := PFRCPIT2(X_1, X_0)

Die Codesequenz für eine »exakte« Division könnte demnach wie folgt aussehen:

```
MOVD MM0, Divisor      ; Laden des skalaren Divisors in
                       ; MM0[31..0]
PFRCP MM1, MM0         ; Bildung des Kehrwertes in
                       ; MM1[31..00] und MM1[63..32]
PUNPCKLDQ MM0, MM0     ; Kopieren des Divisors in
                       ; MM0[63..00] für PFRCPIT1
PFRCPIT1 MM0, MM1      ; erste Iterationsstufe
PFRCPIT2 MM0, MM1      ; zweite Iterationsstufe
MOVQ MM1, Dividend     ; Laden der SingleReals in
                       ; M1[31..00] und MM1[63..32]
PFMUL MM0, MM1         ; Multiplikation
```

Operanden Zieloperand für das Ergebnis von PFRCP ist immer ein MMX-Register, Quelloperand und somit Argument für die Kehrwertbildung kann ein MMX-Register oder eine Speicherstelle sein. PFRCPIT1 und PFRCPIT2 benötigen zwei Quelloperanden. Diese werden wie üblich im ersten und zweiten Operanden des jeweiligen Befehls übergeben. Somit ist bei diesen Befehlen der erste Quelloperand auch gleichzeitig Zieloperand. Die Befehlssequenz sieht somit für die drei Befehle identisch aus (XXX steht für PFRCPIT1 und PFRCPIT2):

- Kehrwertbildung einer Zahl in einem MMX-Register bzw. aus einer Speicherstelle:
  ```
  PFRCP MMX, MMX
  PFRCP MMX, Mem64
  ```
- Iterationen zur Kehrwertbildung mit einem Startwert aus einem MMX-Register und einem zweiten Startwert aus einem MMX-Register oder einer Speicherstelle:
  ```
  XXX MMX, MMX
  XXX MMX, Mem64
  ```

PFRSQRT PFRSQIT1 Die gleiche Problematik liegt bei der Bildung der reziproken Quadratwurzel vor! Auch in diesem Fall wird von einer skalaren SingleReal ausgegangen, für die mittels PFRSQRT ein »ungenauer« Wert generiert und im Zielregister in die beiden SingleReals der ShortPackedReals abgelegt wird. Auch in diesem Fall muss eine weitere Iterationsstufe mit PFRSQIT1 und sogar eine zweite mit PFRCPIT2 herangezogen werden, will man die Genauigkeit auf die vollen 24 Stellen erhöhen. Ein Beispiel hierzu folgt etwas weiter unten.

Quadratwurzel?

Wenn man sich das *instruction set* von AMDs Fließkomma-MMX-Erweiterungen ansieht, so fällt einem auf, dass die Bildung einer Quadratwurzel fehlt. Warum? Ist AMD der Meinung, dass dies nicht notwendig ist? Nein! Der Hintergrund liegt in der gewöhnungsbedürftigen Art und Weise, wie AMD Reziprokwerte und reziproke Quadratwurzeln berechnet. Denn mit dem Algorithmus zur Berechnung der reziproken Quadratwurzel ist auch die Berechnung der Quadratwurzel möglich, und zwar über den einfachen mathematischen Zusammenhang:

$$a = \sqrt{x} = x^{1/2} = x^{1-1/2} = x^1 \cdot x^{-1/2} = x \,/\, \sqrt{x} = x \cdot (1 \,/\, \sqrt{x})$$

Multipliziert man also die berechnete reziproke Quadratwurzel eines Wertes mit dem Wert selbst, so bekommt man die Quadratwurzel des Wertes.

Ich überlasse nun jedem selbst, zu entscheiden, ob es nicht sinnvoller wäre, die paar *microcode instructions* noch im Prozessor zu implementieren, die man braucht, um »in einem Rutsch« Reziprokwerte, Quadratwurzeln oder reziproke Quadratwurzeln zu berechnen und daher mit AMD ein wenig zu schmollen ...

Ich schmolle nicht und stelle Ihnen nun vor, wie man die »ungenauen« Quadratwurzeln und ihre reziproken Werte berechnet und wie das Gleiche mit höherer Genauigkeit zu realisieren ist. Auch in diesem Fall werden dazu zwei Iterations-Routinen eingesetzt: PFRSQIT1 und das schon bekannte PFRCPIT2. Auch hier die Reihenfolge der Operatoren:

1. `PFRSQIT1 [Input von PFRSQRT], [Output von PFRSQRT]` oder
   `PFRSQIT1 [Output von PFRSQRT], [Input vom PFRSQRT]`

2. `PFRCPIT2 [Output von PFRSQIT1], [Output von PFRSQRT]`

Generell sind die Befehle somit wie folgt einzusetzen:

```
sq^-1:= PFRSQRT(a)
sq    := PFMUL(sq^-1, a)
```

für den »ungenauen« Fall und

```
X_0   := PFRSQRT(a)
X_1   := PFMUL(X_0, X_0)
X_2   := PFRSQIT1(a, X_1)
sq^-1:= PFRCPIT2(X_2, X_0)
sq    := PFMUL(sq^-1, a)
```

für den »exakten«. In Codesequenzen ausgedrückt heißt das, wenn man z.B. Makros einsetzt:

```
RSQRT_approx Macro Argument
       MOV    MM1, Argument
       PFRSQRT MM0, MM1
ENDM

SQRT_approx Macro Argument
       RSQRT_approx Argument
       PFMUL   MM0, MM1
ENDM

RSQRT_exact Macro Argument
       MOVD   MM2, Argument
       PFRSQRT MM1, MM2
       MOVD   MM0, MM1
       PFMUL   MM0, MM0
       PFRSQIT1 MM0, MM2
       PFRCPIT2 MM0, MM1
ENDM

SQRT_exact Macro Argument
       RSQRT_exact Argument
       PFMUL   MM0, MM2
ENDM
```

Bitte beachten Sie, dass die Berechnung der Quadratwurzel und ihres reziproken Wertes nur mit skalaren Reals erfolgt.

Operanden Zieloperand für das Ergebnis von PFSQRT ist immer ein MMX-Register, Quelloperand und somit Argument für die Kehrwertbildung kann ein MMX-Register oder eine Speicherstelle sein. PFRSQIT1 benötigt zwei Quelloperanden. Diese werden wie üblich im ersten und zweiten Operanden des jeweiligen Befehls übergeben. Somit ist bei diesem Befehl der erste Quelloperand auch gleichzeitig Zieloperand. Die Befehlssequenz sieht somit für beide Befehle identisch aus:

- Quadratwurzelbildung einer Zahl in einem MMX-Register bzw. aus einer Speicherstelle

  ```
  PFRCP MMX, MMX
  PFRCP MMX, Mem64
  ```

- Iterationen zur Quadratwurzelbildung mit einem Startwert aus einem MMX-Register und einem zweiten Startwert aus einem MMX-Register oder einer Speicherstelle
  ```
  PFSQIT1 MMX, MMX
  PFSQIT1 MMX, Mem64
  ```

PFACC

PFACC ist eine »Akkumulations«-Instruktion. Darunter versteht AMD die Addition von zweimal zwei SingleReals:

```
MMx[31..00] := MMx[31..00] + MMx[63..32]
MMx[63..32] := MMy[31..00] + MMy[63..32]
```

Operanden

Zieloperand für das Ergebnis von PFACC und erster Quelloperand ist immer ein MMX-Register, zweiter Quelloperand kann ein MMX-Register oder eine Speicherstelle sein:

- Akkumulation zweier Operanden aus MMX-Registern
  ```
  PFACC MMX, MMX
  ```
- Akkumulation eines Operanden aus einem MMX-Register mit einem aus einer Speicherstelle
  ```
  PFACC MMX, Mem64
  ```

PFMAX
PFMIN

AMD realisiert auch die Bestimmung der Minima und Maxima aus den beiden übergebenen Operanden mittels PFMIN und PFMAX:

```
MMx[31..00] := EXTREME(MMx[31..00], MMy[31..00])
MMx[63..32] := EXTREME(MMx[63..00], MMy[63..00])
```

Operanden

Zieloperand für den Extremwert und erster Quelloperand ist immer ein MMX-Register, zweiter Quelloperand kann ein MMX-Register oder eine Speicherstelle sein (XXX steht für PFMAX oder PFMIN):

- Extremwertbildung zweier Operanden aus MMX-Registern
  ```
  XXX MMX, MMX
  ```
- Extremwertbildung mit einem Operanden aus einem MMX-Register und einem aus einer Speicherstelle
  ```
  XXX MMX, Mem64
  ```

PFCMPEQ
PFCMPGT
PFCMPGE

Die durch AMD realisierten Vergleichsbefehle ähneln, auch wenn es zunächst nicht so aussieht, Intels CMPPS. Während Intel seinem Befehl ein Prädikat mitgibt, das angibt, welcher Vergleichstyp verwendet werden soll, stellt AMD drei Befehle zur Verfügung, die auf Gleichheit, »größer als« und »größer als oder gleich« prüfen. Wie bei Intels Instruktionen auch werden keine Flags als Resultat gesetzt, sondern Codeworte in die entsprechenden Teile der Datenstrukturen eingetragen: Trifft

der Vergleich zu, werden alle Bits der entsprechenden SingleReal im Zielregister gesetzt, andernfalls gelöscht.

Operanden Zieloperand für den Ergebniscode und erster Vergleichsoperand ist immer ein MMX-Register, zweiter Vergleichsoperand kann ein MMX-Register oder eine Speicherstelle sein (XXX steht für PFCMPEQ, PFCMPGT oder PFCMPGE):

- Vergleich zweier Operanden aus MMX-Registern:
  `XXX MMX, MMX`
- Vergleich eines Operanden aus einem MMX-Register mit einem aus einer Speicherstelle:
  `XXX MMX, Mem64`

PF2ID
PI2FD

Fehlen noch zwei Befehle, die eine Datenkonversion ermöglichen. Dies sind die Befehle PF2ID (*packed floating value to Integer*) und PI2FD (*packed Integer to floating value*). Die beiden »D« im Befehlsnamen sollen signalisieren, dass die Datengröße in jedem Fall ein Doppelwort, also vier Bytes ist.

Wir haben auch bei AMDs Lösung die gleichen grundsätzlichen Probleme mit der Konvertierung, die wir auf Seite 336 bereits bei der Besprechung der Intel-Analoga diskutiert haben: Eine Integer mit 32 signifikanten Stellen kann nicht immer exakt auf eine Realzahl mit 24 signifikanten Stellen abgebildet werden. Das bedeutet, dass eventuell eingegriffen werden muss.

Bei Intel erfolgte dies über das Feld *rounding control* im MXCS-Register, das ja für die Realzahlen in den XMM-Registern zuständig ist. Hier haben wir zwar auch das Feld *rounding control* im *control register* der FPU; dennoch können wir damit nichts anfangen, da 3DNow! auf MMX aufsetzt und damit nichts mit den FPU-Registern und deren Mechanismen zu tun hat – auch wenn sich MMX und FPU die Hardware in Form der physikalischen Register teilen! Es gibt somit keinerlei Möglichkeit, auf den Korrekturmechanismus von außen einzugreifen.

AMD musste daher eine Lösung finden, die den besten Kompromiss aus den verschiedenen »Rundungsmöglichkeiten« für Fließkommazahlen darstellt. Und das war die »truncation«. Das bedeutet, dass bei einer Konvertierung alles in Richtung »0« gerundet (also de facto abgeschnitten) wird, was die Zahl signifikanter Stellen der Mantisse übersteigt. Und »nach oben« hin, also bei einem Überlauf über $\pm 2^{31}$-1, erfolgt das, was Intel »Sättigung« nennt: Alle Werte über 2^{31}-1 werden

auf $7FFFFFF (=$2^{31}$-1) gesetzt, alle Werte unter -2^{31}-1 auf $80000000 (= -$2^{31}$-1).

Operanden

Da unter 3DNow! die MMX-Register sowohl mit Fließkommazahlen als auch mit Integers arbeiten, sind Quelle und Ziel der Konvertierungsbefehle unabhängig von ihrer Richtung gleich. Wie unter 3DNow üblich kann der zweite Operand, der die zu konvertierenden Zahlen enthält und damit Quelle der Befehle ist, entweder ein MMX-Register oder eine Speicherstelle sein, während der erste Operand als Ziel ein MMX-Register sein muss (XXX steht für PF2ID oder PI2FD):

- Konvertierung einer Zahl aus einem MMX-Register und Ablage des Ergebnisses in ein MMX-Register
  `XXX MMX, MMX`
- Konvertierung einer Zahl aus einer Speicherstelle und Ablage des Ergebnisses in einem MMX-Register
  `XXX MMX, Mem64`

MMX-Erweiterungen

Unter 3DNow! hat auch eine Erweiterung der MMX-Befehle stattgefunden. So wurden ein neuer Befehl zur Bildung eines Durchschnittswertes eingeführt (PAVGUSB), ein »verbesserter« EMMS-Befehl (FEMMS), ein weiterer Multiplikationsbefehl für Integers (FMULHPW) sowie zwei Befehle zur Beschleunigung der Datenströme (PREFETCH, PREFETCHW).

PAVGUSB

Das »USB« im Namen steht für *unsigned byte*. Damit ist klar, dass die Durchschnittsbildung nur mit vorzeichenlosen Daten des Typs ShortPackedByte möglich ist. Der Mittelwert wird gebildet, indem jeweils die beiden korrespondierenden Bytes in der gepackten Datenstruktur addiert werden. Dann wird zusätzlich eine »1« addiert und das Ergebnis intern um eine Stelle nach rechts verschoben, was einer Integer-Division mit 2 ohne Restbildung entspricht:

```
MMx[07..00] := (MMx[07..00] + MMy[07..00] + 1) SHR 1;
MMx[15..08] := (MMx[15..08] + MMy[15..08] + 1) SHR 1;
MMx[23..16] := (MMx[23..16] + MMy[23..16] + 1) SHR 1;
MMx[31..24] := (MMx[31..24] + MMy[31..24] + 1) SHR 1;
MMx[39..32] := (MMx[39..32] + MMy[39..32] + 1) SHR 1;
MMx[47..40] := (MMx[47..40] + MMy[47..40] + 1) SHR 1;
MMx[55..48] := (MMx[55..48] + MMy[55..48] + 1) SHR 1;
MMx[63..56] := (MMx[63..56] + MMy[63..56] + 1) SHR 1;
```

Die zusätzliche Addition von 1 bewirkt, dass der Mittelwert immer auf die nächste Integer gerundet ist. Wir haben das bereits auf Seite 311 für die Intel-Instruktionen besprochen.

Operanden Zieloperand für den Mittelwert und erster Summand ist immer ein MMX-Register, zweiter Summand kann ein MMX-Register oder eine Speicherstelle sein:

- Mittelwertbildung zweier Operanden aus MMX-Registern
  `PAVGUSB MMX, MMX`
- Mittelwertbildung eines Operanden aus einem MMX-Register mit einem aus einer Speicherstelle
  `PAVGUSB MMX, Mem64`

PMULHRW Dieser Befehl ist eine Abart des in beiden (AMDs und Intels) *instruction sets* enthaltenen Befehls PMULHW, indem er wie dieser zwei Worte miteinander multipliziert und das höherwertige Wort des entstehenden Doppelwortes in den Zieloperanden schreibt. Der Unterschied der beiden Befehle liegt darin, dass PMULHW lediglich die »oberen« 16 Bit kopiert, was einer Integerdivision (DIV) mit dem Wert $10000 entspricht, während PMULHRW nach der Multiplikation $8000 zum »unteren« Wort des temporär entstandenen Doppelwortes addiert, bevor das höherwertige Wort in das Zielregister geschrieben wird. Dies ist gleichbedeutend mit einer Rundung im Rahmen der Integerdivision mit dem Wert $10000.

Operanden Ziel für das Ergebnis und Multiplikand ist immer ein MMX-Register, Multiplikator kann ein MMX-Register oder eine Speicherstelle sein:

- Multiplikation zweier Operanden aus MMX-Registern:
  `PMULHRW MMX, MMX`
- Multiplikation des Multiplikanden aus einem MMX-Register mit einem Multiplikator aus einer Speicherstelle:
  `PMULHRW MMX, Mem64`

FEMMS AMD hat diese EMMS-Abart ins Leben gerufen, um einen schnelleren Wechsel des Kontextes vor oder nach MMX-Befehlen zu ermöglichen. Dies erfolgt, indem bei FEMMS anders als bei EMMS, das bei AMDs Prozessoren natürlich auch vorhanden ist, die Inhalte der Register als undefiniert markiert werden. AMD begründet dies damit, dass die Inhalte der MMX-Register ja nach einer MMX-Routine sowieso nicht mehr benötigt werden und man sich den Overhead, der durch das »Legalisieren« der Inhalte via EMMS erfolgt, sparen kann. Dies gelte auch

beim Eintritt in eine MMX-Routine, da dann klar ist, dass die eventuell vorhandenen FPU-Daten ebenfalls ungültig sind.

Wenn's scheee macht ...

Operanden

FEMMS hat keinen Operanden und kann nur wie folgt aufgerufen werden:

```
FEMMS
```

PREFETCH
PREFETCHW

Ich verweise analog den Intel-Befehlen zum prefetchen auf Sekundärliteratur, falls jemand diese Befehle tatsächlich braucht. Eine genauere Erklärung mit Anleitung zum Einsatz würde den Rahmen dieses Buches sprengen.

Operanden

PREFETCHTx hat einen Operanden, der auf eine Byte-Speicherstelle zeigen muss. Daher können alle PREFETCH-Varianten nur wie folgt aufgerufen werden:

```
PREFETCHTx Mem8
```

1.3.9 3DNow!, die Zweite: das AMD-SSE2

3DNow!-X

Vorbemerkung: AMD macht meines Wissens keinen Unterschied in der Bezeichnung der Extensions, die mit dem K6 bzw. dem Athlon in den Befehlssatz eingeflossen sind. Um aber hier die Unterschiede besprechen zu können, habe ich mir erlaubt, die mit dem Athlon eingeführten Erweiterungen als 3DNow!-X zu bezeichnen. 3DNow! ist damit eine Teilmenge von 3DNow!-X.

Die Erweiterungen, die AMD mit 3DNow!-X seinen Prozessoren spendiert hat, laufen im Prinzip darauf hinaus, möglichst weitgehend die Unterschiede zur Intel-Lösung unter SSE2 zu nivellieren. Zwar setzt AMD auch heute noch auf die 64-Bit-MMX-Register auch bei Fließkommazahlen und insofern ist es nicht nur schwer, sondern absolut vergebliche Liebesmüh, hier weitgehende Konformität zu erreichen. Weshalb AMD es auch unterlässt und für Fließkommazahlen lediglich ein paar sinnvolle Ergänzungen vornimmt. Für die Integer-Manipulationen in den MMX-Registern trifft das aber vollständig zu. So hat AMD alle MMX-Erweiterungen, die Intel bis zu SSE eingeführt hat, auch realisiert. SSE2 wurde erst mit dem Pentium 4 und damit nach dem Erscheinen des Athlon eingeführt, weshalb es nicht verwunderlich ist, dass zum derzeitigen Zeitpunkt (Anfang 2001) AMD noch keine SSE2-Analoga implementiert hat. Im Einzelnen:

Fließkommazahlen

Die wenigen Erweiterungen des Fließkomma-Befehlssatzes sind zwei weitere Konvertierungsbefehle (PF2IW und PI2FW), zwei weitere Befehle zum Akkumulieren (PFNACC, PFPNACC) und ein Swap-Befehl (PSWAPD):

PF2IW
PI2FW

Sie sind absolut identisch mit den Befehlen PF2ID und PI2FD mit der Ausnahme, dass die Konvertierung nicht von SingleReal zu LongInt (4-Byte-Datum ↔ 4-Byte-Datum) und umgekehrt erfolgt, sondern zu Integer (4-Byte-Datum ↔ 2-Byte-Datum) und umgekehrt. Ansonsten bleibt alles beim Alten: *truncation* zur Bereinigung von Ungenauigkeiten während der Konvertierung und Sättigung auf den jeweils höchsten positiven oder negativen Wert bei Überschreitung des Wertebereiches der Integer ($\pm 2^{15}$-1).

Operanden

Da unter 3DNow! die MMX-Register sowohl mit Fließkommazahlen als auch mit Integers arbeiten, sind Quelle und Ziel der Konvertierungsbefehle unabhängig von ihrer Richtung gleich. Wie unter 3DNow üblich kann der zweite Operand, der die zu konvertierenden Zahlen enthält und damit Quelle der Befehle ist, entweder ein MMX-Register oder eine Speicherstelle sein, während der erste Operand als Ziel ein MMX-Register sein muss (XXX steht für PF2IW oder PI2FW):

- Konvertierung einer Zahl aus einem MMX-Register und Ablage des Ergebnisses in ein MMX-Register
  ```
  XXX MMX, MMX
  ```
- Konvertierung einer Zahl aus einer Speicherstelle und Ablage des Ergebnisses in einem MMX-Register
  ```
  XXX MMX, Mem64
  ```

PFNACC
PFPNACC

Diese beiden Befehle sind Abarten der PFACC-Instruktion. PFNACC führt eine »negative« Akkumulation durch, was nur bedeutet, dass die einzelnen SingleReals nicht addiert, sondern subtrahiert werden. PFPNACC ist der Gemischtwarenhändler der drei Befehle, indem er mit einem Teil der SingleReals eine positive, mit den anderen Teil eine negative Akkumulation durchführt:

```
PFNACC
MMx[31..00] := MMx[31..00] - MMx[63..32]
MMx[63..32] := MMy[31..00] - MMy[63..32]

PFPNACC
MMx[31..00] := MMx[31..00] - MMx[63..32]
MMx[63..32] := MMy[31..00] + MMy[63..32]
```

Bei PFPNACC wird die niedrigerwertige SingleReal aus der Differenz der beiden Ziel-SingleReals gebildet und die höherwertige SingleReal aus der Summe der beiden Quell-SingleReals.

Operanden

Zieloperand für das Ergebnis von PFNACC und PFNACC und erster Quelloperand ist immer ein MMX-Register, zweiter Quelloperand kann ein MMX-Register oder eine Speicherstelle sein (XXX steht für PFNAC oder PFNACC):

- Akkumulation zweier Operanden aus MMX-Registern
  `XXX MMX, MMX`
- Akkumulation eines Operanden aus einem MMX-Register mit einem aus einer Speicherstelle
  `XXX MMX, Mem64`

PSWAPD

Der Swap-Befehl macht das, was man von ihm erwartet. Er tauscht die Plätze der beiden SingleReals des Quelloperanden aus und legt sie im Zieloperanden ab:

```
MMx[31..00] := MMy[63..32]
MMx[63..32] := MMy[31..00]
```

Operanden

Zieloperand für das Ergebnis von PFACC und erster Quelloperand ist immer ein MMX-Register, zweiter Quelloperand kann ein MMX-Register oder eine Speicherstelle sein:

- Tausch zweier Operanden aus MMX-Registern
  `PSWAPD MMX, MMX`
- Tausch eines Operanden aus einem MMX-Register mit einem aus einer Speicherstelle
  `PSWAPD MMX, Mem64`

MMX-Erweiterungen

Die restlichen Erweiterungen betreffen die Implementation von folgenden Befehlen:
MASKMOVQ, MOVNTQ, PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PREFTECHT0, PREFETCHT1, PREFETCHT2, PREFETCHNTA, PSADBW, PSHUFW, SFENCE.

Wenn Sie bitte vergleichen wollen: Es sind exakt die MMX-Befehle, die Intel im Rahmen der SSE-Erweiterung auch eingeführt hat. Falls Sie also dort nachlesen wollen ...

1.3.10 3DNow!, die Dritte: das Intel-SSE

3DNow!-Professional

Unter dem Namen *3DNow!-Professional* sind in Athlon-Prozessoren mit Palomino-Kern die SSE-Befehle der Intel-Prozessoren realisiert – inklusive der erforderlichen XMM-Register. Somit können auch AMD-Prozessorbesitzer das Kapitel »SIMD, die Zweite: SSE« ab Seite 307 lesen.

Offensichtlich haben die SSE2-Befehle noch nicht Einzug in einen mir bekannten AMD-Prozessor gefunden. Aber – was nicht ist, kann ja noch werden ...

1.3.11 Exceptions unter 3DNow!, 3DNow!-X und 3DNow! Professional

3DNow!, 3DNow!-X

AMDs eigene Philosophie zu SIMD mit Fließkommazahlen ist grundsätzlich eine andere als die von Intel. Während Intel SSE/SSE2 als »Erweiterung« der Möglichkeiten von Realzahlen sieht und daher konsequent Hardware in Form von eigenständigen Registern und einem MXCSR implementiert, ist für AMD 3DNow! lediglich »MMX mit anderen Daten«. Dies führt dazu, dass es nicht wie bei Intel-Prozessoren eigene Exceptions und deren Behandlung gibt. Vielmehr treten unter 3DNow! die gleichen Ursachen für Exceptions auf wie unter MMX und damit lediglich diejenigen, die auch bei Berechnungen in Allzweckregistern auftreten können, also CPU-Exceptions.

3DNow! Professional

Da die Befehle unter 3DNow Professional die gleichen sind wie die unter Intels SSE, gilt für Exceptions hier das Gleiche wie dort.

1.3.12 Ist 3DNow! verfügbar?

Es erhebt sich auch bei AMD-Prozessoren die Frage, wie festgestellt werden kann, ob die MMX- und 3DNow!-Erweiterungen überhaupt implementiert und damit nutzbar sind. Aufgrund der Unterschiede zwischen 3DNow! und SSE ist das Vorgehen ein wenig anders als das bei Intel-Prozessoren. Zunächst wird allerdings auch geprüft, ob der CPUID-Befehl verfügbar ist, da auch AMD über diesen Befehl die Features seiner Prozessoren bekannt gibt:

- CPUID-Test; im EFlags-Register ist Bit 21, das *ID flag*, für CPUID verantwortlich. Ist dieses Bit umschaltbar, so ist CPUID verfügbar, ansonsten nicht!
- MMX-Test; das Ausführen des CPUID-Befehls mit dem Argument $00000001 in EAX liefert in EDX als Ergebnis *feature flags* (*ff*). Die *ff*

sind ein Bitfeld, bei dem Bit 23 (*MMX flag*) anzeigt, dass MMX unterstützt wird. Dies alleine reicht eigentlich schon, um die Verfügbarkeit der MMX-Erweiterungen zu signalisieren. Trotzdem sollte noch geprüft werden, ob das FPU-Emulationsbit EM (Bit 2) im Kontroll-Register 0 gesetzt oder gelöscht ist. Ist es nämlich gesetzt, so ist die FPU-Emulation aktiviert und die Ausführung eines MMX-Befehls führt zum Auslösen einer #UD (*invalid opcode exception*). Diese Art der MMX-Detektion ist Intel-spezifisch und wurde daher aus Kompatibilitätsgründen auch in AMD-Prozessoren realisiert.

- MMX-Test; es gibt jedoch auch einen anderen, alternativen und typisch AMD-spezifischen Weg. Hierbei wird zunächst geprüft, ob der Prozessor »extended functions« unterstützt. Dies erfolgt, indem dem CPUID-Befehl als Argument in EAX der Wert $80000000 übergeben wird. Unterstützt der Prozessor die erweiterten Funktionen, was er durch die Rückgabe eines größeren Wertes als $80000000 in EAX signalisiert, so wird sofort die »extended function #1« des CPUID-Befehls aufgerufen, indem im EAX-Register $80000001 übergeben wird (Unterschied zu Funktion #1: gesetztes Bit 31). Ist nach dem CPUID-Befehl Bit 23 in EDX gesetzt, wird die Multimedia-Technologie unterstützt.
- 3DNow!-Test; an dieser Stelle kann dann auch gleich geprüft werden, ob auch 3DNow! unterstützt wird. Dies ist der Fall, wenn im Rückgabewert in EDX der CPUID-Funktion $80000001 das Bit 31 gesetzt ist.
- 3DNow!-X-Test; verfügt der Prozessor über die erweiterten 3DNow!-Befehle? Bit 30 der *extended feature flags*, die durch die *extended function #1* des CPUID-Befehls zurückgegeben werden, gibt hierüber Auskunft.
- MMX-Erweiterungen; Ob auch der MMX-Befehlssatz erweitert wurde, signalisiert nicht wie bei Intel die pure Einführung von SSE/SSE2. Vielmehr ist, wie AMD kommuniziert, 3DNow! ein »open standard«, an dem jeder ein wenig herumbasteln darf. Daher sollte man nicht davon ausgehen, dass die Einführung der 3DNow!-Erweiterungen auch gleichzeitig den MMX-Befehlssatz erweitert hat. Ob er nun erweitert ist, signalisiert Bit 22 der extended feature flags.
- FXSR-Test; Bit 24 der *feature flags* (*FXSR flag*) signalisiert die Verfügbarkeit des FXSAVE-FXRSTOR-Paares, mit dem die Umgebungen gesichert oder geladen werden können.
- OS-Unterstützung FXSR; dieser Test dient der Klärung der Frage, ob das Betriebssystem das FXSAVE-FXRSTOR-Paar unterstützt. Hierzu

wird das *control register* 4 des Prozessors ausgelesen. Ist Bit 9, das OSFXSR flag, gesetzt, unterstützt das Betriebssystem die Befehle, ansonsten nicht. Auch hier kann gleichzeitig geklärt werden ...

- OS-Unterstützung SIMD exceptions; ... ob das Betriebssystem *exceptions* der SIMD-Fließkomma-Instruktionen unterstützt. Das ist der Fall, wenn das *OSXMMEXCPT flag* Bit 10 des *control registers* 4 gesetzt ist. Aber auch hier: Nur unter privilege level 0 das CR4 ansprechen, ansonsten gibt's eine general protection exception (#GP)!



Auf der beiliegenden CD-ROM befindet sich ein Windows-Programm, das die Verfügbarkeit von SIMD prüft.

2 Hintergründe und Zusammenhänge

In diesem Kapitel werde ich Ihnen Informationen geben, die Sie beim Programmieren mit Assembler, aber auch mit Hochsprachen gebrauchen können. Sie brauchen dieses Kapitel nicht von vorne bis hinten durchzulesen! Vielmehr wurde im Kapitel 1 an verschiedenen Stellen auf einzelne Themen dieses Kapitels verwiesen (z.B. Datenformate, Exceptions), sodass Sie eventuell einige Abschnitte bereits durchgelesen haben. Auch kann es sein, dass Sie zu anderen Themen keine weiteren Informationen benötigen, da Sie z.B. schon die Unterschiede der verschiedenen Betriebsmodi der Prozessoren kennen oder wissen, was es mit der Speichersegmentierung auf sich hat.

Die einzelnen Themen dieses Kapitels bauen nicht aufeinander auf, sondern sind mehr oder weniger in sich abgeschlossene Teile, die ohne eine bestimmte Regel aneinander gefügt wurden. Benutzen Sie dieses Kapitel daher als »Nachschlagewerk« für Themen, die Sie nachlesen wollen.

2.1 Stack

»Stack« ist ein Begriff, mit dem man als Hochsprachenprogrammierer nur selten zu tun hat, obwohl, vor allem unter Pascal und Delphi (Stichwort: »lokale Parameter«), alle Programmierer ihn ausgiebig nutzen – unbewusst. Der Stack ist ein Bereich, auf den der Prozessor direkten Zugriff hat: Manche Befehle verändern ihn (PUSHx, POPx), manche Befehle benutzen ihn als Ablage (CALL, INTx).

Ein Stack ist zunächst einmal nichts anderes als eine Abfolge von Daten. Und wie »normale« Daten in einem »Datensegment« aufbewahrt werden (vgl. Seite 411), liegen diese in einem eigenen »Segment«, dem

Stack-Segment. Einzelheiten zum Stack*segment* finden Sie in »Stacksegmente« auf Seite 415.

2.1.1 Der Stack – ein Stapel Daten

Daten in Datensegmenten werden üblicherweise über ihre Adresse in ihrem Segment angesprochen, die in einem Offset zur Segmentbasis besteht. Dies erfolgt allerdings auf der Hardware-Ebene: Sowohl Hochsprachen wie auch Assembler gönnen dem Programmierer den »Luxus«, Adressen mit einem Namen zu versehen. Folglich kann der gestresste Programmierer auf Daten zugreifen, indem er dem Compiler/Assembler den »Namen« des Datums, das er ansprechen will, übergibt; dieser besitzt eine interne Liste, in der jedem Namen ein Offset im Datensegment zugeordnet wird, das dann entweder als effektive oder als logische Adresse beim Zusammenbauen (nichts anderes heißt »assemble«) der Befehlssequenzen benutzt wird.

Voraussetzung dafür, dass der Assembler/Compiler Sie bei der Benutzung von Daten auf diese Weise entlasten kann, ist jedoch, dass diese alle vor der Assemblierung/Kompilierung deklariert wurden! Denn die Zuordnungsliste Namen – Offsets kann er nur in diesem Fall erstellen. Das bedeutet, dass diese Art der Datenverwaltung nur mit statischen Daten möglich ist: Das Datum existiert während der gesamten Laufzeit des Programms.

Nicht immer aber hat man statische Daten. Sehr häufig kommt es vor, dass Daten nur für einen bestimmten Zeitraum benötigt werden. Denken Sie z.B. an Variable, die innerhalb einer Routine deklariert und benutzt werden (»lokale Daten« im Gegensatz zu den im Datensegment liegenden »globalen Daten«, die im gesamten Programm verfügbar sind). Ihre Existenz ist eng mit dem Aufenthalt des Prozessors »in der Routine« verbunden. Oder denken Sie an Daten, die im Rahmen der objektorientierten Programmierung in Objekten eingesetzt werden: Sie existieren erst, wenn das Objekt erzeugt wurde, und nur so lange, bis das Objekt zerstört wird. Auch kann es sein, dass die Datengröße im Vorhinein nicht definiert werden kann, sondern abhängig von bestimmten Randbedingungen ist, wie z.B. der Größe einer Textdatei. Wenn man dann flexibel reagieren können will (muss), wird die Größe in dem Augenblick definiert, in dem sie bekannt ist. Und das ist praktisch immer während der Laufzeit des Programms der Fall, nicht bei seiner Assemblierung/Kompilierung. In solchen Fällen spricht man von dynamischen Daten.

Zwar kann man auch solchen Daten Namen vergeben, wie jeder Programmierer weiß. Doch benötigt man dann ebenfalls eine statische Variable, die eine Adresse aufnimmt: die Adresse des jetzt dynamisch für dieses Datum reservierten Speicherbereichs. Und diese Adresse ist eben so lange Null, bis der Speicher »alloziert« wurde. Dann wird dieser statischen Variablen die Adresse des dynamisch erzeugten Datenbereiches zugeordnet. Und diese Adresse kann sich während der Laufzeit ändern, je nachdem, ob die dynamischen Daten zwischenzeitlich zerstört und wieder aufgebaut wurden und was dazwischen passierte. Das bedeutet, dass Sie auch zur Verwaltung von dynamischen Daten zumindest eine statische Variable brauchen, wollen Sie mit Variablennamen arbeiten. (Immerhin: Während bei statischen Daten Adresse und Datengröße ab dem Zeitpunkt der Programmerzeugung zementiert sind, können beide bei dynamischen Daten mittels »Allozierung/Deallozierung« noch zur Laufzeit verändert werden.)

Und noch ein Unterschied: Bei statischen Daten ist nicht nur die Adresse statisch, sondern auch die Datumsgröße. Durch Definition eines Datums vom Typ Soundso weiß der Assembler/Compiler, um was für ein Datum es sich handelt. Das bedeutet, dass er zum einen das Datensegment optimal und lückenlos aufbauen kann, zum anderen zu jedem beliebigen Zeitpunkt während der Assemblierung/Kompilierung eine Typprüfung durchführen kann (und somit Laut geben kann, wenn dem Register CX ein Wert aus `DVar` vom Typ DoubleWord zugeordnet werden soll). Danach ist das nicht mehr nötig.

Bei dynamischen Daten ist das anders. So kann der Assembler/Compiler z.B. in der Regel überhaupt nicht prüfen, um was für einen Datentyp es sich handelt, der da soeben dynamisch alloziert wurde (weshalb viele Hochsprachen Variablen, die Zeiger auf dynamisch erzeugte Daten aufnehmen, auch als »untypisierte« Variablen bezeichnen). Wenn im gewissen Umfang dennoch eine »Typprüfung« möglich ist, dann nur deshalb, weil man für eine bestimmte Datenstruktur einen Typ mit einer bestimmten, festen Größe definiert hat. Dann aber haben wir praktisch den gleichen Fall, wie wenn das Datum von vornherein statisch deklariert wurde. Dynamisch (im wahrsten Sinne des Wortes) erzeugte Daten lassen sich nicht typisieren!

Das bedeutet: Bei solchen Daten muss nicht nur die Adresse bekannt sein, sondern auch deren Größe. Bei der Allozierung von dynamischem Speicher erfolgt das, indem man der Routine, die für die Allozierung zuständig ist, die Größe des gewünschten Bereiches übergibt. Diese Größe muss dann auch der Deallozierungsroutine übergeben werden.

Generell gibt es zwei Möglichkeiten, solche dynamischen Daten zu organisieren: In »Stapeln« oder »Haufen«. Der Unterschied zwischen beiden ist erheblich:

- Heaps, was nichts anderes als »Haufen« heißt, sind im wahrsten Sinne des Wortes eine »Anhäufung« von Speicherbereichen, die entweder gerade Daten enthalten (weil der Bereich alloziert wurde) oder nicht (weil er freigegeben oder dealloziert wurde). Demgemäß sieht nach einer Weile heftigsten Treibens eines Programms mit dynamischen Daten ein einen Heap beherbergendes Datensegment wie ein Schweizer Käse aus: Es ist durchsetzt mit Löchern freigegebenen dynamischen Speichers unterschiedlichster Größe.
- Stacks, also »Stapel«, dagegen sind immer aufgeräumt! Sie besitzen keine Löcher (was für die Statik eines Stapels auch sehr schlecht wäre!) und sind immer nur so groß wie die Gesamtmenge der Daten, die auf dem Stapel liegen.

Das aber hat weitere Auswirkungen: An ein Datum auf dem Heap kommt man recht einfach heran: Man kennt seine Adresse und in der Regel auch seine Größe, sodass ein einfacher Speicherzugriff ausreicht. Bei Stapeln muss man ggf. anfangen zu suchen – und den Stapel abräumen, wenn man an ein Datum heran möchte. Denn von einem Stapel kennt man in der Regel nur eines: seine Spitze. Und nur an die Daten, die hier liegen, kommt man leicht heran.

Doch Heaps und Stacks unterscheiden sich in einer weiteren Hinsicht. In der Zeit, da Speicher noch knapp war, hat man sich gefragt, wie man wohl den »freien« Speicher, also den, der nicht durch Code- und Datensegmente belegt war, am sinnvollsten und effektivsten nutzen kann. Man wollte ihn *sowohl* durch Heaps *als auch* durch Stacks nutzen. Folglich blieb nichts anderes übrig, als den Heap an einem Ende des freien Speichers anzusiedeln und den Stack am anderen. Man definierte, dass Stacks am »oberen« Ende des Bereiches, also bei hohen Adressen, Heaps am »unteren« bei niedrigen angesiedelt werden.



Das hat aber weit reichende Konsequenzen. Denn während sich ein Heap damit zu unser aller Zufriedenheit »ganz normal« verhält, indem zusätzlicher Speicher am oberen Ende des Heaps angefordert wird, sobald er benötigt wird, funktioniert das beim Stack nicht! Hier muss zusätzlicher Speicher »unterhalb« der Adresse reserviert werden, die die Spitze des Bereiches angibt. Stacks stehen also »auf dem Kopf«.



Um sich das zu merken, gibt es ein schönes Bild: Tropfsteinhöhlen. Stacks und Heaps wachsen aufeinander zu, so wie es Stalagtiten und Stalagmiten tun. Die Stacks stellen hierbei die »hängenden« Stalagtiten dar, die Heaps die »stehenden« Stalagmiten.



Dieses Bild ist sogar so gut, dass es anschaulich darstellt, was passiert, wenn es keinen Platz mehr zwischen Stalagtiten und Stalagmiten mehr gibt, weil sie zusammengewachsen sind: Das Wasser läuft an ihnen ab, der GAU ist da! Auf den Computer übertragen nennt man das Stack-Heap-Kollision, die Daten gehen verloren.

Summieren wir an dieser Stelle, was wir über Stacks wissen:

- Sie stehen auf dem Kopf, was bedeutet, dass sie »oben« (bei hohen Adressen) beginnen und »nach unten« (zu niedrigen Adressen) wachsen.
- Sie besitzen eine Basis (Stackbasis) und eine Spitze (Stackspitze)
- Sie können mit anderen »Datenspeichern« wie Heaps oder Datenbereichen zusammenleben, ohne sich zu stören.

2.1.2 Stack frames – Verwaltung eines Stapels

Um nun ein klein wenig Ordnung in den Stapel zu bekommen, ihn also zu »strukturieren«, legt man so genannte Stack-Rahmen, *stack frames*, an. So ein Stack-Rahmen ist nichts Geheimnisvolles: Es sind zwei Adressen, die den Anfang und das Ende des Bereiches darstellen, der »gerahmt« werden soll. Somit hat man einen »privaten« Teil des Stacks, wenn man sich diese beiden Adressen merkt.

Die Hardware unterstützt stack frames, indem sie zwei Register zur Verfügung stellt, die die Anfangs- und Endadresse aufnehmen. Es sind das EBP- und ESP-Register. EBP, extended base pointer, zeigt hierbei auf den Anfang des Stack-Rahmens (und somit die »Basis«), ESP, extended stack, pointer auf das Ende des Rahmens. Mit Hilfe dieser beiden Register ist also der in diesem Stack-Rahmen eingeschlossene Bereich adressierbar. Dies muss jedoch über eine sog. »indirekte Adressierung« erfolgen (vgl. »Speicheradressierung« auf Seite 816).



Achtung! Da zwei Adressen bekannt sind, nämlich Anfang und Ende des Rahmens, kann innerhalb dieses Rahmens auch mit Hilfe zweier Methoden zugegriffen werden:

- Offset zum Anfang, der dann vom Inhalt des ESP abgezogen werden muss
- Offset zum Ende, der dann zum Inhalt von ESP addiert werden muss.

Beide Methoden sind möglich und verschiedene Hochsprachen verwenden diese beiden Methoden in unterschiedlicher Weise.



Natürlich können Sie auch Variablen deklarieren, die einzelne Adressen innerhalb eines Stacks referenzieren. Dies ist sogar eine im Rahmen der Hochsprachenprogrammierung gängige Methode, wenn auf Parameter zugegriffen werden soll, die einer Routine »über den Stack« übergeben wurden, oder auf »lokale« Variablen. Im Rahmen von Hochsprachen brauchen Sie sich um nichts zu kümmern, das macht der Compiler für Sie (weshalb Sie als unbedarfter Assemblerneuling bislang vermutlich wenig Kontakt zu Stacks hatten)! Doch auch die Hochspracheninterfaces von Assemblern stellen Ihnen den Komfort zur Verfügung. Wir werden bei der Besprechung der Assembler-Direktiven darauf zurückkommen.

Wenn dem so ist – warum dann die indirekte Adressierung über zwei Zeiger in ESP und/oder EBP? Ganz einfach! Zunächst einmal ist der Stack ein Notizzettel für den Prozessor, den ihm ein Programm zur Verfügung stellen muss. (Ich greife ein wenig vor: Ein Programm muss für jede der vier möglichen Privilegstufen einen Stack zur Verfügung stellen, den der Prozessor nutzen kann. Aber das werden wir bei der Besprechung der Schutzkonzepte noch erfahren).

Auf diesem Notizzettel vermerkt der Prozessor eine Menge Dinge: Zustand der Flags und Rücksprungadresse, sobald eine Routine aufgerufen wird, Fehlercodes bei Exceptions usw. Das bedeutet, der Stack ist Spielfeld des Prozessors und der Programmierer wird dort nur geduldet!

Das aber bedeutet, dass der Prozessor immer nur mit der Stapelspitze arbeiten kann. Andernfalls müsste er sich ja bestimmte Adressen merken können. Und dazu hat er, außer EBP und ESP, keine weiteren Register. Wo also könnte er solche Listen führen?

Dies ist letztendlich auch der Grund, warum Stack-Rahmen eine Bedeutung haben. So kann für jeden Kontext ein eigener Bereich auf dem Stack reserviert werden. Stellen Sie sich vor, Sie würden aus einem Hauptprogramm eine Routine aufrufen. Dadurch wechseln Sie »in eine andere Welt«, die zunächst einmal mit der »alten Welt« eines gemeinsam hat: den Ursprung, von dem aus Sie die Reise angetreten haben. Weniger prosaisch heißt das: die Rücksprungadresse.

Doch wie richtet man einen Stack-Rahmen ein? Betrachten Sie einmal Abbildung 2.1. Stellen Sie sich einfach einmal vor, es bestünde bereits ein Stack-Rahmen. Dies soll die Ausgangssituation auf der linken Seite der Abbildung sein. EBP zeigt auf die Basis des Rahmens, ESP auf die Spitze.

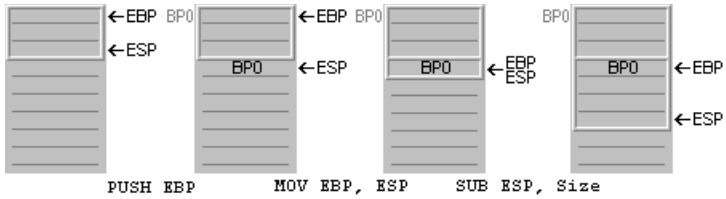


Abbildung 2.1: Einrichtung eines stack frames

Im ersten Schritt wird nun der Inhalt des EBP-Registers auf den Stack kopiert, was man »auf den Stack PUSHen« nennt. Wozu? Irgendwo müssen wir ja die Adresse der Basis des alten Stack-Rahmens ja speichern, wenn wir die Basis eines neuen Rahmens speichern wollen. Der stack ist damit um ein Datum gewachsen, weshalb ESP nun auch auf die nächstniedrige Adresse zeigt. Ermöglicht hat das der Befehl PUSH, dem als Argument das EBP-Register übergeben wurde.

Nun erklären wir die derzeitige Stackspitze zur neuen Basis, indem wir in das EBP-Register den Inhalt des ESP-Registers eintragen. Schließlich muss ja Rahmen auf Rahmen folgen, ohne »Löcher« zu hinterlassen. Dies erfolgt mit dem Befehl *MOV EBP, ESP*. Letzter Schritt: Wir ziehen im Register ESP die Menge von Bytes ab, die der neue Stack-Rahmen haben soll. Das macht der Befehl *SUB ESP, Size*.

Sie sehen, es ist kinderleicht, einen Stack-Rahmen zu erzeugen! Doch gehen wir einen Schritt weiter. Sie wollen wieder zurück in die »alte Welt«. Wie geht das? Nicht weniger einfach! Denn Sie wissen ja eines: Der Inhalt des EBP, also die Basis des aktuellen Stack-Rahmens, war ja die Spitze des »alten« Stack-Rahmens. Also ist es doch einfach, den In-

halt von EBP in ESP zu kopieren. Dies ist der erste Schritt in der Zerstörung des neuen Rahmens, wie Abbildung 2.2 zeigt.

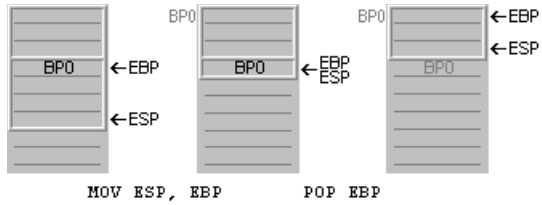


Abbildung 2.2: Entfernung eines stack frames

An der Stelle, auf die jetzt EBP und ESP zeigen, liegt ja die vorher gesicherte Adresse des alten stack frames. Also kopieren wir diese »vom Stack« in das EBP-Register. Das nennt man POPpen des Stack. Es erfolgt durch *POP EBP*.



Bitte passen Sie auf folgende Übereinkunft auf! Alles, was »unter« dem aktuellen Stack-Rahmen liegt (also was einmal »auf dem Stack« gelegen hat, bevor der Rahmen zerstört wurde), ist *nicht existent*! Das ist nicht trivial! Denn wie Sie anhand des Mechanismus der Stackrahmen-Zerstörung gesehen haben, werden keine Daten überschrieben, sondern nur Zeiger hin- und hergeschoben! Somit enthalten die benutzten Speicherstellen immer noch die Daten, mit denen Sie davor gearbeitet haben! Wirklich? Sind Sie sich sicher? Kann nicht der Prozessor in der Zwischenzeit, ohne das Sie das merkten, in genau diesem Bereich Daten überschrieben haben, z.B. mit Fehlercodes bei Exceptions oder Ähnlichem? Hat vielleicht eine Betriebssystemroutine, die Sie bewusst oder unbewusst aufgerufen haben, einen eigenen Stack-Rahmen dort errichtet, wo Sie Ihren hatten? Sie wissen es nicht! Also können Sie auch nicht sicher sein, dass die Daten, die Sie vor der Zerstörung verwendet haben, nachher auch noch unversehrt vorliegen.

Daher nochmals und sehr eindringlich: Daten unterhalb der Adresse, die in ESP als aktuelle Stackspitze steht, sind nicht existent und Zeiger auf solche Bereiche zeigen ins Nirwana! Nicht umsonst nennt man Variablen, die man auf dem Stack deklariert, lokal!

2.1.3 Stack Switching

Das soll als Hintergrundinformation zu Stacks ausreichen. Abschließend sei lediglich noch eine Information gegeben.

Ein Stack (nicht die Stack-Rahmen! Das sind nur Hilfen zur Strukturierung.) ist immer auf wundersame Weise mit dem Codesegment verbunden. Eigentlich ist das ja auch klar, handelt es sich doch beim Stack um den Notizblock des Prozessors, wenn er Code ausführt. Und so verwundert es nicht ernsthaft, dass der Prozessor auch den Stack wechselt, wenn sich das Codesegment ändert. Ich weiß, ich greife hier den nächsten Kapiteln vor. Denn bei dieser Information sollten Sie bereits wissen, wie die Speicherverwaltung funktioniert, was task switching ist und was es mit Privilegstufen auf sich hat. Behalten Sie daher diese Information ggf. ohne wirklich zu verstehen im Hinterkopf, bis Sie die entsprechenden Kapitel gelesen haben. Es geht nicht anders, denn hier handelt es sich um einen Teufelskreis: Ich muss etwas erklären, das die Kenntnis etwas anderen voraussetzt. Dies ist aber von der Kenntnis dessen abhängig, was hier erklärt werden soll.

Sobald das Codesegment geändert wird, ändert sich in der Regel auch das Stacksegment. Denn bei den modernen 32-Bit-Betriebssystemen ist jeder Code, den ein Programm benötigt, in einem Codesegment! Die Änderung wird somit dann und nur dann notwendig, wenn entweder in eine andere Privilegstufe gesprungen werden soll (z.B. in Betriebssystemroutinen), Interrupts behandelt werden oder ein task switch erfolgt. In all diesen Fällen wird aber ein jeweils »eigener« Stack verlangt.

Bitte denken Sie daher daran, dass

- bei jedem task switch
- bei jeder Nutzung eines call gates mit Codesegmenten, die eine höhere Privilegstufe haben
- bei jedem Interprivileg-CALL oder -JMP
- bei jedem Interrupt, der über einen task behandelt wird

neben dem Codesegment-Switch auch ein Stack-Switch erfolgt.

Bei »einfachen« Interrupts dagegen erfolgt kein Stack Switch! (Das wäre auch fatal, da ja der Stack für die Sicherung des Flagregisters und der Rücksprungadresse erforderlich ist!) Hier wird der jeweils aktuelle Stack »belastet«.

2.2 Speicherverwaltung

Das Kapitel Speicherverwaltung ist ein typisches Kapitel, in dem Ihnen Informationen gegeben werden, die nicht dazu dienen, Ihnen Wege aufzuzeigen, bestimmte Dinge zu tun oder zu umgehen. Vielmehr soll Ihnen in diesem Kapitel ein Eindruck vermittelt werden, wie die Dinge ablaufen und warum dies oder jenes so oder anders ist.

Dieses Kapitel wird Ihnen daher nicht detaillierte Informationen zu den einzelnen Themengebieten geben. Es vermittelt Ihnen lediglich einführende Hintergrundinformation, die Sie anhand von weiterführender Literatur vertiefen müssen, so Sie tatsächlich bestimmte Aspekte realisieren müssen oder wollen. Beispiel: Exception und Interrupt-Behandlung oder Taskwechsel.

2.2.1 Speicherorganisation

flat model

Stellen Sie sich vor, Sie hätten einen Speicher verfügbar, in dem Sie 4 GByte an Informationen abspeichern können, dessen »Linearer Adressraum« also, wie man sagt, 4 GByte umfasst. Und stellen Sie sich weiter vor, Sie hätten auch 32 Adressleitungen, um diesen Adressraum anzusprechen. Dann könnten Sie mit diesen 32 Leitungen 2^{32} Speicherstellen ansprechen, also insgesamt 4 GByte – Ihren gesamten Adressraum. Sie könnten somit durch Spielen auf Ihrer 32-Bit-Klaviatur jedes verfügbar Byte direkt und ohne Klimmzüge ansprechen. Programmiermodelle, in denen das möglich ist, nennt man »Flache Modelle« (*flat models*), weil Sie wie in einer Ebene bis zum Horizont alles sehen können, ohne durch Barrieren behindert zu werden.

segmented model

Doch es mag Gründe geben, nicht alles flach zu lassen und solche Barrieren, die einem den ungehinderten Blick zum Horizont verwehren, aufzubauen. So ist es sicherlich sinnvoll, z.B. alle Daten eines Programms zusammenzufassen und in einen Container zu tun, auf dem »Daten« steht. Analog kann man mit dem Programmcode verfahren und mit dem Notizblock des Prozessors, dem Stack. Dadurch hat sich zwar nichts in der Adressierbarkeit selbst geändert. Doch Sie haben den unstrukturierten Adressraum strukturiert, indem Sie ihn in »Segmente« aufgeteilt haben: Ein Codesegment, ein Datensegment und ein Stacksegment. Modelle, die diese Grundlage haben, nennt man »Segmentierte Modelle« (*segmented models*).

Welchen Vorteil hat nun ein segmentiertes Modell gegenüber einem flachen? Denn an der Adressierung hat sich ja trotz Einführung der Seg-

mente zunächst einmal nichts geändert: 32 physikalisch vorhandene Adressleitungen können bis zu 4 GByte adressieren. Und in der Tat: Da durch die Segmentierung der lineare Adressraum lediglich formal aufgeteilt wurde, ist auch innerhalb der Segmente jede Stelle linear (also direkt!) mit einer eindeutigen Adresse ansprechbar – egal, wie groß die Segmente sind. Unter einer Voraussetzung: Die Segmente dürfen nicht größer werden können als der gesamte lineare Adressraum. Was also bringt Segmentierung?

Speichersegmentierung hat Vorteile: Sie können nämlich einem Segment bestimmte Eigenschaften zuordnen, mit denen es sich von anderen unterscheidet. Die einfachsten sind, natürlich, seine Lage im linearen Adressraum, also an welcher Adresse es beginnt. Ferner können Sie eine Größe dieses Segmentes angeben. Allein schon durch diese beiden Angaben können Sie eine primitive Art eines Schutzkonzeptes implementieren: Wenn Sie z.B. verhindern wollen, dass die Informationen in Ihrem Codesegment von irgendjemandem verändert werden (DOS lässt grüßen!), so brauchen Sie nur dann, wenn dieser Jemand auf eine beliebige Adresse zugreifen will, prüfen, ob die Adresse innerhalb des zu schützenden Segmentes liegt. Ist das der Fall, so verbieten Sie einfach den Zugriff, ansonsten nicht. Voraussetzung hierzu ist allerdings, dass Sie Mechanismen und Institutionen entwickeln, die diese Prüfung auch tatsächlich durchführen und für den Schutz sorgen. Sie brauchen also ein »Betriebssystem«, dessen Aufgabe es ist, solche Schutzkonzepte zu realisieren und sie mehr oder weniger restriktiv durchzusetzen. Und Sie brauchen Hardware, die das Betriebssystem in seinem Bemühen unterstützt.

2.2.2 Segmente

Doch Sie können Segmenten noch weitere Eigenschaften vergeben. So können Sie z.B. Informationen vorsehen, welchen Typ Daten sie enthalten: Handelt es sich in einem konkreten Fall um ein Segment mit ausführbaren Befehlen (»Codesegment«)? Oder enthält es Daten (»Datensegment«)? Ist es vielleicht ein Segment, was Informationen für die Mechanismen (»Betriebssystem«) beinhaltet, die die Schutzkonzepte durchsetzen (»Systemsegmente«), oder ist es nur der Notizblock des Prozessors (»Stacksegment«). Soll das Segment nur lesbar sein oder darf es auch inhaltlich verändert werden? Und von wem? Je nachdem, wer dann auf welches Segment wie zugreifen will, können Sie dies erlauben oder nicht.

Segmente sind also Container! Sie enthalten bestimmte Informationen, die zusammengehören und damit eine Einheit bilden. Hierbei ist es vollkommen unerheblich, um was für Informationen es sich handelt. Wie gesehen, kann das z.B. der gesamte ausführbare Code eines Programms sein oder die Gesamtheit seiner Daten. Es können Tabellen oder Listen mit bestimmten Aufgaben sein. Oder es können Strukturen sein, in denen der augenblickliche Zustand oder die Umgebung eines Tasks verzeichnet werden. All diese Informationen werden in Segmenten gehalten, die daher alles andere als einheitlich sind.

segment descriptor

Das bedeutet, dass es zu jedem Segment Informationen geben muss, die ein Segment beschreiben. Diese Informationen werden in einer Struktur zusammengefasst, die man sinnigerweise und recht treffend *segment descriptor* nennt. In diesem »Segmentbeschreiber« sind die wesentlichen Informationen des Segmentes wie Basisadresse im linearen Adressraum, Größe, Art der beinhalteten Information und bestimmte Eigenschaften (= Attribute) verzeichnet. Jedes Segment, egal, was es beinhaltet, hat somit einen Deskriptor. Ohne seinen Deskriptor ist ein Segment nicht existent! Abbildung 2.3 zeigt einen solchen Deskriptor.

Ein Deskriptor besteht aus 64 Bit an Informationen, die in zwei aufeinander folgenden DoubleWords zusammengefasst sind, wobei üblicherweise das erste immer als »unteres«, das zweite als »oberes« DoubleWord dargestellt wird. Die Informationen liegen historisch bedingt zerstückelt vor (der 80286 hatte nur 24 Adressleitungen, weshalb die Basisadresse auch nur 24 Bits umfassen konnte), was uns aber nicht zu interessieren braucht: Der Prozessor fügt sie automatisch zusammen. Wie Sie sehen, sind die eben genannten Informationen hier verzeichnet.

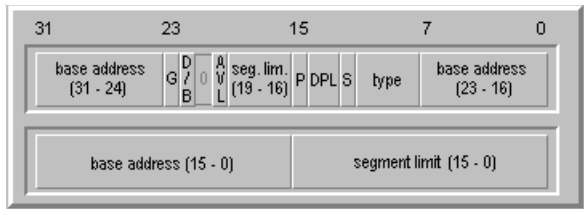


Abbildung 2.3: Speicherabbild eines Segment-Deskriptors

base address, segment limit

So gibt es eine 32 Bit breite Basisadresse (*base address*). Dies ist die lineare Adresse, an der das Segment beginnt. Segmente können somit theoretisch überall im Adressraum von 4 GByte beginnen, der mit 32 Adressleitungen ansprechbar ist. Praktisch allerdings sind Segmente »ausgerichtet«, was bedeutet, dass sie (z.B. aus Gründen, die wir beim

Paging kennen lernen werden,) an ganz bestimmten Adressen beginnen. Ferner gibt es eine als *segment limit* bezeichnete, 20 Bit breite Größe des Segmentes. Diese 20 Bit erlauben 2^{20} = 1.048.576 = 1 MByte große Segmente.



Die Basisadresse mit ihren 32 Bit ist eine »echte« lineare Adresse, die Grundlage für die Berechnung einer »virtuellen« Adresse ist (worum es sich dabei handelt, werden wir noch sehen!). Das Limit dagegen ist keine Adresse, wie man vielleicht im ersten Augenblick annehmen könnte, sondern ein numerischer Wert, der zu der Basisadresse hinzu addiert werden muss (= Offset), um die Adresse des Segment-Endes zu berechnen. Es ist damit die *Differenz* aus zwei Adressen, die die Größe des Segmentes angibt.

granularity flag

Nur 1 MByte große Segmente? Das ist ja *nur* der Adressraum, der in grauer Vorzeit zu Zeiten des *disk operating System* DOS einmal der gesamte Raum war, in dem sich alles abspielte und der sehr schnell zu klein wurde. Ist das daher unter Umständen nicht ein wenig zu klein, vor allem, wenn man an Code- und Datensegmente denkt? Richtig! Daher besitzt ein Deskriptor auch mit dem *granularity flag* G (Bit 23 des zweiten DoubleWords) ein Flag, das signalisiert, wie das Segmentlimit zu interpretieren ist. Ist es gelöscht, so ist die »Auflösung« (*granularity*) des Segments das Byte und es können »nur« 1.048.576 Einheiten à 1 Byte = 1 MByte große Segmente realisiert werden. Seine Mindestgröße ist dann eine Einheit = 1 Byte. Ist es dagegen gesetzt, so beträgt die Auflösung 4-KByte-Einheiten, was bedeutet, dass das Segment den gesamten linearen Adressraum ausfüllen kann: 1.048.576 Einheiten à 4 KByte = 4 GByte. Es muss dann aber auch mindestens eine Einheit = 4 KByte groß sein.



Was bedeutet das in praxi? Heißt das, dass bei gesetztem *granularity flag* nicht mehr einzelne Bytes, sondern nur noch 4-KByte-Blöcke angesprochen werden können? Nein! Die Segmentgröße ist ja »nur« Teil eines Schutzkonzeptes, nicht aber Teil der Adressierung einer Speicherstelle selbst. Mit ihr wird also lediglich die Größe des Segmentes festgelegt. Um auf eine Speicherstelle zugreifen zu können, müssen Sie zu der Basisadresse des Segmentes noch einen Zeiger addieren, der die relative Position des gewünschten Datums zum Beginn dieses Segmentes angibt. Dieser als *Offset* bezeichnete Zeiger ist selbst ein 32-Bit-Wert, sodass man mit ihm jedes beliebige Byte im linearen Adressraum ansprechen kann. Um aber tatsächlich die *Erlaubnis* zum Zugriff zu erhalten, wird nun gemäß unserer oben geäußerten Gedanken zunächst geprüft,

ob dieser Offset die Grenzen des Segmentes respektiert. *Und diese Prüfung kann mit unterschiedlicher Auflösung erfolgen!* Das bedeutet, dass bei einer *byte granularity* bei gelöschtem granularity flag bis zum einzelnen Byte hinunter festgestellt werden kann, ob die Grenzen ggf. verletzt werden, da die Größe des Segmentes in Byte-Einheiten angegeben ist. Bei *page granularity*, wie man die Auflösung von 4-KByte-Einheiten auch gerne nennt, kann nur noch bis auf die Page-Ebene festgestellt werden, ob ein Zugriff erlaubt ist, weil hier die Größe des Segmentes in 4-KByte-Einheiten gemessen wird.



Physikalisch erfolgt dies, indem bei gelöschtem granularity flag der Offset direkt mit dem Limit verglichen wird. Sind dann einzelne der Bits 20 bis 31 des Offsets gesetzt (womit er offensichtlich das 20-Bit-Limit überschreitet) oder ist seine durch seine Bits 0 bis 19 repräsentierte Größe wertmäßig größer als das Limit, so wird der Zugriff verwehrt und eine exception ausgelöst. Bei gesetztem granularity flag dagegen werden die Bits 0 bis 11 des Offsets nicht in die Überprüfung einbezogen, was bedeutet, dass alle Bytes in dieser 4-kByte-Einheit (2^{12} = 4.096 = 4 k) gleich behandelt werden. Die Überprüfung erfolgt nun durch Testen der »oberen« 20 Bits 31 bis 12 des Offsets gegen die 20 Bits des Limits, die hier als Bit 31 bis 12 eines virtuellen 32-Bit-Limits interpretiert werden.

S Flag Das S-Flag gibt an, ob das vorliegende Segment ein Systemsegment ist (S = 0) oder ein Code- bzw. Datensegment (S = 1). Diese Unterscheidung ist wichtig, da Code- und Datensegmente andere Aufgaben haben als Systemsegmente und daher einige Unterschiede in der Bedeutung ihrer Informationen aufweisen. Wir werden bei der Besprechung der einzelnen Segmenttypen noch darauf zurückkommen.

Type So ist z.B. die Bedeutung des mit *type* bezeichneten Bit-Feldes der Bits 8 bis 11 des zweiten DoubleWords vom Segmenttyp abhängig. In diesem Feld werden weitere Untertypisierungen des Segmentes vorgenommen. Auch hierauf werden wir weiter unten noch genauer eingehen.

DPL Ein weiteres Bit-Feld aus den beiden Bits 13 und 14 des zweiten DoubleWords wird mit DPL bezeichnet. Dieser *descriptor privileg level* ist Teil des Schutzkonzeptes und wird im Kapitel 2.4 auf Seite 470 näher erläutert.

P Flag Das *present flag* P wird beim sog. Paging-Mechanismus eingesetzt und zeigt an, ob das Segment derzeit verfügbar (»present«) ist oder nicht. Weitere Informationen hierzu finden Sie weiter unten bei der Bespre-

chung des Paging-Mechanismus. Ist es gelöscht, so sieht der Deskriptor wie in Abbildung 2.4 dargestellt aus. Die als »available« markierten Bereiche können dann vom Betriebssystem benutzt werden, um den Ort zu speichern, an dem sich das ausgelagerte Segment befindet.

D/B

Je nach Typ des Segmentes hat das Bit 22 des zweiten Doppelwortes des Segment-Deskriptors unterschiedliche Bedeutung und damit auch einen unterschiedlichen Namen (D bzw. B Flag). Auch auf dieses Flag werden wir bei der Besprechung der einzelnen Segmenttypen noch zu sprechen kommen.

AVL

Bleibt noch das Flag AVL, *available*. Es wird durch die Hardware nicht benutzt und steht dem *Betriebssystem* zur freien Benutzung zur Verfügung.



Abbildung 2.4: Speicherabbild eines Deskriptors, der ein als »not present« markiertes Segment beschreibt

2.2.3 Die Betriebsmodi des Prozessors

Das segmentierte Speichermodell mit seinen Segmenten und deren »Beschreibern« führt also recht konsequent und geradlinig zu einem Konzept, in dem man nicht nur mehrere Programme gleichzeitig im Speicher halten und damit durch geeignete Mechanismen quasi gleichzeitig ausführen kann (»multi-tasking«), sondern auch dazu, dass sich diese Programme nicht gegenseitig ins Gehege kommen und stören können. Das Konzept, das diese Dinge ermöglichte, setzte neben einem neuen, multi-tasking-fähigen Betriebssystem auch die hardwareseitige Unterstützung voraus. Intel nahm diese Herausforderung mit der Realisierung des protected mode als Betriebsmodus für seine Prozessoren an.

Protected Mode

Der protected mode wurde mit dem 80286 ins Leben gerufen. Damals wurde der 1-MByte-Adressraum des 8086 auf »wahnsinnige« 16 MByte aufgebohrt, indem die Anzahl der Adressleitungen auf 24 erhöht wurde (2^{24} = 16.777.216 = 16 M). Allerdings hatten die Register des 80286

immer noch »nur« 16 Bit Breite, sodass das »Standarddatum« durch Words gebildet wurde. Die Segmentgrößen konnten daher maximal 64 KByte erreichen (mit Words waren nur 16-Bit-Offsets realisierbar). Aus dieser Zeit stammt auch noch die etwas merkwürdige Zerstückelung der Segment-Deskriptoren: Das segment limit konnte ebenfalls nur mit 16 Bit codiert werden und »passte« damit in das »unterste« Word der Speicherabbildung. Die base address musste zwangsläufig gestückelt werden: Bit 0 bis 15 kam ins zweitunterste Word und Bit 16 bis 19 in die ersten vier Bits des dritten Words. Die verbleibenden 12 Bits des dritten Words nahmen dann die »Attribute« des Segmentes auf (vgl. Abbildung 5.56 auf Seite 898). Word #4 des Deskriptors existierte zwar bereits (aus Gründen der Prozessor-Architektur und Datenstruktur, die geradzahlige Vielfache von Words forderte), war aber auf »0« gesetzt und wurde nicht benutzt.

Mit Einführung des ersten 32-Bit-Prozessors 80386 wurde dann ein neuer Deskriptor notwendig, der die erweiterten Möglichkeiten berücksichtigte. Aus Kompatibilitätsgründen zu seinem Vorgänger wurde jedoch die Struktur der »alten« Deskriptoren beibehalten, was dazu führte, dass das bislang unbenutzte vierte Word des Deskriptors die verbleibenden 4 Bits für das 20-Bit-Limit und die verbleibenden 8 Bits für die 32-Bit-Basisadresse aufnehmen musste. Die restlichen vier Bits konnten für die neuen, zusätzlichen Attribute herangezogen werden. Denn nun musste ja die Granularität der Segmente gespeichert werden können sowie die Frage, ob die Segmente 16- oder 32-bittig zu interpretieren sind. (Auch dies ist eine Notwendigkeit, die aus der Abwärtskompatibilität resultiert: Da der 80286 trotz seiner 24 Adressleitungen immer noch ein echter 16-Bit-Prozessor war, konnten ab dem 80386 bestimmte Aspekte des protected mode, wie *gates*, die wir weiter unten noch kennen lernen werden, oder die Organisation der Datensegmente sowohl in einer 16-Bit-Welt – 80286-kompatibel – wie auch in einer 32-Bit-Welt angesiedelt sein.)

Real Mode Verlassen wir für einen Moment den protected mode! Traditionell und aufgrund der von Intel bislang strikt eingehaltenen Abwärtskompatibilität der Prozessoren verschiedener Generationen verfügen diese heute über verschiedene Betriebsmodi. Begonnen hat alles mit dem Modus, in dem die ersten Prozessoren vom Typ 8086 und 8088 aus diesem Hause liefen: dem *real mode.*

Im letzten Kapitel wurde behauptet, dass auch dann, wenn aufgrund der Hardwarevoraussetzungen ein »genügend großer« Adressraum

direkt (»linear«) ansprechbar ist, aufgrund der Einführung von Schutzkonzepten Speichersegmentierung eine wünschenswerte Angelegenheit ist. Doch gibt es auch noch andere Gründe, die einen dazu bringen, Speicher zu segmentieren?

Segmente, die Zweite!

Ja, die gibt es! Weiter oben sollten Sie sich vorstellen, dass Sie 32 Adressleitungen haben, mit denen Sie den nutzbaren Adressraum aufspannen können. Diesen 32 Adressleitungen entsprechen 32-Bit-Register, in denen die dazugehörigen 32-Bit-Adressen verwaltet werden können. Jetzt stellen Sie sich für den Moment einmal vor, Sie hätten »nur« 20 Adressleitungen. Und die Register, die Adressen aufnehmen können, sind 16 Bit groß. (Wer denkt nun an den 8086 von Intel?) Dann können Sie mit Ihren Adressregistern nur 2^{16} = 65.536 Byte = 64 KByte große Bereiche des 2^{20} = 1.048.576 Byte = 1 MByte großen Adressraum ansprechen! *In dieser Situation sind Sie gezwungen, den Adressraum zu segmentieren, da Sie mit Ihren Adressregistern nur Teile des verfügbaren Raumes, eben die Segmente, ansprechen können!* Diese Segmente sind hier bis zu 64 KByte groß[1]. Doch wie könnte diese Segmentierung dann realiter aussehen?

Nehmen wir zunächst den Fall an, dass alle sechzehn 64-KByte-Segmente, die Sie in 1 MByte unterbringen können, artig hintereinander angeordnet sind, ohne sich zu überlappen. Dann beginnt Segment 0 an der Basisadresse 0 (= $\$0_0000^2$) und reicht, da 64 KByte groß, bis Adresse 65.535 (=$0_FFFF), Segment 1 beginnt an Basisadresse 65.536 (= $1_0000) und reicht bis 131.071 (= $1_FFFF), Segment 2 erstreckt sich von Basisadresse 131.072 (= $2_0000) bis 196.607 (= $2_FFFF) usw. bis das letzte Segment, Segment 15, an Adresse 983.040 (= $F_0000) beginnt und bei 1.048.575 (= $F_FFFF) endet. Das bedeutet also, dass zu den jeweiligen 16-Bit-Offsets $0000 bis $FFFF, mit denen Sie sich innerhalb eines Segmentes bewegen können, jeweils eine 20-Bit-Basisadresse ($0_0000, $1_0000, ..., $F_0000) addiert werden muss, um über den gesamten Adressraum verfügen zu können.

---

1. Dem aufmerksamen Leser wird nicht entgangen sein, dass beim 80286 neben der Forderung nach Schutzmechanismen die gleichen Gründe zur Speichersegmentierung ebenfalls vorlagen: Auch er hatte nur 16-Bit-Register. Das führte zum »16-Bit-Protected-Mode«.

2. Wenn ich hier mit den unter »Datenformate« angegebenen, zwecks Übersichtlichkeit selbst geschaffenen Konventionen der Auffüllung von Daten mit führenden Nullen gemäß ihrer Größe breche, so nur, um die zur Verwendung stehenden 20 Bit (= 2½ Bytes) deutlicher darzustellen.

Dilemma

Doch wo halten Sie diese 20-Bit-Basisadressen? Im protected mode beinhalten die 16-Bit-Segmentregister, wie wir noch sehen werden, Selektoren auf Segment-Deskriptoren, in denen die Basisadresse des Segmentes verzeichnet ist. Wie Sie sich erinnern werden, wurden diese Deskriptoren eingeführt, um Segmente genauer definieren und auf diese Weise ein Schutzkonzept einführen zu können. Zu Zeiten von DOS und dem 8086 dachte noch niemand an Schutzkonzepte, da sich damals niemand vorstellen konnte, dass einmal mehr als ein Programm gleichzeitig im Speicher residieren könnte (»single task systems«) – wozu auch? Und tatsächlich war DOS ja auch ein Betriebssystem, das kein Multitasking beherrschte und nur ein aktives Programm zu jedem Zeitpunkt zuließ – was ja dann auch zu den TSRs (»*terminate but stay resident programs*«) und dem Verbiegen von Interrupts auf eigene Module mit allen daraus resultierenden Konsequenzen führte, um zumindest ein wenig mehr an Programmierfreiheiten zu bekommen. Wozu also Schutzkonzepte? Der einzige Berechtigungsgrund für Segment-Deskriptoren war also nicht gegeben. Daher mussten die Segmentregister nicht Selektoren auf (damals noch gar nicht angedachte) Segment-Deskriptoren beherbergen, sondern konnten direkt die Basisadressen aufnehmen.



Bitte nochmals, da das wichtig ist! Im real mode enthalten die Segmentregister die *Basisadressen der Segmente selbst*, während sie im protected mode auf *Segment-Deskriptoren* zeigen, in denen dann die Basisadresse des Segmentes steht und aus dem sie ausgelesen werden muss, um eine korrekte Adresse berechnen zu können.

Quadratur des Kreises

Doch trotz der Aufteilung der Adresse auf zwei Anteile, Segmentadresse und Offset, blieb das Dilemma: Wie packte man eine 20-Bit-Basisadresse des Segmentes in ein 16-Bit-Segmentregister? Einfach: Indem man sie durch 16 teilte und damit von 20 auf 16 Bit reduzierte. Das Problem der Adressberechnung im real mode war also einfach lösbar: Inhalt von einem Segmentregister (= Segment-Selektor) mit 16 multipliziert ergibt die Basisadresse des Segments, zu der man einen 16-Bit-Offset addiert, um jede Speicherstelle im Adressraum ansprechen zu können.



Üblicherweise erfolgt die Darstellung einer vollständigen Adresse nach der Konvention, dass zunächst »das Segment« angegeben wird, dem

dann ein Doppelpunkt folgt, dem sich der Offset-Anteil der Adresse anschließt:

Logische Adresse = Segment : Offset

Hierbei wird für das Segment der Segment-Selektor, also die durch 16 dividierte Basisadresse des Segmentes verwendet, so wie sie auch in den Segmentregistern verzeichnet ist:

Logische Adresse = $4C00 : 03DF

Da dieser Segment-Selektor in einem Segmentregister steckt, ist es auch legitim, das entsprechende Register anstelle des Selektors anzugeben, wenn es diesen Selektor enthält. Wenn also z.B. das Datensegment-Register DS den Selektoren $4C00 für das Datensegment enthält und man auf die Speicherstelle $03DF in diesem Segment zugreifen will, sind folgende Darstellungen identisch:

Logische Adresse = $4C00 : $03DF
Logische Adresse = DS : $03DF



Diese Darstellung ist allgemeingültig und nicht auf den real mode begrenzt. Allgemein werden logische Adressen in der Form Segmentregister: effektive Adresse angegeben. Unterschiedlich in den einzelnen Betriebsmodi sind lediglich die Mechanismen, die hinter der eigentlichen Adressberechnung stecken. So wird im real mode, wie gesagt, die logische Adresse gebildet, indem der Selektor (Inhalt des Segmentregisters) mit 16 multipliziert wird und dann der Offset, die »effektive Adresse«, dazu addiert wird. Im protected mode ist der Selektor, wie wir noch sehen werden, ein Zeiger in eine Deskriptoren-Tabelle, der auf einen zum Segment dazugehörigen Deskriptor zeigt. Dieser enthält die Basisadresse des Segmentes, zu der dann der Offset, sprich die effektive Adresse, addiert wird.

Wie groß ist das Segment?

Doch zurück zu den Segmenten des real mode. Denken wir ein bisschen weiter! Das Fehlen von Segment-Deskriptoren hat als Konsequenz, dass es nirgendwo Informationen darüber gibt, wie groß das Segment eigentlich ist. Gut – wir wissen, es kann maximal 64 KByte und muss mindestens 16 Byte groß sein, da aufgrund der Berechnung der Basisadressen durch Multiplikation mit 16 Segmente nur bei den *Segmentgrenzen* beginnen und enden konnten, Basisadressen, die Vielfache von 16 sind. Doch zwischen 16 Byte und 64 KByte liegen viele, viele Bytes! Wie groß ist das uns interessierende Segment nun? Antwort: *Wis-*

sen wir nicht! Wissen wir wirklich nicht. Es gibt außer dem Programmierer des jeweiligen, auf den Segmenten basierenden Programms niemanden, der uns dies sagen könnte.

Wo beginnt das Segment?

Nächstes Problem: Als Basisadressen brauchen wir eigentlich »nur« Werte zwischen $0_0000 und $F_0000 mit Inkrementen von $1_0000, um die Segmente, wie oben angenommen, hintereinander anzuordnen. Dividieren wir das Ganze durch 16 und führen es von der Basisadress-Ebene auf die Selektor-Ebene, so heißt das, dass die Segmentregister nur Werte zwischen $0000 und $F000 mit Inkrementen von $1000 annehmen brauchen, die »unteren« 12 der 16 Bits also getrost auf »0« gesetzt werden können. Doch hat niemand verboten, genau dies *nicht* zu tun und in Segmentregister auch z.B. den Wert $4C00 (oder noch schlimmer: $4C7D!) zu schreiben. Nach Berechnung der Basisadresse des zugehörigen Segmentes durch Multiplikation mit 16 resultiert hieraus die Segmentadresse $4_C000. Wie ist das nun zu interpretieren?

korrupte Offsets

Zum einen, wenn wir das Bild von oben mit den 16 artig hintereinander aufgereihten Segmenten à 64 KByte Größe beibehalten, als 5. Segment mit der Basisadresse $4_0000, das aber bereits einen vorgegebenen »Start«-Offset von $C000 hat, zu dem dann der eigentliche Offset noch dazu addiert wird. Das aber hat Konsequenzen! Denn erstens, nachdem es keine negativen Offsets gibt, kann man auf die ersten $BFFF Bytes des Segmentes nicht mehr zugreifen. Und zweitens kann der Offset ja Werte bis $FFFF annehmen, was bedeutet, dass die resultierende Adresse bis $5_BFFF gehen kann *und man damit durch geeignete Wahl des Offsets ein Segment verlassen und weit in das nächste Segment hineingehen kann*! Es müsste also der Offset auf einen Maximalwert von $3FFF begrenzt werden, wenn er artig innerhalb des Segmentes bleiben soll. Doch wer tut das, wer prüft das, wer verhindert einen Missbrauch? DOS sicherlich nicht!

korrupte Segmente

Für eine weitere Interpretation dieser »ungewöhnlichen« Segmentadresse $4_C000 geben wir das Bild der geordneten Segment-Kette auf, nehmen aber weiterhin an, dass Segmente 64 KByte groß sind. Das aber heißt dann, dass Segmente eben nicht sauber *in line* liegen müssen, sondern sich sehr wohl überlappen können: Dann kann ein Segment ($4_C000) mitten in einem anderen Segment beginnen ($4_0000) oder enden ($5_0000). Dies aber öffnet die Tore zu bewussten oder unbewussten Zugriffsverletzungen.

Segment-Inflation

Schließlich ist eine weitere mögliche Interpretation, dass es erheblich mehr Segmente als die oben genannten 16 geben kann, nämlich 2^{16} = 65.536, die dann nicht mehr notwendigerweise 64 KByte groß sind (aber sein können!) und sich zwangsläufig mehr oder weniger überlappen müssen, sofern man den jeweiligen Offset nicht auf maximal 15 und die Segmentgröße dadurch auf 16 festlegt. Doch wer sollte das tun? DOS auf keinen Fall!

Weil Schutzkonzepte fehlten und die Adressberechnung nicht eindeutig war, konnte (fast) jedes Segment auf (fast) jedes Segment zugreifen und ordentlich Unruhe stiften, was dann ja auch ausgiebig geschah! So wurde beispielsweise häufig direkt auf den Bildschirmspeicher zugegriffen, weil die Systemroutinen zur Bildschirmausgabe, die das Betriebssystem DOS, das im real mode arbeitete, zur Verfügung stellte, ziemlich langsam und unbeholfen waren.

Trotzdem hat der real mode auch heute noch und selbst unter Windows 2000 eine wichtige, wenn auch zeitlich sehr begrenzte Bedeutung! Denn die Initialisierung nach dem Einschalten oder einem Reset des Prozessors lässt ihn im real mode anfahren. Es ist dann Aufgabe des (Real-Mode-)Betriebssystem-Laders (also eines Teils des Betriebssystems!), in den protected mode umzuschalten, was in der Regel auch unmittelbar erfolgt.



Vielleicht erinnern Sie sich daran, dass man zu DOS-Zeiten ganz stolz war, »doch etwas« über die magische 1-MByte-Grenze hinauskommen zu können. Voraussetzung war allerdings, dass man mindestens einen 80286er hatte. Denn da der 8086 nur 20 Adressleitungen hatte, konnte der Addierer, der Segmentadresse und Offset addierte, nicht über 2^{20} – 1 = $FFFFF addieren. Weil aber sowohl für Segmentadresse als auch für den Offset jeweils $FFFF gültige Werte waren, kam der Addierer bei der Berechnung 16 · $FFFF + $FFFF = $10FFEF über das Maximum $FFFFF hinaus und führte daher einen automatischen »wrap-around« aus, indem er die führende »1« einfach vergaß. Somit war die Adresse $FFFF:$FFFF identisch mit der Adresse $0000:$FFEF.

Beim 80286 allerdings standen 24 und ab dem 80386 gar 32 Adressleitungen zur Verfügung. Der Adressaddierer musste (konnte) also keinen wrap-around durchführen, weshalb die Adresse $10FFEF auch unter DOS tatsächlich berechnet und angesprochen werden konnte. Dies führte dazu, dass man den DOS-Adressraum um knapp 64 KByte (exakt: 65.519 statt 65.536) über 1 MByte aufbohren konnte. Allerdings

hatte man (Intel!) eine Möglichkeit geschaffen, Kompatibilität herzustellen: Durch ein »Gatter« an der Adressleitung A20 (an der die führende »1« anliegen würde) konnte diese permanent auf »0« gesetzt werden, egal, was die Adressberechnung ergab, und somit ein wrap-around erzwungen werden. Gesteuert hat dies der Tastatur-Kontroller in Verbindung mit dem Treiber HIMEM.SYS.

Virtual 8086 Mode

Sicherlich werden Sie vom »virtual 8086 mode« gehört haben. Was ist das? Der Teil »8086« lässt vermuten, dass er etwas mit dem real mode zu tun haben könnte. Richtig! Er hat aber auch etwas mit dem protected mode zu tun. Vereinfacht ausgedrückt: Der virtual 8086 mode ist ein Modus, der die gleichen Schutzkonzepte wie der protected mode implementiert, aber eine Umgebung schafft, wie sie im real mode herrscht. Somit ist der V86-Modus, wie er auch genannt wird, ein in den protected mode »eingebetteter« real mode.

Die bis in die Tage des Pentium (1993) hineinreichende große Bedeutung des Real-Mode-Betriebssystems DOS ließ Intel diesen virtual 8086 mode schaffen, durch den DOS-Programme in einem *virtuellen* real mode ablaufen konnten. Das bedeutet, dass immer dann aus dem protected mode in den virtual 8086 mode umgeschaltet wird, wenn unter Windows eine DOS-Box aufgemacht wird oder DOS-Programme gestartet werden. Der virtual 8086 mode verzichtet hierbei weder auf die Schutzkonzepte des protected mode, noch auf dessen Möglichkeiten wie Multitasking und Paging. Das bedeutet, dass die DOS-Box selbst ein protected mode task ist und sich so verhält. Aber in ihrem Inneren, da wo das DOS-Programm abläuft, simuliert sie eine Umgebung, die das DOS-Programm für den real mode hält. (Weshalb der Modus auch »virtual 8086« heißt!) Und nachdem es ein »normaler« protected mode task ist, kann er auch mehrfach aufgerufen werden, was zu mehrfachen DOS-Boxen oder quasi-gleichzeitig ablaufenden DOS-Programmen führen kann (aber nicht notwendigerweise muss: unter Windows 9.x/ME laufen alle DOS-Programme in einer DOS-Box ab!).



Schutzkonzepte implementiert der virtual 8086 mode nur »nach außen«! Das bedeutet, dass »innerhalb« des virtuellen real mode keinerlei Schutz besteht: DOS-Programme können sich wie im »wirklichen« real mode das Leben sehr schwer machen! Schutz erfolgt nur in der Weise, dass diese virtuelle Umgebung gegen die reale Umgebung, in der sie abläuft, abgeschirmt wird: Ein fehlerhaftes DOS-Programm kann die

virtuelle Umgebung killen, nicht aber andere Protected-Mode-Anwendungen.



Neben den Hardware-Voraussetzungen muss natürlich auch das Betriebssystem diesen virtuellen real mode unterstützen. Dies erfolgt in der Regel über einen »virtual 8086 monitor«, dessen Aufgabe es ist, z.B. die Interrupts oder Portzugriffe abzufangen und in Protected-Mode-gerechte Anforderungen an einen anderen Teil des Betriebssystems weiterzureichen, der sie dann emuliert: die »8086-Services«. Da die Nutzung des virtual 8086 mode eine Fähigkeit des Betriebssystems ist und wir im Rahmen dieses Buches das Betriebssystem nicht antasten wollen, belassen wir es bei dieser kurzen Hintergrundinformation.

System Management Mode

Nur der Vollständigkeit halber sei ein weiterer Betriebsmodus erwähnt, der *system management mode* (SMM). Es ist ein Modus, der unabhängig *neben* den eigentlichen Betriebsmodi real mode, protected mode oder virtual 8086 mode existiert. Er dient einem sehr speziellen Zweck: Über ihn ist z.B. eine Form des *power management* des Rechners realisiert, über ihn kann die Hardware kontrolliert werden oder es kann Code aufgerufen werden, der für den speziellen Rechner, in dem der Prozessor steckt, geschrieben wurde (»proprietary OEM code«).

Es gibt weder für das Betriebssystem noch für Anwendungsprogramme die Möglichkeit, den SMM aufzurufen. In ihn kann man nur durch Hardware über einen speziellen, nicht maskierbaren Interrupt gelangen, der ausgelöst wird, wenn ein Signal an einen speziell für diesen Zweck reservierten Pin des Prozessors angelegt wird. Aus diesen Gründen wird auf eine weitere Besprechung dieses Betriebsmodus verzichtet.

2.2.4 Segmenttypen, Gates und ihre Deskriptoren

Kommen wir zurück zur Speichersegmentierung. Der kurze Ausflug in die verschiedenen Betriebsmodi, die der Prozessor unterstützt, hat gezeigt, dass der Modus, in dem die Prozessoren von heute arbeiten, der protected mode mit seinen Schutzkonzepten und Möglichkeiten wie Multitasking und Paging ist. Wir werden daher auch im Folgenden diesen Modus behandeln.

Bei der Vorstellung der verschiedenen Segmente wird der Begriff »Privilegstufe« auftreten. Er ist Teil der Schutzkonzepte, die der protected mode gestattet, und wird daher in dem entsprechenden Kapitel bespro-

chen werden. Für den Augenblick mag genügen, dass es verschiedene Stufen an Zugriffsrechten gibt, die man Privilegstufen nennt: je höher die Privilegstufe, desto privilegierter ist ein Programm und desto mehr Zugriffsrechte hat es.

Codesegmente und Codesegment-Deskriptoren

Weiter oben wurden einige Typen von Segmenten angesprochen, die bestimmte Aufgaben haben. So werden alle Codeteile eines Programms in ein Segment gesteckt, dass man sinnigerweise Codesegment nennt. Bitte beachten Sie hierbei, dass zwar niemand verbietet, mehrere Codesegmente zu definieren und innerhalb eines tasks durch entsprechende Versionen des CALL-Befehls zwischen diesen unterschiedlichen Codesegmenten hin- und herzuspringen. Doch macht das wenig Sinn: Aufgrund der Möglichkeit, mit den 32 Bits eines Adressoffsets den gesamten verfügbaren Adressraum anzusprechen, gibt es keinen Grund dafür, mehrere Codesegmente zu erzeugen, aber viele dagegen. Einer davon ist, dass ein Intersegment-CALL auch nichts anderes machen würde als ein Intrasegment-CALL eines entsprechend größeren Segmentes, aber (u.a. aufgrund der Einbindung der Schutzkonzepte) wesentlich zeitaufwändiger wäre und damit nicht zu einer Verbesserung der Performance beitrüge. (Das heißt natürlich nicht, dass ein Prozess nicht verschiedene Codesegmente haben kann. Immerhin setzt er sich ja aus verschiedenen Komponenten zusammen, unter anderem aus dem Code der Anwendung, eventuellen DLLs und anderen Bibliotheken und Betriebssystemteilen. Diese residieren natürlich alle in eigenen Codesegmenten. Aber jedes einzelne dieser Puzzleteile hat in der Regel nur ein Codesegment.)

In Codesegmenten liegt also ausführbarer Code – und nichts anderes. Das Segment selbst darzustellen ist auf der einen Seite langweilig, da es sich ja lediglich um eine Abfolge von Bytes handelt, die auf den ersten Anschein wahllos aneinander gereiht sind (und hinter deren Struktur und Sinn man erst kommt, wenn man akribisch die Bytes auswertet, in Opcodes und Operanden aufteilt und in Mnemonics disassembliert); und andererseits müßig, da die Inhalte von Codesegmenten so kunterbunt und flüchtig sind wie Seifenblasen.

Sehr viel spannender ist es, einen Segment-Deskriptor darzustellen, den die Spezialisierung dieses Segmentes erforderlich macht und dem Rechnung trägt. Denn er ist sehr strukturiert. In Abbildung 2.5 ist er dargestellt.

Die Felder *segment limit*, *base address* und *DPL* sowie die Flags *present* und *granularity* haben wir bereits weiter oben bei der Vorstellung des grundsätzlichen Aufbaus eines Segment-Deskriptors besprochen. Auch zeigt der Wert des Systemflags S an, dass es sich um kein Systemsegment handelt (S = 1), sondern, wie das »oberste« Bit des Feldes *type* anzeigt, um ein Codesegment (Bit 11 = 1). Die drei restlichen Bits des Type-Feldes repräsentieren in diesem Segmenttyp die von einander unabhängigen Flags C, R und A. Außerdem hat Bit 22 des zweiten DoubleWords die Bedeutung »D«.

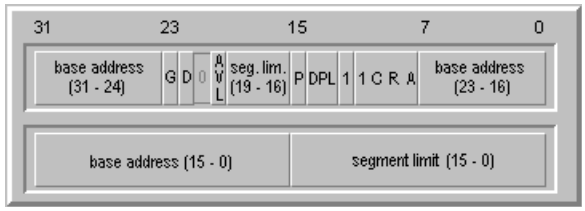


Abbildung 2.5: Speicherabbild eines Codesegment-Deskriptors

default length flag

Das Flag D, *default length*, zeigt die Standardgröße der Operanden von Befehlen und von Adressen an. Ist D gesetzt, so wird standardmäßig mit 32-Bit-Adressen in 32-Bit-Registern gearbeitet und Operanden der Instruktionen haben entweder die Größe 8 Bit (Byte) oder 32 Bit (DoubleWord). Bei gelöschtem D-Flag wird mit 16-Bit-Adressen in 16-Bit-Registern gerechnet und die Operanden umfassen entweder 8 Bit (Byte) oder 16 Bit (Word). In beiden Fällen kann mit Hilfe der Befehlspräfixe *address size override* und/oder *operand size override* diese Standardvorgabe für den folgenden Befehl außer Kraft gesetzt werden.



Bitte beachten Sie, dass die Präfixe je nach Wert des Flags D entgegengesetzte Bedeutung haben. So ist bei gesetzten Flag D, wie gesagt, die Standardsituation »32-Bit-Adressen« und »8-/32-Bit-Operanden«. Ein vorangestellter Präfix *address size override* definiert für den folgenden Befehl (und nur diesen!) die Situation »16-Bit-Adressen« und »8-/32-Bit-Operanden«, »erniedrigt« also die Adressengröße, während ein vorangestellter Präfix *operand size override* die Situation »32-Bit-Adressen« und »8-/16-Bit-Operanden« einstellt, also die Operandengröße »erniedrigt«. Nur dann, wenn beide Präfixe verwendet werden, würden 16-Bit-Adressen und 8-/16-Bit-Operanden verwendet, also insgesamt ein Zustand hergestellt, als ob für den nächsten Befehl das D-Flag gelöscht würde. Bei gelöschtem D-Flag dagegen ist die Standardsituation

»16-Bit-Adressen« und »8-/16-Bit-Operanden«. Der Präfix *address size override* »erhöht« hier die Adressgröße, indem er die Verwendung von 32-Bit-Adressen erzwingt, während weiterhin 8-/16-Bit-Operanden zum Tragen kommen. Analog führt die Verwendung des *operand size override prefix* zur Nutzung von 16-Bit-Adressen und einer »Erhöhung« der Operandengröße auf 8-/32-Bit-Operanden. Nur dann, wenn wiederum beide Präfixe verwendet werden, werden 32-Bit-Adressen und 8-/32-Bit-Operanden. benutzt, was einem temporären (und nur auf den folgenden Befehl beschränkten) Setzen des D-Flags entspricht.

accessed flag Das accessed flag A wird immer dann gesetzt, wenn ein Zugriff auf das Segment erfolgt. Es muss vom Betriebssystem explizit gelöscht werden, andernfalls bleibt es gesetzt. Das A-Flag kann daher von bestimmten Programmen (Debugger, Betriebssystem) für bestimmte Zwecke benutzt werden.

read enable flag Codesegmente enthalten üblicherweise nur ausführbaren Code, der nur den Prozessor zu interessieren hat. Es macht daher keinen Sinn, anderen Programmteilen einen lesenden Zugriff auf das Segment zu gestatten. Codesegmente, die diesen Zugriff verbieten, nennt man *execute-only code segments*. Sie liegen vor, wenn das *read enable flag* R gelöscht ist. Ist es dagegen gesetzt, wird ein lesender Zugriff erlaubt. In diesem Fall spricht man von *execute/read code segments*. Sie machen dann Sinn, wenn ein Segment sowohl Code als auch Daten enthält, was selten der Fall sein dürfte. Ein Beispiel hierfür sind Code und Daten, die in einem ROM/EPROM untergebracht sind. Solche Segmente können lesend entweder mit einem *segment override prefix* CS: für Speicherzugriffsbefehle (MOV) angesprochen werden oder durch Laden des Codesegment-Selektors in eines der Datensegment-Register (DS, ES, FS, GS).



Im protected mode können maximal lesende Zugriffe auf das Codesegment erfolgen. Ein schreibender Zugriff auf ein Codesegment ist grundsätzlich nicht möglich. Ist in einer besonderen Ausnahmesituation ein Schreiben in ein Codesegment erforderlich, so muss ein Datensegment-Deskriptor erzeugt werden, der als Basisadresse das Codesegment hat und entsprechende Zugriffsrechte vergibt.

conforming flag Es gibt zwei Arten von Codesegmenten: »anpassungsfähige« oder »sich anpassende« (*conforming*) und »nicht-anpassungsfähige« (*non-conforming*). Bei den *non-conforming code segments* ist ein Sprung mittels CALL oder JUMP in dieses Segment nur dann möglich, wenn das anzuspringende Segment die gleiche Privilegstufe hat wie das, aus dem

der Sprung erfolgt, es sei denn, man springt es über ein *call gate* an. (Was das ist, werden wir gleich sehen!) Andernfalls wird eine *general protection exception* #GP ausgelöst. Non-conforming segments sind daher tatsächlich sehr »starr« und wenig an die Situation anpassungsfähig. Anders ist das bei *conforming code segments*. Sie passen sich der Privilegstufe des rufenden Codes an und erlauben so den Ansprung aus einer niedriger privilegierten Stufe. Einzelheiten hierzu entnehmen Sie bitte dem Kapitel »Schutzmechanismen« ab Seite 467.



Mittels CALL oder JUMP können grundsätzlich nur Segmente gleicher (non-conforming) oder höherer (conforming) Privilegstufen angesprungen werden. Ein JUMP oder CALL in ein niedrigerer privilegiertes Codesegment führt in jedem Fall zu einer general protection exception #GP, egal ob das Zielsegment conforming ist oder nicht. Ist ein solcher Sprung erforderlich, so muss er über ein call gate erfolgen.

Datensegmente und Datensegment-Deskriptoren

Der andere wichtige Segmenttyp neben Codesegmenten ist das Datensegment. Auch hier gilt das, was bei der Besprechung von Codesegmenten bereits gesagt wurde: Es besteht zwar theoretisch die Möglichkeit, verschiedene Datensegmente zu erzeugen und zu benutzen. Doch es macht in praxi genauso wenig Sinn wie bei Codesegmenten. (Ein Grund mag jedoch bestehen: Manche Hochsprachen-Programme unterscheiden zwischen Datensegmente für »uninitialisierte« Daten, also Variablen, die ihren Inhalt frühestens mit dem Start des Programms erhalten, und in Datensegmente für »initialisierte« Daten, also Konstanten, bei denen der Inhalt bereits vor dem Programmstart bekannt und im Speicherabbild bereits abgelegt ist. Häufig sind solche »Konstanten-Datensegmente« auch noch gegen schreibenden Zugriff geschützt. In diesem Fall macht es nicht nur Sinn, sondern ist sogar erforderlich, zumindest zwei unterschiedliche Datensegmente zu definieren. Und noch eines können wir ungeprüft von den Codesegmenten übernehmen: Die Darstellung eines Datensegments an dieser Stelle ist wenig sinnvoll. Daher wenden wir uns auch hier lieber den Deskriptoren für Datensegmente zu.

Auch Datensegmente haben über die bereits bei der allgemeinen Definition der Segmente besprochenen (segment limit, base address, DPL, P- und G-Flag) bestimmten Eigenschaften, die sie von anderen Segmenten unterscheiden und daher einen eigenen Deskriptor erfordern. In Abbildung 2.6 ist dieser Datensegment-Deskriptor dargestellt.

Bit 22 des zweiten DoubleWords hat hier die Bedeutung »B«, was für *big flag* steht. Das S-Flag ist gesetzt und signalisiert somit ein Nicht-Systemsegment, das das gelöschte Bit 11, das »oberste« Bit des *type fields*, als Datensegment ausweist. Die anderen Bits des type fields sind voneinander unabhängig und haben in solchen Segmenten die Bedeutung *expansion direction flag* (E), *write-enable flag* (W) und *accessed flag* (A).



Abbildung 2.6: Speicherabbild eines Datensegment-Deskriptors

big flag

Analog dem default length flag D in Codesegmenten gibt das big flag B in Datensegmenten an, wie das Datensegment organisiert ist. Ist B gesetzt, so ist das Segment 32-bittig organisiert, bei gelöschtem B 16-bittig. Wichtig ist diese Information vor allem bei Stacksegmenten, bei dynamischen Datensegmenten, die »nach unten« wachsen (siehe expansion direction flag) und bei der Prüfung, welche Standardgröße die Operanden von Befehlen haben, die auf dieses Segment zugreifen. So ist bei gesetzten big flag die Standard-Operatorengröße für Befehle mit Speicheroperanden 32 Bit; bei Verwendung des *operand size override prefix* wird sie dann ggf. für den folgenden Befehl auf 16 Bit reduziert. Bei gelöschtem big flag dagegen ist die Standard-Operatorengröße 16 Bit, der operand size override prefix erhöht sie für den folgenden Befehl auf 32 Bit.

expansion direction flag

Datensegmente sind häufig dynamische Strukturen, was bedeutet, dass sie während ihrer Existenz wachsen und schrumpfen können. Hierbei verändert sich der Eintrag im Feld *segment limit*. Tun sie das, so können sie in zwei Richtungen wachsen: wie Stalagtiten in Tropfsteinhöhlen »von oben nach unten« oder wie Stalagmiten »von unten nach oben«. Das expansion direction flag E codiert diese »Wachstumsrichtung«. Ist es gelöscht, so haben wir ein »normales« Datensegment, das »unten« eine feste Basis an der Adresse des ersten Bytes des Segments (»segment base«) hat und an dem das dynamische Wachsen oder Schrumpfen an seiner Spitze (»segment limit« = Adresse des letzten Bytes des Segments) bei höheren Offsets erfolgt. Das Wachsen äußerst sich darin, dass der Inhalt des Feldes segment limit größere Werte annimmt, beim Schrumpfen nimmt es kleinere Werte an. Solche Segmente nennt man

expand-up segments, da sie »nach oben«, zu höheren Adressen expandieren. Bei der Integritätsprüfung eines Offsets in dieses Segment wird geprüft, ob der Offset kleiner oder gleich dem segment limit ist, was für einen integren Offset spricht. Ist er dagegen größer als das segment limit, ist er korrupt und löst eine general protection exception #GP aus. Die Stellung des big flags spielt in diesem Falle keine Rolle, da unabhängig von der »Körnung« des Datensegments (16- oder 32-bittig) der Offset immer nur zwischen 0 (Segmentbasis) und segment limit liegen kann.

Anders ist das bei *expand-down segments*, bei denen das E-Flag gesetzt ist. Hier ist die Basis (nicht zu verwechseln mit der »segment base«, also der Adresse des physikalisch ersten Bytes des Segments!) wie bei Stalagtiten »oben«, sprich bei einem maximal möglichen Offset für das Segment. Die Spitze des Segments liegt wiederum an der Stelle, die durch segment limit definiert wird. Wächst das Segment, so kann es nur zu niedrigeren Adressen ausgedehnt werden, denn die Basis ist ja oben! Solche Segmente wachsen, indem das segment limit kleinere Werte annimmt, und sie schrumpfen, wenn es größere Werte annimmt.

Hierbei stellt sich nur ein Problem: An welcher Adresse liegt die Basis? Bei expand-up segments ist sie identisch mit der Adresse, die in segment base angegeben werden kann. Hier ist also die »Segmentbasis« im wahrsten Sinne der Übersetzung die segment base. Hat also segment base im Falle von expand-down segments ebenfalls die Adresse der Basis, nur dass sie nun oberhalb des im segment limit stehenden Werts liegt? *Nein!* Segment base gibt immer noch die Adresse des, absolut und physikalisch betrachtet, niedrigstwertigen Bytes des Segments an und sollte nun eigentlich nicht mehr segment base, sondern *segment maximum* heißen. Es ist die Adresse des Bytes, bis zu dem das nach unten wachsende Segment sich maximal ausdehnen kann, ohne mit (physikalisch betrachtet) »darunter« liegenden Segmenten zu kollidieren.

Das bedeutet, im Falle von Expand-Down-Segmenten ist segment base = »Segmentmaximum«. Umgekehrt sollte dann doch Segmentbasis = »segment maximum« sein, also die Adresse des (physikalisch betrachtet) maximal adressierbaren Bytes. Bei 16-Bit-Adressen wäre dies der Wert $FFFF, bei 32-Bit-Adressen $FFFF_FFFF!

Und richtig: Die Segmentbasis nach unten expandierender Segmente liegt je nach Körnung des Segmentes bei $FFFF, wenn mit 16 Bit gearbeitet wird (B = 0), und $FFFF_FFFF im Falle von 32-Bit-Segmenten (B = 1).

Das Feld segment limit gibt, wie gesagt, wiederum die Spitze des Segmentes an. Nur liegt diese Spitze jetzt an niedrigeren Adressen als die Basis. Schrumpft das Segment, nimmt segment limit größere Werte an, wächst es, kleinere. Gültige Offsets in ein solchermaßen definiertes Segment liegen also zwischen segment limit und $FFFF oder $FFFF_FFFF. Liegen sie unter segment limit, wird eine general protection exception #GP ausgelöst. Das B-Flag hat hier also eine durchaus wichtige Bedeutung, legt es doch die maximale Größe des Segmentes fest: 64 KByte bei 16-Bit-Körnung bzw. 4 GByte bei 32-Bit-Körnung.



Das hat aber Auswirkungen! Bei der Nutzung von 32-Bit-Datensegmenten kann es nur ein einziges Expand-Down-Datensegment geben! Denn während bei Expand-Up-Segmenten über segment base die Startadresse eines Segments verändert und daher seine Lage im physikalischen Speicher festgelegt werden kann (sie muss nicht immer $0000_0000 betragen), wäre für alle Expand-Down-Segmente die Segmentbasis $FFFF_FFFF, da es kein Feld segment maximum gibt, das analog segment base einen Maximalwert aufnehmen könnte – sie würden überlappen, was der protected mode nicht zulässt.

Bei 16-Bit-Datensegmenten dagegen ist das anders: Da hier der (physikalisch betrachtet) maximale Offset bei $FFFF liegt, könnten zumindest in 32-Bit-Umgebungen wenigstens theoretisch mehrere Expand-Down-Segmente realisiert werden. Aber wer will in 32-Bit-Umgebungen schon 16-Bit-Datensegmente! In 16-Bit-Umgebungen haben wir das gleiche Problem: »Es kann nur eines geben«!



Das ist auch der Grund, warum Stack-Segmente (als Sonderform von Datensegmenten) und alle Datensegmente in der Regel Expand-Up-Segmente sind. So braucht man vor allem im protected mode mehrere Stacksegmente: mindestens eines im Usermodus (Privilegstufe 0) und mindestens eines im Kernelmodus (Privilegstufe 3). Dies ist mit Expand-Down-Segmenten nicht zu machen! Und 16-Bit-Stacksegmente machen in 32-Bit-Umgebungen keinen Sinn.



Übrigens: Statische Datensegmente sind von expand-up segments praktisch nicht zu unterscheiden. Bei ihnen ändert sich lediglich der Inhalt von segment limit im Verlauf der Existenz des Segmentes nicht.

write-enable flag

Datensegmente können nur lesbar (*read-only*) sein, wie das z.B. bei Segmenten, die nur unveränderliche Konstanten aufnehmen, der Fall sein kann. Sie können aber, was der Normalfall ist, auch beschreibbar sein (read/write). Signalisiert wird dies durch das write-enable flag W, das im gesetzten Zustand die Beschreibbarkeit anzeigt.

access flag

Wie im Falle der Codesegmente signalisiert das access flag A, ob auf das Datensegment bereits zugegriffen worden ist oder nicht. So kann z.B. das Betriebssystem diese Information nutzen, um Daten in den Speicher zurückzuschreiben, falls ein Datum verändert wurde.

Stacksegmente

Stacksegmente sind grundsätzlich Datensegmente und verfügen daher über keinen »eigenen« Deskriptor. In jedem Fall muss bei Stacks ein schreibender Zugriff gestattet sein, weshalb auch das write-enable flag gesetzt ist.

Wie eben erläutert, sind praktisch alle Stacksegmente expand-up segments, haben also ein gelöschtes E-Flag.



big flag

Bei Stacksegmenten hat das big flag noch eine wesentliche Bedeutung: Ist es gesetzt, so ist der Stack, wie wir bereits wissen, 32-bittig orientiert. In diesem Fall wird er über das 32-Bit-Stack-Pointer-Register ESP angesprochen. Ist es dagegen gelöscht, ist der Stack 16-bittig orientiert und der stack pointer steht im 16-Bit-»Register« SP.

Stacksegmente werden also nicht irgendwie »definiert«, z.B. durch Setzen eines Flags wie bei Code-, Daten- oder Systemsegmenten! Ein dynamisch wachsendes Stacksegment ist durch nichts von einem expand-down, write-enabled data segment zu unterscheiden. Wer also macht aus einem solchen expand-down, write-enabled data segment ein Stacksegment? Antwort: Der pure Eintrag des Selektors, der auf den Datensegment-Deskriptor zeigt, in das Segmentregister SS. Der gleiche Selektor in Segmentregister DS macht aus dem Stacksegment ein stinknormales Datensegment (allerdings expand-down und write-enabled).



Das heißt aber nun nicht, dass man aus jedem beliebigen Datensegment ein Stacksegment machen kann, indem man den Selektor in das Segmentregister SS einträgt! Denn beim Laden des Selektors wird sehr wohl geprüft, ob das Segment write-enabled ist. Ist es das nicht, so wird die beliebte general protection exception #GP ausgelöst.





Noch einige Anmerkungen, die für Sie hilfreich sein könnten. Sie zielen auf die gerade für den Neuling nicht ganz einfache Problematik »expand-down«, segment limit, segment base, stack pointer, base pointer.

1. Unterscheiden Sie zwischen dem Stack und dem ihn beherbergenden Stacksegment! Ein Stacksegment hat eine Basis (segment base) und eine Ausdehnung (segment limit). Dies sind die physikalischen Grenzen, in denen der Stack angesiedelt ist, sie sind unabhängig vom wachsenden/schrumpfenden Stack! Auch ein Stack hat eine Basis und eine Ausdehnung. Deren physikalische Adressen liegen in base pointer und stack pointer und müssen immer innerhalb des durch das Stacksegment definierten Bereichs liegen.

2. Ein Stack ist grundsätzlich definiert als Datenbereich, der »nach unten« wächst und »nach oben« schrumpft. Das expansion direction flag ändert an diesem Sachverhalt nicht das Geringste, es regelt ein eventuelles Wachsen/Schrumpfen des den Stack beherbergenden Stack*segments*! Denn der wachsende/schrumpfende Stack wird durch das automatische Inkrementieren und Dekrementieren des Stack-Pointers in (E)SP durch den Prozessor realisiert, der den Stackpointer beim Stack-PUSH *immer* dekrementiert, beim Stack-POP inkrementiert. Der Stack wächst also *immer* zu niedrigeren Adressen und schrumpft zu höheren, egal, ob und wie das Segment sich verändert.

3. Hat man es mit statischen Segmenten, in diesem Fall also mit einem statischen Stacksegment zu tun, ist es immer ein Expand-Up-Segment mit gelöschtem E-Flag. In diesem Fall ist die »niedrigste« Adresse für die Stackbasis und die »niedrigste« Adresse, die in (E)SP eingetragen werden kann, die Adresse, die durch segment limit definiert wird und die Spitze des Stacksegments angibt. Die »höchste« Adresse ist dann der in segment base eingetragene Wert, also die Basis des Stacksegments.

4. Ein dynamisches Stacksegment ist nur auf eine Art zu realisieren: als Expand-Down-Segment, da ja ein dynamisch wachsendes Segment nicht an der entgegengesetzten Stelle wachsen kann wie der wachsende Stack – was hätte das für einen Sinn? Und wie bereits erläutert hieße dies: Es gibt nur einen Stack! Bei einem solchen Stack wäre die »niedrigste« Adresse für die Stackbasis und die »niedrigste« Adresse, die in (E)SP eingetragen werden kann, je nach Körnung $FFFF oder $FFFF_FFFF, die »höchste« Adresse die durch segment limit

definierte Adresse. Ein solchermaßen definiertes Stacksegment (und damit auch der beherbergte Stack) könnte bis segment base wachsen.

Systemsegmente und Systemsegment-Deskriptoren

Systemsegmente sind Datenstrukturen, die dem Betriebssystem als Datenbank dienen, mit denen es gewisse Aufgaben erfüllen kann. Es gibt zwei Arten von Systemsegmenten:

- Deskriptor-Tabellen und
- Task-Zustandssegmente (»task state segments«).

Systemsegmente werden durch Deskriptoren beschrieben, die analog den Code- und Datensegment-Deskriptoren aufgebaut sind. In ihnen ist lediglich das *system flag* S auf 0 gesetzt, um zu signalisieren, dass ein Systemsegment beschrieben wird. Das Bit D/B hat keine Bedeutung und ist mit »0« vorbesetzt. Das *type field* dient zur weiteren Unterscheidung der Systemsegmente; durch dieses Feld kann einerseits zwischen den beiden Systemsegmenten im engeren Sinne unterschieden werden, zum anderen werden hierüber die Gates definiert.

Deskriptor-Tabellen

Es gibt zwei Arten von Deskriptor-Tabellen: die global descriptor table GDT, die Deskriptoren enthält, die zu jedem Zeitpunkt und unter allen Umständen verfügbar sein müssen und die sich daher z.B. bei Taskwechseln nicht ändert. Und die local descriptor tables, LDTs, die ebenfalls Deskriptoren enthalten; diese sind aber task-abhängig und können daher, »lokal« im aktuellen task, variieren. Jeweils aktiv aber kann immer nur eine, die »task-eigene« LDT des aktuellen tasks, sein. Auf beide Tabellen kommen wir im nächsten Abschnitt noch ausführlich zurück.

Es macht wenig Sinn, ein Speicherabbild dieser beiden Tabellen abzudrucken. Denn wie bereits mehrfach geäußert, setzen sich die Tabellen aus Deskriptoren zusammen, die jeweils acht Byte Umfang haben. Ich glaube, das können Sie sich auch ohne Abbildung vorstellen.

GDT segment descriptor

Es gibt nur eine, globale GDT, die auch noch hinsichtlich ihrer Eigenschaften recht genau festgelegt ist, also auch ohne Segment-Deskriptor recht genau beschrieben werden kann. Für ihre Nutzung sind lediglich ihre Basisadresse und ihre Größe notwendig – dazu ist kein Deskriptor erforderlich, zumal die GDT auch nicht zugriffsgeschützt sein oder ausgelagert werden darf und damit Attribute entfallen. Die wenigen erforderlichen Daten können (und müssen!) auch anders gehalten werden. Einen GDT segment descriptor gibt es somit nicht!

LDT segment descriptor

Demgegenüber kann es jedoch praktisch beliebig viele LDTs geben, die auch hinsichtlich ihrer Eigenschaften (Größe, Zugriffsbeschränkungen etc.) sehr unterschiedlich sein können. Daher gibt es im Gegensatz zur GDT für LDTs Segment-Deskriptoren, die LDT-Segment-Deskriptoren (»local descriptor table segment descriptors«). Diese unterscheiden sich von anderen Deskriptoren nur durch wenige Einzelheiten, wie Abbildung 2.7 zeigt.



Abbildung 2.7: Speicherabbild eines LDT-Segment-Deskriptors

Wie Sie sehen können, finden Sie alle »globalen« Felder wieder, die für Schutzkonzepte und Paging notwendig sind (G, P, DPL). Das *type field* enthält den Wert 0010_b, der Code für »LDTS descriptor«.

Task State Segments

Ihnen wird auch der Begriff »task« bekannt sein, der beim »Multitasking« eine Rolle spielt. So müssen über die einzelnen Tasks sehr viele Informationen verfügbar sein, die beim *task switch*, also dem Wechsel zwischen einzelnen Tasks benötigt werden, wir werden im Kapitel »Multitasking« auf Seite 462 noch darauf zurückkommen. Diese Informationen werden in einem Segment verwaltet, dem *task state segment* (TSS). Ein solches Segment ist sehr spannend, da es anders als Code- und Datensegmente sehr strukturiert ist und immer die gleichen Informationen enthält. Abbildung 2.8 auf Seite 420 zeigt den Aufbau eines TSS im 32-Bit-Format.

Es bedeuten:

PTL — previous task link; dieses Feld enthält den Selektor auf ein TSS des vorangegangenen Tasks. Dieses Feld wird nur dann benutzt und aktualisiert, wenn der aktuelle Task vom vorhergehenden via CALL, Interrupt oder Exception aufgerufen wurde. PTL erlaubt das »Zurückschalten« zum vorangehenden Task durch ein IRET.

SSx:ESPx — Für jede der Privilegstufen 0, 1 und 2 gibt es im TSS ein Feld für den Inhalt von SS und ESP des Stacks, der in der entsprechenden Privilegstufe zu benutzen ist. Der Stack

für die aktuelle Privilegstufe (also meistens Privilegstufe 3) wird in den Feldern SS und ESP gehalten.

CR3 — Das Feld CR3 enthält den Inhalt des control registers #3. Dies ist deshalb gegenüber den anderen control registers privilegiert, da es die Basisadresse des page directories hält und daher auch page directory base register (PDBR) heißt.

LDT SS — LDT segment selector; jeder task kann seine eigene local descriptor table besitzen (und tut das in der Regel auch!). Deren Selektor wird in diesem Feld gehalten.

T — debug trap flag; ist dieses Flag (Bit 0 an Offset $64) gesetzt, so wird jedes Mal eine debug exception ausgelöst, wenn ein switch zu diesem task erfolgt.

I/O Map — I/O map base address; dieses Feld hält den Offset der »I/O permission bit map« im TSS, der in diesem Feld stehende Wert ist also die Byte-Nummer, an der im TSS die bit map beginnt. Indirekt gibt dieses Feld auch die Lage der 32 Byte umfassenden »software interrupt redirection bit map« an, die diese immer 32 Bytes unterhalb der Adresse der I/O permission bit map beginnt. Bei der I/O permission bit map handelt es sich um ein in Lage und Ausdehnung flexibles Feld, das für jede mögliche I/O-Adresse ein (Byte-Ports), zwei (Word-Ports) oder vier Bits (Double-Word-Ports) zur Verfügung stellt und durch eine Folge von acht gesetzten Bits abgeschlossen wird. Der Zugriff auf die I/O-Adresse ist immer dann gestattet, wenn das/alle entsprechende(n) Bit(s) gelöscht ist (sind). Sind sie gesetzt oder ist in der bit map kein Bit für eine bestimmte I/O-Adresse reserviert, so wird der Zugriff verweigert. Die software interrupt redirection bit map stellt für jeden der 256 möglichen Interrupts ein Bit zur Verfügung (256 Bits = 32 Bytes!), das darüber Auskunft gibt, ob bei einem im virtual 8086 mode laufenden Real-Mode-Programm die Interrupt- und Exception-Handler dieses Programms verwendet werden sollen oder diejenigen aus dem umgebenden protected mode.

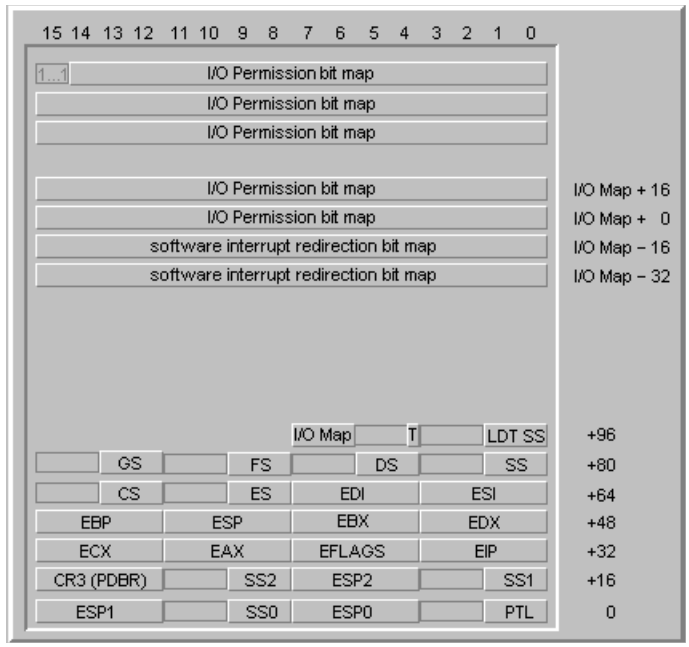


Abbildung 2.8: Speicherabbild des Task State Segments

Neben diesen Feldern gibt es noch Felder für die Allzweckregister EAX, EBX, ECX, EDX, ESP, EBP, ESI und EDI, das EFlags-Register sowie die Segmentregister CS, DS, ES, FS, GS und SS.

Man unterscheidet die im TSS realisierten Felder in die dynamischen Felder, die immer dann aktualisiert werden, wenn ein task switch erfolgt, und in die statischen Felder, die nur bei der Erstellung eines tasks initialisiert werden. Zu den dynamischen Feldern gehören die Felder für die Allzweckregister EAX, EBX, ECX, EDX, ESP, EBP, ESI und EDI, die Segmentregister CS, DS, ES, FS, GS und SS sowie das EFlags-Register, deren Inhalte vor einem task switch gesichert werden. Auch das PTL und das EIP-Register gehören zu den dynamischen Feldern.

Die statischen Felder bestehen aus den restlichen Feldern: dem LDT SS, CR3, dem T-Flag, dem Feld für die I/O base address map und die Stack-Pointer in den drei Privilegstufen 0 bis 2.

Mit den in diesem Segment stehenden Angaben lässt sich ein task vollständig beschreiben und verwalten.

TSS Descriptor Analog der LDT besitzt auch das TSS einen eigenen Deskriptor, den TSS Deskriptor. Er ist in Abbildung 2.9 dargestellt.

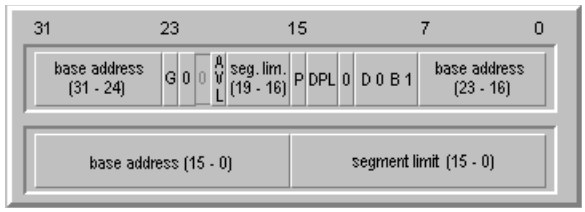


Abbildung 2.9: Speicherabbild eines TSS-Deskriptors

Der TSS-Deskriptor bietet gegenüber dem LDT-Segment-Deskriptor lediglich eine Besonderheit: Sein type field enthält zwei Flags, das *size flag* D und das *busy flag* B. B ist immer dann gesetzt, wenn der dazugehörige Task *busy* ist, also läuft, nicht aber, wenn er unter- (»suspended«) oder abgebrochen (»terminated«) wurde. D gibt an, ob mit 16-Bit- (D = 0) oder mit 32-Bit-Task-State-Segmenten (D = 1) gearbeitet wird. Somit sind für *task state segments* vier *type field codes* reserviert: 1001_b für inaktive und 1011_b für aktive Tasks mit 32-Bit-TSS und 0001_b für inaktive und 0011_b für aktive Tasks mit 16-Bit-TSS.

Gates und ihre Deskriptoren

Bislang haben wir Segmente kennen gelernt, die bestimmte Daten (im weitesten Sinne) aufnehmen können: Code-, Daten- und Systemsegmente. Die dazugehörigen Deskriptoren beschreiben die Segmente ganz allgemein, also an welcher Stelle im Speicher sie liegen, wie groß sie sind und welche Eigenschaften sie haben bzw. welche Privilegien erforderlich sind, um auf sie zuzugreifen.

Betrachten wir aber einmal das Codesegment etwas genauer. Hier sind alle Routinen untergebracht, die im aktuellen Prozess benötigt werden. Neben dem eigentlichen Code des Programms handelt es sich also noch um Routinen aus zugeladenen DLLs, um Betriebssystem-Routinen, die Programme nutzen können, u. v. m. Das bedeutet, dass Codesegmente mit einem Codesegment-Deskriptor nur grundsätzlich beschrieben werden können – zu der Adresse der nutzbaren Routinen sagt es nichts aus. Codesegmente müssen daher verschiedene »Tore« haben, über die man ins Codesegment einfallen kann. Das Wichtigste dieser Tore ist die Startadresse des auszuführenden Programms, denn das Programm muss ja gestartet werden. Andere wichtige Tore sind die veröffentlichten Adressen von Programm-, DLL- und Betriebssystem-Routinen. Und nicht ganz unwesentlich sind auch Tore zu anderen Tasks.

Solche Tore in das Codesegment gibt es wirklich. Und sie heißen auch noch so: »*gates*«. Gates sind also nichts anderes als definierte Eintrittspforten, durch die man bestimmte Codeteile im Codesegment anspringen kann. Und weil im protected mode Schutzkonzepte eine wesentli-

che Rolle spielen, wundert es wohl kaum, wenn auch der Zugang über solche gates einer strengen Prüfung unterliegt. Also werden wieder Deskriptoren erforderlich, die ein solches gate beschreiben: Gate-Deskriptoren. Sie haben gemäß ihrer Funktion einen etwas anderen Aufbau als »normale« Deskriptoren, wie Abbildung 2.10 zeigt.



Abbildung 2.10: Speicherabbild eines Gate-Deskriptors

Die Abbildung zeigt eigentlich zunächst nicht viel! Auch bei Gate-Deskriptoren ist Bit 12 des zweiten DoubleWords gelöscht, was bedeutet, dass der Deskriptor ein Systemsegment beschreibt. Das mag hier zwar etwas hochtrabend klingen. Denn schließlich ist ein Gate ja kein »echtes« Segment im Sinne eines via Deskriptor genau beschreibbaren Bereiches im Speicher, an dem Daten (im weitesten Sinne) zu finden sind, sondern lediglich ein spezifizierter Punkt in einem Codesegment, das durch einen Codesegment-Deskriptor bereits definiert wird. Doch vereinfacht diese Sichtweise das weitere Verständnis ungemein, wenn man Gates auch als »Segmente« auffasst und entsprechend behandelt. So tun es auch Intel und Microsoft, weshalb wir uns dem hier nicht entziehen wollen und Gates als »Systemsegmente« akzeptieren.

Das *present flag* P ist ebenfalls vorhanden, genauso wie der *descriptor privileg level* DPL. Das Feld *type* wird zum genaueren Typisieren verwendet, um was für ein Gate es sich handelt – es gibt nämlich mehrere Gate-Typen. Und davon abhängig sind die Einträge in den verbleibenden, hier frei gelassenen Feldern.

Es gibt vier Gates: das *call gate*, das *interrupt gate*, das *task gate* und das *trap gate*.

call gates

Teil der Schutzkonzepte moderner Betriebssysteme ist, dass man nicht mehr einfach und wahllos Programme oder -teile aufrufen kann, die nicht zum eigenen Programm gehören. So wird z.B. grundsätzlich verboten, dass ein Anwendungsprogramm mittels CALL einfach Systemmodule aufrufen kann. Das bedeutet, dass der Aufruf von CALL mit einer Adresse, die außerhalb des eigenen Segmentes liegt, zu der belieb-

ten Exception »Allgemeine Schutzverletzung« führt – es sei denn, man hat entsprechende Zugriffsrechte, was wohl eher selten der Fall sein dürfte. Andernfalls machten die Schutzkonzepte keinen Sinn!

Andererseits kann es dennoch sehr sinnvoll sein, eine Möglichkeit für Außenstehende zu schaffen, Module trotzdem anspringen zu können, z.B. wenn ein Anwendungsprogramm eine Routine in einer (geschützten) Systembibliothek nutzen möchte. Windows als Betriebssystem ist ja eine Sammlung von Bibliotheksroutinen, die dem Anwendungsprogrammierer zur Verfügung gestellt werden. Dieser »Zugang« zu geschützten Ressourcen muss aber sehr kontrolliert erfolgen, um zu verhindern, dass über eine solche Möglichkeit das gesamte Schutzkonzept ad absurdum geführt wird. Daher hat man »Tore« definiert, über die ein Programm auch dann in geschützte Module gelangen kann, wenn es nicht über die erforderlichen Zugriffsrechte verfügt. Diese *call gates* beziehen sich also wie alle Gates immer auf ein bestehendes Segment, weshalb sie in den Bits 16 bis 31 des ersten DoubleWords auch einen Segment-Selektor beherbergen (vgl. Abbildung 2.11). Darüber hinaus besitzen sie in den Bits 0 bis 15 des ersten und 16 bis 31 des zweiten DoubleWords eine 32-Bit-Einsprungadresse in dieses Segment.

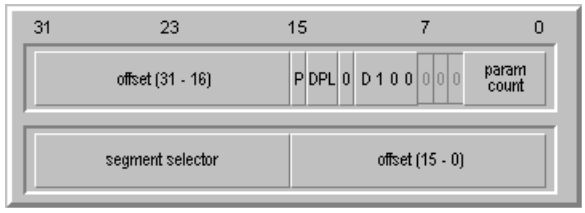


Abbildung 2.11: Speicherabbild eines Call-Gate-Deskriptors

Auf diese Weise ist gewährleistet, dass für Nichtprivilegierte ein Zugriff auf das gewünschte Segment nur an dieser definierten Stelle möglich ist und nirgendwo anders (es sei denn, es gibt weitere call gates mit anderen Einsprungadressen!).



Dem Assembler ist vollkommen egal, ob das rufende Programm die Zugriffsrechte zum Ansprung des ausgewählten Segmentes besitzt oder nicht, da er einen erlaubten Zugriff naturgemäß nicht feststellen kann. Das bedeutet, dass er für den CALL-Befehl grundsätzlich eine Adresse benötigt, die entweder nur aus einem Offset besteht, wenn innerhalb des Segmentes aufgerufen wird, oder die aus einer qualifizierten Adresse aus Segment und Offset besteht, wenn das Segment verlas-

sen werden soll. Denn schließlich kann ja, so man die entsprechenden Privilegien hat, auf diese Weise zwischen Segmenten herumgesprungen werden. Daher müssen Sie dem CALL-Befehl auch eine qualifizierte Adresse bestehend aus Segment und Offset angeben, wenn Sie ein Gate anspringen wollen, auch wenn die einzig mögliche Einsprungadresse im Gate-Deskriptor festgemauert ist. Denn woran sollte der Assembler erkennen können, dass Sie ein Gate anspringen? Wundern Sie sich daher nicht, wenn Sie dem CALL-Befehl den Segment-Selektor des Gates und einen beliebigen Offset angeben müssen. Erst die Prüfmechanismen im Rahmen des CALL-Befehls stellen dann fest, dass ein Gate angesprungen werden soll, und benutzen daraufhin die im Gate-Deskriptor angegebene Adresse aus Segment-Selektor und Offset.

Das *gate size flag* D im *type field* ermöglicht wiederum die Unterscheidung, ob ein 16-Bit-Call-Gate vorliegt (D = 0) oder ein 32-Bit-Gate (D = 1). Das type field kann also zwei Werte annehmen: 0100_b für call gates in 16-Bit-Modulen und 1100_b für solche im 32-Bit-Modulen.

Die fünf Bits 0 bis 4 im zweiten DoubleWord des Deskriptors (»param count«) geben an, wie viele Bytes bei einem eventuell erforderlichen *stack switch* zwischen den Stacks kopiert werden sollen, damit auf dem Stack liegende Parameter auch in der neuen Umgebung zur Verfügung stehen.



Auch Gates haben ein DPL-Feld und können daher Zugriffsbeschränkungen auferlegen! Allerdings kann dieses DPL eine niedrigere Privilegstufe angeben, als es das Segment im DPL-Feld seines Deskriptors verlangt. So könnte z.B. bei Betriebssystemmodulen im Deskriptor vermerkt sein, dass nur mit der höchsten Privilegstufe 0 auf dieses Segment zugegriffen werden kann. Kein Programm aus den weniger privilegierten Stufen 1 bis 3 kann es dann nutzen. Doch könnte im DPL des Gates eine Stufe 2 angegeben werden. Dann kann zwar ein Programm auf der Stufe 3 immer noch nicht auf das Modul zugreifen, wohl aber Programme auf den Stufen 1 und 2.

interrupt gates

Die Nutzung eines Interrupts ist letztlich eine etwas andere Art, ein Unterprogramm aufzurufen. Es ist daher nicht verwunderlich, dass es neben den *call gates* auch *interrupt gates* gibt, die einen gezielten Einsprung in Segmente zulassen, in denen die erforderlichen Interrupthandler implementiert sind. Diese interrupt gates unterscheiden sich, wie Abbil-

dung 2.12 zeigt, von call gates nur in zwei Punkten: einem anderen Code im *type field* und dem Fehlen des Feldes *param count*.

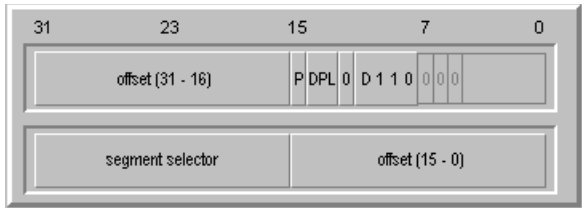


Abbildung 2.12: Speicherabbild eines Interrupt-Gate-Deskriptors

Auch interrupt gates haben das gate size flag D, mit dem das Gate als 16-bittig (D = 0) oder 32-bittig (D = 1) definiert werden kann. Das Feld type kann daher zwei Werte annehmen: 0110_b und 1110_b.

trap gates

Ein trap gate ist ein interrupt gate! Es wird daher nicht verwundern, wenn es den gleichen Aufbau wie dieses hat, was Abbildung 2.13 zeigt. Der einzige Unterschied ist das gesetzte Bit 0 im Feld type (= Bit 8 des DoubleWords), das bei interrupt gates gelöscht ist.

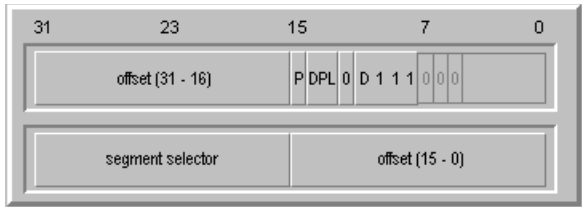


Abbildung 2.13: Speicherabbild eines Trap-Gate-Deskriptors

Wozu zwei gleiche Gates? Gibt es nicht doch einen Unterschied? Ja, es gibt ihn: Es ist die unterschiedliche Würdigung des Flags IF im Register EFlags. Wird ein Interrupt über ein interrupt gate behandelt, so löscht der Prozessor u.a. das IF. Dadurch wird verhindert, dass die Behandlung des aktuellen Interrupts durch einen weiteren Interrupt unterbrochen werden kann. Letzterer muss warten, bis der Handler unter Restauration des IF wieder verlassen wird. Bei Interuptbehandlung via trap gates ist das anders: Der Zustand von IF bleibt erhalten, was bedeutet, dass der Handler von einem weiteren Interrupt unterbrochen werden kann. Ansonsten gibt es keinen weiteren Unterschied.



Trap gates machen also immer dann Sinn, wenn die Behandlung des Interrupts nicht kritisch ist und die Notwendigkeit besteht, parallel auftretende Interrupts dennoch zu berücksichtigen. Trap gates werden daher z.B. bei Debuggern eingesetzt, bei denen nach jedem Befehl ein Interrupt ausgelöst wird, um die neue Situation darzustellen. Diese Darstellung ist nicht kritisch, wohl aber wäre es kritisch, wenn »wichtige« Interrupts aufgrund des gelöschten IF hier nicht behandelt werden könnten.

task gates

Um einem task auch die Möglichkeit zu geben, unter bestimmten Voraussetzungen einen anderen task aufrufen zu können, gibt es analog zu den call, interrupt und trap gates auch task gates. Versteht sich von selbst, dass diese gates auch einen Deskriptor benötigen. Ihn sehen Sie in Abbildung 2.14.

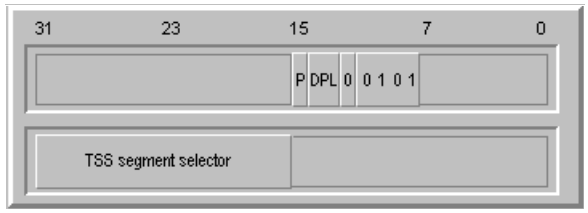


Abbildung 2.14: Speicherabbild eines Task-Gate-Deskriptors

Ein task gate braucht eigentlich nicht viel an Information, weshalb die Abbildung auch sehr »magere« Deskriptoreninhalte zeigt. Denn die wesentliche Information über einen Task ist in seinem task state segment (TSS) verzeichnet. Daher muss ein task gate lediglich einen Selektor auf das entsprechende task state segment beinhalten. Dass es außerdem noch ein DPL-Feld enthält, ist sinnvoll, um analog zu call gates die Möglichkeit zu eröffnen, Zugriffsrechte zu erweitern, nicht aber ganz abzuschaffen. Das present flag ist eigentlich überflüssig und damit praktisch immer auf »1« gesetzt! Denn der TSS segment selector zeigt ja auf einen TSS descriptor, der ebenfalls ein present flag enthält. Und nur das letztere ist in der Lage, anzugeben, ob das task state segment verfügbar ist oder nicht.



Setzt man das P-Flag des task gates dennoch auf »0«, so führt der Zugriff auf das task gate zu einer Exception, in deren Handler es dann auf »1« gesetzt werden kann, um den eigentlichen Zugriff zu gestatten. Auf diese Weise ist es möglich, die Anzahl der Zugriffe auf einen Task über dieses Gate zu protokollieren.

2.2.5 Deskriptorentabellen

An dieser Stelle eine kleine Pause zur Rekapitulation dessen, was wir bislang über das Prinzip der Speichersegmentierung erfahren haben. Zunächst: Je nach den betrachteten Daten werden verschiedene Typen von Segmenten unterschieden: Daten- und Stack-, Code- und Systemsegmente. Diese Segmente sind Speicherbereiche, die eine Basisadresse, eine definierte Größe und Eigenschaften haben. All diese Informationen über die Segmente werden in speziellen Datenstrukturen, den Segment-Deskriptoren, gehalten. So weit, so gut. Doch wo findet man diese Deskriptoren?

Antwort: In dafür bestimmten Tabellen. Der Prozessor kennt drei Arten von solchen Tabellen:

- eine globale Tabelle für Deskriptoren (*global descriptor table*, GDT),
- eine lokale Tabelle für Deskriptoren (*local descriptor table*, LDT) und
- eine globale Tabelle für Gate- oder TSS-Deskriptoren, die bei der Behandlung von Interrupts eine Rolle spielen (*interrupt descriptor table*, IDT).

Global Descriptor Table

Die GDT ist, wie der Name schon sagt, global verfügbar! Das bedeutet, dass die Einträge (= Deskriptoren) in dieser Tabelle zu jedem beliebigen Zeitpunkt verfügbar sein müssen und sind. Somit ist sie (neben der IDT) die wichtigste Tabelle – ohne sie läuft im wahrsten Sinne des Wortes nichts! Sie ist, wie bereits gesagt, nicht in einem eigenen Segment untergebracht, sondern als »einfache« Struktur, die an 8-Byte-Grenzen ausgerichtet sein sollte, um beste Performance zu gewährleisten.

In der GDT können fast alle Arten von Deskriptoren stehen:

- Deskriptoren auf lokale Deskriptorentabellen (LDTs)
- Deskriptoren of Code-, Daten- und Stacksegmente
- Deskriptoren auf Task State Segments
- Deskriptoren auf Call- und Task Gates

Der erste Eintrag in der GDT wird nicht benutzt – es handelt sich um einen Null-Eintrag, der zu Prüfzwecken reserviert ist. Die Adresse der GDT wird in einem speziellen CPU-Register, dem GDTR (global descriptor table register), abgelegt. Wir werden es weiter unten kennen lernen. Auf diese Weise hat man jederzeit Zugriff auf diese lebensnotwendige Tabelle.

Local Descriptor Table

Anders als die GDT ist die LDT nicht notwendigerweise global verfügbar. So handelt es sich bei der LDT im Prinzip um eine für jeden Task »private« Tabelle für Deskriptoren.

Im Prinzip kann auch die LDT die meisten Arten von Deskriptoren enthalten, wenn dies sinnvoll ist. So macht es sicherlich in den meisten Fällen wenig Sinn, wenn ein LDT-Eintrag auf eine weitere LDT zeigt. Daher finden Sie in der Regel in LDTs keine LDT-Deskriptoren. Auch Deskriptoren von task state segments oder task gates sind hier nicht zu finden, da diese bei task switches benötigt werden. Ein task switch ist aber nicht eine Sache, die lokal behandelt werden kann – es ist etwas tief Greifendes, Globales. Daher findet man solche Deskriptoren nur in der GDT.

Ein weiterer Unterschied zur GDT liegt darin, dass es nur eine GDT gibt, aber beliebig viele LDTs existieren können. Daher reicht es nicht, wenn analog dem GDTR auch ein LDTR existiert, in dem die Adresse der LDT liegt. Denn die Adresse welcher LDT ist hier enthalten? Antwort: die der jeweils aktuellen, sprich durch den Task benutzten. Daher benötigt man noch einen Ort, an dem eine Liste aller verfügbaren LDTs verwaltet werden kann.

Dieser Ort ist die GDT. Das bedeutet, jede LDT hat einen Deskriptor in der GDT, der über einen Selektor angesprochen werden kann. Im Gegensatz zur GDT ist eine LDT also ein echtes Segment, sogar ein Systemsegment.

Interrupt Descriptor Table

Die letzte der Deskriptorentabellen ist die IDT (interrupt descriptor table). Sie ist wie die GDT global verfügbar (muss sie auch sein!) und ähnelt auch ansonsten der GDT. So besitzt sie mit dem IDTR ein eigenes Register für ihre Basisadresse. Und wie die GDT ist sie zwar eine Struktur, jedoch kein Segment.

In IDTs machen nur Einträge Sinn, die im Rahmen des Interrupt-Mechanismus benötigt werden, also

- Deskriptoren auf ein task state segment, wenn der dadurch beschriebene Task Interrupts oder Exceptions behandeln kann, oder
- Deskriptoren auf interrupt, trap oder task gates. In diesem Fall muss das Gate auf einen Handler zeigen, der den Interrupt bzw. die Exception bedienen kann.

2.2.6 Selektoren

Wir stellen also fest, dass wir »das Pferd von hinten aufgezäumt« haben. Wir sind von der Definition von Segmenten ausgegangen, haben unterschiedliche Segmente und ihre Funktion kennen gelernt und auch erfahren, dass es drei Tabellen gibt, in denen die Segmente verwaltet werden und in denen Informationen zu den einzelnen Segmenten stehen: die Segment-Deskriptoren-Tabellen.

An dieser Stelle fehlt uns eigentlich nur noch eine Information: Wie komme ich an ein bestimmtes Segment heran? Und spätestens hier müsste uns klar sein, dass man eine Adresse braucht, wenn man jemanden besuchen möchte. Und diese Adresse heißt in unserem Fall: Selektor.

Man benötigt also einen Selektor, wenn man mit einem Segment arbeiten will. Dieser Selektor ist im Prinzip nichts anderes als ein Zeiger in eine der Deskriptoren-Tabellen, die wir bereits kennen gelernt haben. Haben wir diesen Zeiger, haben wir alle Informationen, die wir benötigen. So können wir durch Auslesen des Segment-Deskriptors in einer Deskriptoren-Tabelle (GDT oder LDT – IDT macht keinen Sinn!) an der Stelle, auf die der Zeiger zeigt, die Basisadresse, die Größe und den Typen des Segmentes feststellen, das wir ansprechen möchten. Und wir können anhand der Attribute, die hier verzeichnet sind, noch viele andere Informationen erhalten.

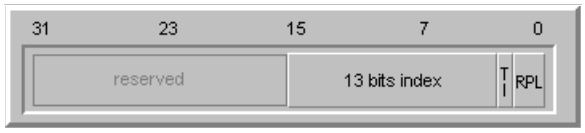


Abbildung 2.15: Speicherabbild eines Segment-Selektors

Segment-Selektor

Ein Segment-Selektor (kurz: *Selektor*) besteht daher aus einem 13-Bit-Index, der auf einen der möglichen 2^{13} = 8.192 Einträge in eine der Systemtabellen zeigt: in die GDT, global descriptor table, oder die LDT, local descriptor table. Um diesen Tabelleneintrag, einen Deskriptor, auslesen zu können, muss der Offset, an dem er steht, berechnet werden. Da Deskriptoren jeweils zwei DoubleWords (= vier Bytes) umfassen, wird der Index mit 8 multipliziert. Der so erhaltene Offset kann nun zur Basisadresse der Tabelle addiert werden, die er entweder aus dem GDTR oder aus dem LDTR ausliest. Welche Tabelle betroffen ist, sagt TI, der *table indicator*: Ist TI = 0, zeigt der Index in die GDT und das

GDTR wird ausgelesen, andernfalls zeigt er in die aktuelle LDT und das LDTR wird verwendet.

Bit Bits 0 und 1 stellen den *»requestor privileg level«*, RPL, dar, der im Rahmen der Schutzkonzepte eine Rolle spielt.



Bitte beachten Sie auch den Abschnitt »Selektoren« in »›Unschärfen‹ und Ungenauigkeiten in diesem Buch« auf Seite 776.

Nullselektor Als »Nullselektor«, exakter eigentlich Null-Segment-Selektor (»null segment selector«), bezeichnet man einen Selektor, der auf das *Nullsegment* zeigt, wie der Name schon sagt. Das Nullsegment ist das Segment, das der erste Deskriptor in der global descriptor table (GDT) beschreibt. Daher muss bei einem Nullsektor auch Bit 2, das als TI-Flag ansonsten für die Unterscheidung GDT/LDT zuständig ist, »0« sein (siehe Abbildung 2.16).

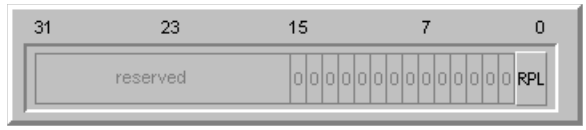


Abbildung 2.16: Speicherabbild eines »Nullselektors«

Der erste Eintrag der global descriptor table wird vom Prozessor nicht benutzt! Das bedeutet, dass dieser Eintrag keinen Deskriptoren enthält, der ein Segment beschreibt. Falls ein Zugriff auf das durch den Null-Deskriptor bezeichnete, nicht vorhandene »Null-Segment« erfolgen soll, löst die CPU eine general protection exception #GP aus.



Nicht-initialisierte Segmentregister enthalten automatisch den Null-Selektor. Daher führt jeder Zugriff auf ein nicht-initialisiertes Segmentregister zu dieser #GP. Dies ist auch der eigentliche Sinn des Null-Selektors.



Der erste Eintrag der aktuellen local descriptor table kann sehr wohl benutzt werden. Bei LDTs gibt es somit keine Null-Selektoren, der Begriff ist für die GDT reserviert.

ErrorCode

Im Rahmen der Überprüfung von Privilegien können auch Exceptions ausgelöst werden, sobald ein Zugriff unerlaubterweise erfolgt oder sonst etwas mit dem Segment »nicht stimmt«. Einige Exception-Handler erwarten dann auf dem Stack einen Fehlercode, der Einzelheiten zur Ursache der Exception angibt. Dieser »ErrorCode« ist praktisch ein Selektor auf den die Exception verursachenden Segment-Deskriptor, wie Abbildung 2.17 zeigt. Der Unterschied besteht lediglich in den Bits 0 und 1, die bei einem Selektor den *requestor privileg level* angeben, der bei einem Fehlercode dagegen keinen Sinn macht. Hier werden die Bits dazu verwendet, anzugeben, ob der Fehler durch »externe« Ursachen ausgelöst wurde (extern heißt hier: außerhalb des aktuellen Kontextes. Das kann also hardwarebedingt sein oder auch Ursachen außerhalb des eigenen tasks haben, wie z.B. bei Interrupts.). Ferner wird angezeigt, ob der durch den Selektor repräsentierte Deskriptor in der interrupt descriptor table zu suchen ist.



Abbildung 2.17: Speicherabbild des ErrorCodes, der im Rahmen von Exceptions dem exception handler übergeben wird

Einzelheiten zum Fehlercode entnehmen Sie bitte dem Kapitel »Exceptions und Interrupts« auf Seite 486ff.

2.2.7 Hardwareunterstützung für Deskriptoren und Deskriptortabellen

System-Register

Die CPU verfügt über vier Systemregister, auf denen die Speicherverwaltung aufbaut. Zwei dieser vier Register nehmen die Adressen zur Lage und Größe der global verfügbaren Deskriptoren-Tabellen GDT und IDT auf, ein weiteres die der Task-abhängigen, lokalen Deskriptoren-Tabelle LDT. Das letzte Register schließlich ist für das Halten der Informationen über den aktuellen Task zuständig. Abbildung 2.18 zeigt den Aufbau dieser Register.

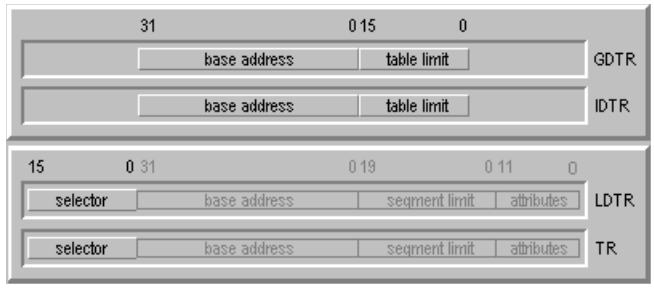


Abbildung 2.18: Speicherabbild der Systemregister des Prozessors

Das wichtigste und im wahrsten Sinne des Wortes existentiellste Register ist das *global descriptor table register* (GDTR). Ohne dieses Register läuft gar nichts! Es ist ein 48 Bit breites Register, in das die lineare 32-Bit-Basisadresse und die 16-Bit-Größe der *global descriptor table* eingetragen wird. Diese oberste Instanz der Speicherverwaltung ist (mit Ausnahme der *interrupt descriptor table*) das einzige Segment, das mit einer konkreten Adresse und Größe angegeben werden muss – es gibt für den Prozessor keine andere Möglichkeit, es im Speicher zu lokalisieren! Und daher ist es eine der primärsten Aufgaben des Real-Mode-Betriebssystemladers nach dem Einschalten der CPU, diese Tabelle zu erzeugen und anzusiedeln, bevor in den protected mode geschaltet wird. Alle anderen Segmente und Strukturen sind in Form von Selektoren (also Zeigern in diese Tabelle) eindeutig beschreibbar.

Eine analoge, nicht weniger wichtige Funktion erfüllt die *interrupt descriptor table*, weshalb auch sie über ein eigenes, analog aufgebautes Register, das *interrupt descriptor table register* (IDTR), verfügt. Auch sie muss während des Boot-Vorgangs erzeugt werden und existieren, bevor in den protected mode umgeschaltet wird.

Neben der Globalen Deskriptortabelle (GDT) kann zu jedem Zeitpunkt auch eine (von Task zu Task unterschiedliche und somit Task-abhängige) Lokale Deskriptortabelle (LDT) existieren. Somit gibt es auch ein Register, das *local descriptor table register* (LDTR), das die jeweils aktuelle LDT aufnimmt. Da aber jede LDT mit ihrem Deskriptor in der GDT verzeichnet sein muss, ist für das LDTR ein 16-Bit-Register ausreichend, das einen Selektor (Zeiger in die GDT) aufnimmt. Aufgrund von Performance-Überlegungen verfügt das LDTR jedoch über einen 64-Bit-Cache, der eine 32-Bit-Basisadresse, eine 20-Bit-Segmentgröße und 12 Bits Segment-Attribute aufnehmen kann. Dieser Cache wird immer dann gefüllt, wenn ein neuer Selektor in das LDTR geschrieben wird: Er erhält dann die entsprechenden Informationen aus dem Segment-

Deskriptor der LDT. Auf diese Weise wird vermieden, dass jeweils die zeitaufwändige Konsultation der GDT erfolgen muss, wenn das Betriebssystem Informationen über die LDT benötigt.

Analog aufgebaut ist das *task register* (TR). Dieses Register weist auf den jeweils aktuell ausgeführten Task, genauer gesagt auf das ihm zugeordnete Task State Segment (TSS), in dem sein aktueller Status gehalten wird. Somit benötigt auch das TSS, wie jedes andere Systemsegment auch, einen Deskriptor. Dieser muss in der GDT eingetragen sein, weshalb wie im Falle der LDT ein 16-Bit-Selektor als Zeiger in die GDT ausreicht, um einen Task, hier den aktuellen, eindeutig zu identifizieren. Das TR enthält diesen Selektor. Es hat auch einen 64-Bit-Cache, in dem die Informationen aus der GDT gepuffert werden und somit jederzeit ohne Konsultation der Deskriptoren-Tabelle verfügbar sind.

Segment-Register

In Kapitel 1 wurden die Segmentregister CS, DS, ES, FS, GS und SS bereits dargestellt. Dort wurde behauptet, dass sie 16 Bit breit sind. Das ist nicht ganz korrekt, verfügen sie doch analog zu LDTR und TR über einen 64-Bit-Cache. Auch die Segmentregister enthalten Selektoren (16-Bit-Zeiger in eine der Deskriptortabellen GDT oder LDT), die auf die entsprechenden Deskriptoren zeigen. Deren Inhalt (32-Bit-Basisadresse, 20-Bit-Segmentlimit und 12 Bits Attribute gemäß Segment-Deskriptor) wird beim Beschreiben des Segmentregisters mit einem neuen Selektor im Cache gepuffert, um zeitaufwändige Konsultationen der Deskriptorentabellen überflüssig zu machen.



Abbildung 2.19: Speicherabbild eines Segmentregisters

Die Segmentregister spielen, wie wir im nächsten Kapitel sehen werden, eine wesentliche Rolle beim Zugriff auf die Segmente. Daher sind sie auch spezialisiert. So enthält das CS-Register den Selektoren für das aktuelle Codesegment, das DS- und ES-Register und, je nach Programmiersprache und gewähltem Modell ggf. auch FS- und/oder GS, je einen Selektor auf ein Datensegment. Auf diese Weise können somit bis zu vier Datensegmente gleichzeitig verwaltet werden, meistens jedoch zeigen mehrere oder alle der Datensegment-Register auf das gleiche Datensegment. Mit dem SS-Register gibt es schließlich ein Segmentregister, das für den jeweils aktuellen Stack zuständig ist.

2.2.8 Zugriffe auf den Speicher: Von Adressen und Adressräumen

Programmierer, sowohl Assembler- als auch Hochsprachenprogrammierer, arbeiten mit Symbolen. Solche Symbole sind Variablen-, Konstanten-, Routinennamen oder Labels. Sowohl Assembler als auch Hochsprachen unterstützen dies, indem sie dem Programmierer die fehlerträchtige und aufwändige Adressberechnung abnehmen, die hinter den Symbolen steht. Denn Argumente für die Prozessor-Befehle sind nicht etwa die Symbole, sondern immer Adressen, auch wenn das nicht so aussieht!

Für den Programmierer sieht damit die Welt recht einfach und wie in Abbildung 2.20 dargestellt aus. Für ihn gibt es eine black box, die die Umrechnung der Adresse »seiner Variablen« bzw. »seines Labels« in eine Adresse bewerkstelligt, die der Prozessor für den Zugriff benutzen kann.

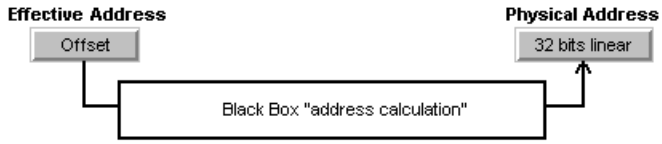


Abbildung 2.20: Beziehung zwischen effektiver und physikalischer Adresse

Diese vereinfachte Sicht reicht aber beim Programmieren unter Assembler bei weitem nicht aus. Man muss zumindest grob wissen, wie die black box arbeitet. Abbildung 2.21 zeigt, dass es sich um einen dreistufigen Prozess über die Bildung einer logischen, einer virtuellen und einer physikalischen Adresse handelt.

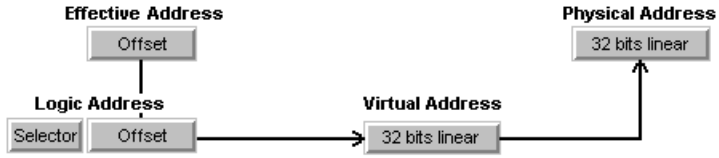


Abbildung 2.21: Der Weg von der (relativen) effektiven zur (absoluten) physikalischen Adresse

2.2.9 Beziehungskisten: Von der effektiven zur logischen Adresse

Diesen dreistufigen Prozess wollen wir im Folgenden etwas genauer untersuchen.

Effektive Adresse

Als effektive Adresse bezeichnet man eine 32-Bit-Adresse, *die relativ zum Segment zu betrachten ist*, in der sie steht. Sie ist damit ein »Offset« zur Basisadresse des Segmentes. Sowohl Hochsprachen als auch Assembler arbeiten grundsätzlich mit solchen effektiven Adressen. Das bedeutet, dass jede Adresse, die ein Compiler oder Assembler aus einem Variablen- oder Labelnamen erzeugt, relativ zum Ursprung des Segments, also des Daten- oder Codesegments zu interpretieren ist.



Effektive Adressen (EAs) sind somit *immer* »relative« Adressen! In 32-Bit-Umgebungen kann fälschlicherweise der Eindruck entstehen, dass die effektive Adresse eine »absolute« Adresse ist, da mit den zur Darstellung der EA verwendeten 32 Bit der gesamte physikalisch ansprechbare Adressraum von 32-Bit-Prozessoren erreicht werden kann.



Falls Sie häufig mit Debuggern arbeiten, wundern Sie sich daher nicht, dass mancher Debugger z.B. einen Sprung zu einem von Ihnen definierten Label namens »MyLabel« evtl. nicht wie erwartet disassembliert. Bei der Assemblierung berechnet nämlich der Assembler die Distanz zwischen der aktuellen Position und dem Sprungziel und speichert eine Befehlssequenz ab, die als Operanden diese Distanz beinhaltet: `E934120000`. Das Symbol *MyLabel* geht hierbei als Information verloren. Der Debugger disassembliert nun zwecks Anwenderfreundlichkeit diese Befehlssequenz wieder in das Mnemonic `JMP`. Da er aber keine Information darüber hat, dass die Position, die 1234h Bytes von der aktuellen Position entfernt ist, von Ihnen *MyLabel* genannt worden war, schafft er sich ein »eigenes« Label. Und dies setzt sich zusammen aus dem Segmentnamen und einem Offset: `JMP` *`MySegment`* `+ 00006789h`. Analoges passiert mit Daten: Die Information, dass die Speicheradresse `$00000012` von Ihnen *MyDate* genannt wurde, geht bei der Assemblierung verloren, weshalb der Debugger einen Zugriff auf diese Adresse als `MOV EAX,` *`MyDataSegment`* `+ 00000012h` darstellt. Übrigens: Auch die Compiler speichern solche für den Prozessor und ein kompiliertes Programm somit nicht erforderlichen Zusatzinformationen nicht notwendigerweise. Fehlen solche Informationen (manche Debugger machen

z.B. darauf aufmerksam, dass kein Quellcode gefunden werden konnte!), kann auch ein kompiliertes Programm nicht mit den selbst definierten Symbolen debuggt werden.

Diese effektive Adresse, mit der aufgrund der »anwenderfreundlichen« Tätigkeit von Compilern und Assemblern in der Regel weder der Hochsprachen- noch der Assemblerprogrammierer direkt zu tun haben wird (weil sie einfach durch die Definition von Symbolnamen wie Variablen, Konstanten, Labels, etc. »versteckt« wird), ist somit *nicht* die Adresse, über die die CPU den notwendigen Zugriff auf den Speicher vornehmen kann. Hierzu braucht sie absolute Adressen.

Zu ihrer Berechnung muss der Umrechnungsmechanismus wissen, welches Segment denn Basis ist und damit als Ursprung zu gelten hat. Bei Codebezügen ist das klar: das Codesegment. Bei Daten jedoch ist das nicht mehr so ganz klar: Standardmäßig ist es das Datensegment – aber es können ja auch Daten im Codesegment stehen, wie es z.B. in ROMs häufig der Fall ist (denken Sie an Windows CE, wo das gesamte Betriebssystem samt Daten im ROM stehen kann). Und es gibt ja auch die Unterscheidung zwischen »initialisierten« Datensegmenten, in denen manche Sprachen ihre Konstanten unterbringen, und »nicht initialisierten« Datensegmenten für die Variablen.

Qualifizierte Adresse

Kurz: Eine vollständige, »qualifizierte« Adresse besteht immer aus der Angabe des Segmentes und einem Offset in dieses Segment. Und mit diesen Angaben muss nun die Umrechnung zur physikalischen Adresse erfolgen, wie die Adresse genannt wird, über die die CPU den Speicherzugriff erreichen kann.

Logische Adresse

Da die Angabe eines Bezugsegmentes erforderlich ist, wenn man eine effektive Adresse nutzen will, hat sich aus der Beziehung Basisadresse des Segments und Offset ein Begriff entwickelt: die »logische Adresse«. Sie ist die eigentliche, »qualifizierte«, absolute Adresse, die man zur Berechnung der physikalischen Adresse benötigt, und ist eigentlich »nur« ein Formalismus:

Logic Address = Segment : Effective Address

Der Doppelpunkt zwischen der Segmentangabe und der effektiven Adresse besagt hierbei, dass noch Rechenarbeit erforderlich ist, um eine physikalische Adresse tatsächlich zu berechnen.



Die logische Adresse ist demnach eine »absolute« Adresse, auch wenn sie auf den ersten Blick nicht so aussieht. Denn in dieser Adresse ist die Angabe über die Adresse des Segments verzeichnet, wenn auch »verklausuliert« in Form eines Selektors in eine Deskriptoren-Tabelle. Die logische Adresse beinhaltet somit alle Informationen, die zur Berechnung einer »linearen«, also »nicht-segmentierten« Adresse erforderlich sind.



Bitte behalten Sie diese Unterschiede immer im Hinterkopf! Es gibt Assemblerbefehle, die mit effektiven Adressen arbeiten, also nur mit Offsets innerhalb eines Segments. Dies sind z.B. alle Sprungbefehle, die mit Adressen innerhalb des aktuellen Segmentes arbeiten (absolute »Intrasegment-CALLs«, »near calls«). Denn hierbei werden die Sprungziele immer auf das aktuelle Codesegment (»near«) bezogen. Argument für die Sprungbefehle ist somit jeweils »nur« eine effektive Adresse. Auch bei Speicherzugriffen auf Adressen, die sich innerhalb des aktuellen Datensegments (selektiert durch den Inhalt von DS) befinden, arbeiten mit effektiven Adressen als Operand.

Andere Befehle, wie die Sprünge und Calls in andere Segmente (»Intersegment-Sprünge/Calls«, »far jumps/calls«) und jeder Zugriff auf den Speicher mittels MOV u.Ä. auf Adressen, die sich außerhalb des aktuellen Datensegments befinden, benötigen absolute, logische Adressen. Hierbei wird immer auch ein Bezug auf das zu verwendende Segment übergeben. Bei Jumps/Calls ist das ein Selektor auf ein in einer der Deskriptoren-Tabellen verzeichnetes, anderes Codesegment (»far«), der zusammen mit der effektiven Adresse als logische Adresse übergeben wird. Bei Speicherzugriffsbefehlen ist es ein »segment override prefix«, mit dem ein anderes Segmentregister als DS als Daten-Bezugsregister gewählt wird. In diesem Segmentregister muss dann der Selektor auf einen Deskriptor stehen, der das gewünschte Datensegment beschreibt. Formal wird somit bei Speicherzugriffsbefehlen zwar immer »nur« eine effektive Adresse als Operand übergeben, de facto aber ist es aufgrund des (impliziten oder expliziten) Verweises auf das zu verwendende Segmentregister eine logische Adresse.

Auch Sprünge/Calls via Gates sind Intersegment-Sprünge, da sie ebenfalls mit logischen Adressen (Selektor auf einen und Einsprungadresse im angegebenen Gate-Deskriptor) arbeiten.



Neben den Sprüngen/Calls mit effektiven oder logischen Adressen gibt es noch solche mit Sprungdistanzen als Argument (relative Intrasegment-Sprünge). Bei diesen relativen Sprüngen ist somit Bezugspunkt die aktuelle Position im Codesegment. Technisch gesehen sind sie somit auch nichts anderes als die *near jumps/calls*, deren Bezugspunkt die Segmentbasis ist. (Übrigens: Jumps – und nur diese! –, deren Distanz zwischen -128 und +127 Bytes liegt, nennt man »short jumps«).

2.2.10 Speichersegmentierung: Von der logischen zur virtuellen Adresse

Wie dem auch sei, als Ergebnis der Angabe einer relativen effektiven Adresse bzw. der damit verknüpften segmentierten absoluten Adresse (mit impliziter oder expliziter Segmentangabe) muss eine lineare absolute Adresse berechnet werden. Während der Zusammenhang zwischen der effektiven und der logischen Adresse unabhängig vom benutzten Betriebsmodus des Prozessors ist – in allen Modi wird mit Segmenten gearbeitet –, steht zu erwarten, dass die Berechnung der linearen, absoluten Adresse abhängig vom Modus ist: Dies haben wir in den vorangehenden Abschnitten bereits festgestellt.

Protected Mode

Und so ist es auch. Im protected mode benutzt man die logische Adresse, um eine *virtuelle Adresse* zu berechnen. Diese virtuelle Adresse ist eine lineare, 32 Bit breite Adresse, die nun unabhängig von irgendwelchen Segmenten ist. Man könnte sagen, sie sei »defragmentiert«. Abbildung 2.22 zeigt, wie die Berechnung abläuft.

Virtuelle Adresse

Im Selektor, der zur Adressberechnung herangezogen wird, signalisiert ein Flag, ob die global descriptor table (GDT) oder die aktuelle local descriptor table (LDT) den Deskriptor beinhaltet, der das Bezugssegment beschreibt. Dieser Selektor ist entweder im spezifizierten Segmentregister verzeichnet oder wird als Teil des Argumentes des Befehls übergeben. Mit diesem Selektor und der Basisadresse der zu verwendenden Deskriptoren-Tabelle (GDT oder LDT), die im GDTR oder LDTR verzeichnet ist, kann der Prozessor die Adresse berechnen, an der der gewünschte Deskriptor im Speicher steht. Diesem Deskriptoren entnimmt er die lineare 32-Bit-Basisadresse des Segments. Zu ihr addiert er den dem Befehl als Argument übergebenen Offset, die effektive Adresse. Das Ergebnis ist die »virtuelle Adresse«.

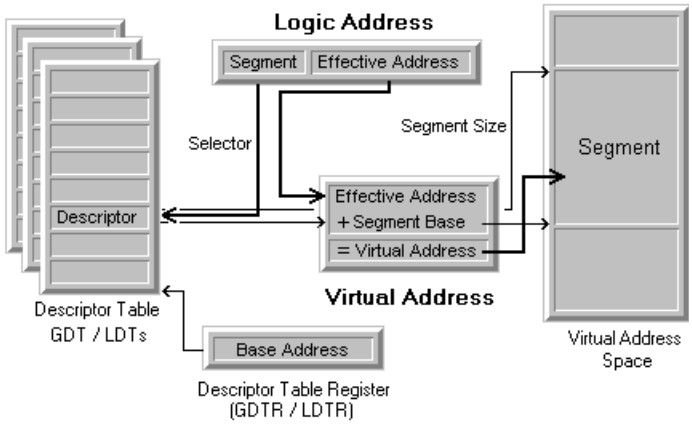


Abbildung 2.22: Berechnung einer virtuellen Adresse aus einer logischen



Der in Abbildung 2.22 dargestellte Aufwand muss nicht wirklich getrieben werden! Wäre dies der Fall, so hätte es katastrophale Auswirkungen auf die Performance: Stellen Sie sich vor, bei jedem Zugriff auf das Datensegment müsste die Deskriptoren-Tabelle konsultiert und die erforderlichen Informationen ausgelesen werden. Aber das ist auch nicht nötig! Wie Sie bereits wissen, wird bei jedem Laden eines Segmentregisters mit einem Selektor der dazugehörige Deskriptor ausgelesen und die Informationen im Cache des Segmentregisters gespeichert. Daher muss zur Berechnung der virtuellen Adresse zum gegebenen Offset nur noch die im Cache bereits eingetragene Basisadresse des Segmentes addiert werden. Der oben dargestellte Mechanismus läuft also realiter beim Eintrag eines Selektors in ein Segmentregister ab und dann nicht wieder.

Die auf diese Weise berechnete Adresse ist eine »echte«, lineare 32-Bit-Adresse (entstanden aus der 32-Bit-Basisadresse und einem bis zu 32 Bits breiten Offset). Mit ihr kann jede Stelle innerhalb des zur Verfügung stehenden Adressraums des Prozessors angesprochen werden. Warum heißt sie dann »virtuell«?

Aus zwei Gründen. Theoretisch kann jedes Segment bis zu 4 GByte (= 2^{32} Bytes) groß sein. Dies ist identisch mit dem maximal adressierbaren Adressraum von 32-Bit-Prozessoren. Es muss aber mindestens drei Segmente geben, damit der Prozessor arbeiten kann: ein Code-, ein Daten- und ein Stacksegment. Somit kann in praxi kein Segment die theoretische Grenze ausnutzen. Verstärkt wird dies dadurch, dass das Betriebssystem ja auch noch geladen sein muss, was bei den heutigen

Realisationen bereits die Hälfte des »virtuellen Adressraums« von 4 GByte ausmacht. Die berechnete Adresse kann also ebenfalls nur virtuell sein! Zweitens: Wohl nur wenige Prozessoren werden tatsächlich auf einen realen physischen Adressraum von 4 GByte zurückgreifen können, da wohl die meisten Rechner (heute noch) mit erheblich weniger RAM bestückt sein dürften. Auch aus diesem Grunde kann die nach dem oben genannten Mechanismus berechnete Adresse nur virtuell sein.

Das bedeutet, die virtuelle Adresse muss noch in eine tatsächlich adressierbare, reale Adresse abgebildet werden.

Real Mode Im real mode ist der Adressraum mit 20 Adressleitungen und somit 20-Bit-Adressen auf 1 MByte (2^{20} = 1.048.576) beschränkt, einen Adressraum, der bereits bei sehr frühen Prozessoren zur Verfügung stand. Hier muss somit keine Abbildung einer Adresse in einen virtuellen Raum erfolgen, vielmehr kann die logische Adresse direkt zur Berechnung der physikalischen Adresse benutzt werden, wie Abbildung 2.23 zeigt.

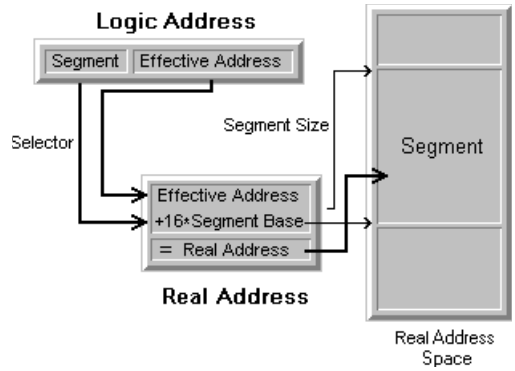


Abbildung 2.23: Berechnung einer realen Adresse aus einer logischen im real mode



Im real mode gibt es keine virtuellen Adressen. Nicht umsonst heißt der real mode ja auch real! Wie das erfolgt, haben wir bereits im Abschnitt »Real Mode« auf Seite 400 erfahren. Auf eine detailliertere Beschreibung kann daher an dieser Stelle verzichtet werden.

2.2.11 Paging: Von der virtuellen zur physikalischen Adresse

Physikalische Adresse

Teil der virtuellen Speicherverwaltung ist die Umsetzung von virtuellen Adressen, wie sie im Rahmen der Speichersegmentierung eingesetzt werden, auf physikalische. Das klingt einfacher, als es vielfach ist. Denn eine solche Umsetzung ist nur dann trivial, wenn der physikalisch vorhandene Speicher den gesamten virtuellen Adressraum abdeckt.

Das aber dürfte auch heute, zu Zeiten relativ billigen Speichers, eher die Ausnahme sein, sprechen wir doch immerhin von 4 GByte! Somit muss mit dem gewirtschaftet werden, was vorhanden ist. Und das sind (in Rechnern im Konsumerbereich) vielleicht einmal gerade 128 MByte, bei »freaks« vielleicht auch einmal 256 MByte. Und auch im Professional-Bereich dürften 256 MByte das sein, was heute »normal« ist (ausgenommen, natürlich, High-End-Server-Systeme!).

Das bedeutet, dass die Kunst nun darin besteht, die 4 GByte virtuellen Speichers abzubilden auf z.B. 128 MByte realen. Und mit dieser Formulierung bekommen Sie auch eine Idee, was eine Speicherverwaltung, die das kann, so nebenher erledigt: Flexibilität hinsichtlich der tatsächlichen Menge physikalischen Speichers. Denn auf einem Rechner sind 128 MByte installiert, auf einem anderen vielleicht 256 MByte, auf wieder einem anderen vielleicht auch nur 64 MByte. Oder 80, 96! Wie funktioniert das?

Pages

Das wird dadurch erreicht, dass der physikalisch verfügbare Speicher in »Seiten«, engl.: *pages* aufgeteilt wird. Eine Page hat eine bestimmte, festgelegte Größe von z.B. 4 kByte oder 4 MByte. Der reale Speicher besteht also aus einer bestimmten Anzahl solcher Pages – bei 64 MByte und 4-kByte-Pages also 16.384 (2^{26} / 2^{12} = 2^{14}).

Page Table

Gleichzeitig wird das Netz der Pages über den virtuellen Adressraum gelegt. Dieser besteht somit aus 2^{32} / 2^{12} = 2^{20} = 1.048.576 Pages. Diesen virtuellen Pages werden nun physikalische Pages zugeordnet, und zwar über Tabellen. So gibt es eine Tabelle, in der für jede virtuelle page die Adresse einer physikalischen Adresse eingetragen wird, wenn möglich. Ist das nicht möglich, so wird der entsprechende Tabelleneintrag entsprechend markiert nach dem Motto: »Tut mir Leid, leider nicht verfügbar«.

Nun ist eine Tabelle aus 1.048.576 solcher Einträge nicht leicht handelbar, umfasst sie doch mindestens 4 MByte Speicher, wenn ein Eintrag aus mindestens vier Bytes = 32 Bit besteht, die für eine physikalische Adresse erforderlich sind. Mindestens deshalb, da je nach Situation eventuell mehr erforderlich werden. Wir werden das sehen! Das aber ist ein nicht unerheblicher Anteil am gesamt verfügbaren Speicherplatz, in unserem Beispiel mit 64 MByte immerhin 6,25% – vor allem, da nicht zu jedem Zeitpunkt jeder der über 1 Mio. Tabelleneinträge benötigt wird. Deshalb hat man sich entschlossen, die Tabelleneinträge auf verschiedene Tabellen zu verteilen. Auf der Suche nach der geeigneten Größe einer solchen Tabelle kam man auf die magische Zahl 1.024. Warum? Zum einen, weil 1.024 mal vier Bytes pro Eintrag 4.096 Bytes sind, also exakt die Größe einer Page. Und zum anderen, weil 1.024 die Quadratwurzel aus 1.048.576 ist – der Gesamtzahl der Pages.

Page Directory

Das bedeutet, der virtuelle Adressraum wird verwaltet von 1.024 Tabellen à 1.024 Einträgen auf je eine Page. Diese 1.024 Tabellen werden wiederum in einer Tabelle geführt, die 1.024 Einträge hat. Diese Tabelle nennt man *page directory.* Sie ist quasi das Inhaltsverzeichnis für die Tabellen, die die Einträge besitzen.

Auslagerungsdatei

Was ist nun gewonnen? Bislang haben wir nur den virtuellen Adressraum kartographiert und mit einem Netz von Pages überzogen, für die Tabellen und eine »Supertabelle« existieren. Ferner wissen wir, dass jede Tabelle einen Eintrag hat, der entweder die physikalische Adresse einer Page im Speicher enthält, der mit dem gleichen Page-Netz kartographiert wurde, oder aber den Vermerk, dass kein physischer Speicher zugeordnet werden konnte. Und genau das ist der Trick! Man wird in der Regel nicht alle Pages des gesamten virtuellen Adressraums gleichzeitig benötigen. Daher könnte man doch die Pages, die gerade nicht benötigt werden, irgendwie aus dem realen Adressraum »auslagern« und bei Bedarf »zurückladen« und gegen dann nicht dringend benötigte austauschen. Genau das erfolgt: Es existiert eine Auslagerungsdatei auf der Festplatte, in der alle Pages eingetragen sind, die gerade »ausgelagert« sind. Wichtig hierbei: In der Auslagerungsdatei liegen Abbilder der Pages, sog. *images*, sodass das einfache »Austauschen« eines Page-Inhaltes mit einem Image ausreicht, die ausgelagerte Information verfügbar zu machen.



Das bedeutet aber, dass eine virtuelle Page an verschiedenen Stellen stehen oder abgebildet sein kann, wie man sagt:

- im physikalisch vorhandenen Speicher an einer Adresse XXXX,
- in der Auslagerungsdatei, falls sie aktuell nicht benötigt wird und nicht ausreichend physikalischer Speicher zur Verfügung steht, um sie aufzunehmen,
- an der physikalischen Adresse YYYY, falls sie im Bedarfsfall zurückgeladen wurde, Adresse XXXX aber durch eine andere benötigte Page belegt ist,
- wiederum in der Auslagerungsdatei, an anderer Stelle, wenn sie wieder ausgelagert werden musste, oder
- an Adresse ZZZZ, falls wieder auf sie zurückgegriffen werden muss.



Und an dieser Stelle wird vielleicht auch klar, was der Exception-Handler leisten muss, sobald eine #NP bzw. #PF ausgelöst wird.

Nach all der Theorie ein wenig Praxis. Wie funktioniert die Berechnung einer physikalischen Adresse nun im Einzelnen? Das hängt davon ab, wie groß die physikalischen Adressen tatsächlich sind. So verfügen die Prozessoren ab dem Pentium Pro über mehr als 32 Adressleitungen, sodass sogar mehr als 4 GByte physikalischer Speicher angesprochen werden können. Somit gibt es zwei grundsätzliche Adressierungsmodi:

- der 32-Bit-Adressierungsmodus zur Unterstützung von 4-GByte-Adressräumen
- der 36-Bit-Adressierungsmodus für 64-GByte-Adressräume, der in zwei Varianten auftritt:
  - im Rahmen einer page size extension (PSE-36-Modus)
  - im Rahmen der physical address extension (PAE-Modus)

Dies ist auf den ersten Blick Verschwendung, da ja virtuelle Adressen in 32-Bit-Systemen weiterhin »nur« 32 Bit breit sind und daher »nur« 4 GByte physikalischen Speicher adressieren können. Bei genauerem Hinsehen jedoch macht auch ein erweiterter Adressraum jenseits der 4 GByte durchaus Sinn. So können ja im 64 GByte bis zu 16 4-GByte-Räume nebeneinander existieren, die sich nicht ins Gehege kommen können. Geeignetes Paging und ein Betriebssystem vorausgesetzt, das so etwas unterstützt, kann es somit sehr wohl Sinn machen, an PSE zu denken, z.B. in Serversystemen. Doch fangen wir langsam und mit dem 32-Bit-Modus an.

32-Bit-Adressierungsmodus

Der im Folgenden dargestellte 32-Bit-Adressierungsmodus ist der »native« Adressierungsmodus, den alle Prozessoren kennen, die im 32-Bit-Protected-Mode arbeiten können. Bei diesem Modus wird eine virtuelle 32-Bit-Adresse in eine physikalische 32-Bit-Adresse übersetzt. Hierbei spielen das PDB-Register (page directory base register, identisch mit Kontrollregister CR3), ein page directory und eine page table eine Rolle.

Ursprung: PDBR

Es beginnt alles mit dem PDBR! Abbildung 2.24 zeigt ein Speicherabbild dieses Registers. Wie man sieht, liegen an den Bits 12 bis 31 die 20 »oberen« Bits einer 32-Bit-Adresse, an der das page directory liegt. Diese Bits werden daher auch »page directory base address« genannt. Durch Verknüpfen des Inhaltes des PDBR mit der Maske $FFFF_F000 (um die 12 »unteren«, reservierten Bits zu löschen) erhält man somit eine Adresse, die an Page-Grenzen (2^{12} = 4.096) beginnt. Das page directory liegt somit selbst in einer 4-KByte-Page, da es mit 1.024 Einträgen à 4 Byte auch exakt eine Page umfasst.



Abbildung 2.24: Speicherabbild des PDBR im 32-Bit-Adressierungsmodus

Die Flags PCD, page-level cache disable, und PWT, page-level writes transparent, steuern das caching und die caching policy des page directory. Ich möchte, weil zu speziell, hier nicht darauf eingehen.

Erste Stufe: Page Directory

Nun braucht der Prozessor einen Index in dieses page directory. Nach dem weiter oben Gesagten sind das die »oberen« 10 Bits der virtuellen Adresse. Sie werden mit vier multipliziert, da jeder Eintrag (»entry«) in der page directory vier Bytes umfasst. An dieser Stelle der page directory findet sich der so genannte *page directory entry*, der auf die page table zeigt, die zuständig ist. Abbildung 2.25 zeigt einen solchen Eintrag.



Abbildung 2.25: Speicherabbild eines Page Directory Entry im 32-Bit-Adressierungsmodus

Auch hier werden nur die »oberen« 20 Bits der 32-Bit-Adresse benötigt, da auch die page tables mit ihren maximal 1.024 Einträgen à vier Bytes genau eine Page beanspruchen und somit an Page-Grenzen beginnen müssen. Neben der Adresse der zuständigen page table ist daher noch Platz genug, einige Attribute zu speichern, die einerseits die Schutzmechanismen unterstützen (U/S, user/supervisor mode; R/W, read-only/read-write; vgl. »Beschränkung der Instruktionen« auf Seite 481), andererseits auch den Paging-Mechanismus selbst betreffen. So gibt P, present, an, ob die Page mit der zuständigen page table, deren Adresse in diesem Eintrag gespeichert ist, im physikalischen Speicher liegt. Ist das der Fall, ist alles in Ordnung und der Prozessor kann die Adresse verwenden. Ist das dagegen nicht der Fall, so wird eine #PF (page fault exception) ausgelöst und der Prozessor lädt die Seite aus der Auslagerungsdatei nach. PWT und PCD kennen wir bereits vom PDBR: Sie kontrollieren die Art und Weise, wie der Cache eingesetzt wird. Das Bit A, accessed, wird gesetzt, wenn auf die page table bereits zugegriffen wurde. Es wird benutzt, um das Rückladen und Auslagern von Pages zu verwalten.

Page size, PS, spielt eine wesentliche Rolle, dem Prozessor mitzuteilen, welche Größe die page table hat, die mit diesem page directory verknüpft ist. Ist dieses Bit gelöscht, werden 4-kByte-Pages verwendet und der page directory entry zeigt auf eine page table. Ist es gesetzt, kommen 4-MByte-Pages zum Einsatz, auf die der Eintrag direkt zeigt. Wir kommen weiter unten darauf zurück.

Das global flag G wurde mit dem Pentium Pro eingeführt und gibt an, dass die referenzierte page table »global verfügbar« sein muss und somit von einem Löschen aus dem translation lookaside buffer (TLB) ausgenommen wird – so das PGE-Flag in Kontrollregister CR4 gesetzt ist.

Avail schließlich gibt drei Bits an, die der Prozessor nicht benutzt und die vom Betriebssystem und/oder dem Paging-Mechanismus verwendet werden können.

Zweite Stufe: Page table

Zurück zur Adressberechnung! Mit der page table base address haben wir somit die Adresse der zuständigen page table. Auch an dieser Stelle benötigen wir einen Index in diese Tabelle, den uns nach dem weiter oben geschilderten die Bits 12 bis 21 der virtuellen Adresse liefern. Analog dem Auslesen des page directory wird somit die page table ausgelesen, indem dieser Index mit vier multipliziert wird (auch Page-Table-Einträge sind vier bytes breit!). An entsprechender Stelle in der page table findet sich ein so genannter page table entry, der in Abbildung 2.26 dargestellt ist.



Abbildung 2.26: Speicherabbild eines Page-Table-Entry im 32-Bit-Adressierungsmodus

Der *page table entry* sieht fast genauso aus wie ein page directory entry: An den Positionen 12 bis 31 finden sich erneut die 20 »oberen« Bits einer 32-Bit-Adresse, die an einer Page-Grenze beginnt – und das ist gut so, ist es doch die Adresse der Page selbst, die wir benötigen.

Die Felder avail, G, A, PCD, PWT, U/S, R/W und P kennen wir schon aus dem page directory entry, sie haben hier analoge Bedeutung. Hinzugekommen ist das Flag D, dirty, das immer dann gesetzt wird, wenn eine Page beschrieben wurde. Es wird zusammen mit dem Flag accessed (A) durch den Paging-Mechanismus ausgewertet. PAT, page attribute table index, wurde mit dem Pentium Pro eingeführt und wird zusammen mit PCD und PWT benutzt, um einen Eintrag in der page attribute table anzuwählen. Einzelheiten hierzu werden nicht genannt, sie würden den Rahmen des Buches sprengen.

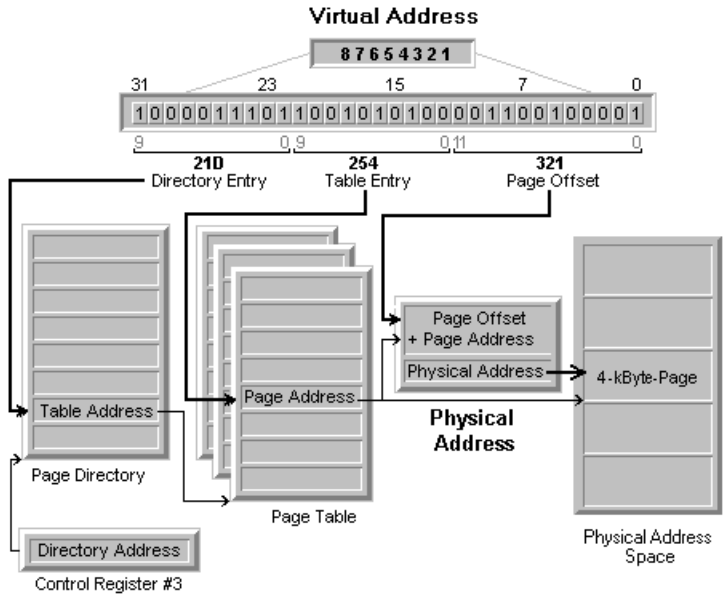


Abbildung 2.27: Berechnung einer physikalischen Adresse aus einer virtuellen unter Benutzung von 4-kByte-Pages im 32-Bit-Adressierungsmodus

Endpunkt: Offset

Da wären wir nun. Von unserer virtuellen Adresse sind die »unteren« 12 Bits noch nicht ausgewertet. Sie stellen den Offset in die Page dar, die wir eben über das PDBR, das page directory und die page table identifiziert haben. Mit diesen 12 Bits lässt sich die gesamte Page von 4 kByte (2^{12} = 4.096) ansprechen.

Abbildung 2.27 zeigt den gesamten Ablauf der Umsetzung einer 32-Bit virtuellen Adresse in eine 32-Bit physikalische Adresse als Schaubild.

Der Vollständigkeit halber sollte noch ein kurzer Blick auf einen page directory entry oder einen page table entry geworfen werden, der auf eine nicht vorhandene Page verweist. Da die Page nicht vorhanden ist, gibt es auch keine Adresse. Und sicherlich machen auch die Flags keinen Sinn! Somit stehen in einem solchen entry die Bits 1 bis 31 dem Paging-Mechanismus zur freien Verfügung, wie Abbildung 2.28 zeigt.

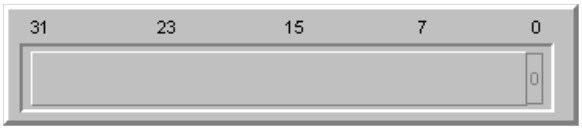


Abbildung 2.28: Speicherabbild eines Page Table Entry und Page Directory Entry für eine Page, die nicht physikalisch verfügbar ist

Der Paging-Mechanismus verwendet diesen Bereich zur Referenz der betreffenden Seite in der Auslagerungsdatei.

PSE-Modus

Bei der Besprechung des page directory entry wurde das Flag PS angesprochen, das die Größe der verwendeten Page angibt. Oben wurden 4-kByte-Pages besprochen, weshalb in diesem Fall PS gelöscht ist. Hier nun wollen wir sehen, was passiert, wenn es gesetzt ist und somit 4-MByte-Pages zum Einsatz kommen. Voraussetzung für die Nutzung von »größeren« Pages ist allerdings, dass der Prozessor mittels Setzen des Flags PSE im Kontrollregister CR4 die Erlaubnis zur Nutzung der page size extension (PSE) erhält – und das überhaupt möglich ist. Ist PSE gelöscht, »kennt« der Prozessor nur 4-kByte-Pages, das PS-Flag spielt dann keine Rolle.

4-MByte-Pages

Um einen Offset in eine 4-MByte-Page zu realisieren, werden 22 Bits benötigt: 2^{22} = 4.194.304. Wir benötigen also unbedingt 22 Bits für diesen Offset. Andererseits haben wir bereits 10 Bits als Index für das page directory belegt. Dies sind insgesamt 32 Bits, also die vollständige 32-Bit-Adresse.

Als Konsequenz fällt in diesem Fall eine gesamte Ebene weg: die page table. Vielmehr zeigt nun die im page directory entry stehende Adresse direkt auf die 4-MByte-Page. Da diese Page jedoch vier MByte umfasst, muss sie an 4-MByte-Grenzen liegen. Zur Codierung einer solchen Adresse reichen somit 12 Bits aus, weshalb der page directory entry in diesem Fall wie in Abbildung 2.29 dargestellt aussieht.



Abbildung 2.29: Speicherabbild eines Page Directory Entry im 32-Bit-Adressierungsmodus bei Verwendung von 4-MByte-Pages

Beachten Sie bitte, dass das Flag PAT hier nicht an Position 7 wie im page table entry einer 4-kByte-Page liegt, sondern an Position 12.

Abbildung 2.30 zeigt den schematischen Ablauf der Umsetzung einer virtuellen 32-Bit-Adresse in eine physikalische für den Fall, das 4-MByte-Pages verwendet werden. Im Vergleich mit Abbildung 2.27 wird die fehlende Ebene (page table) deutlich.

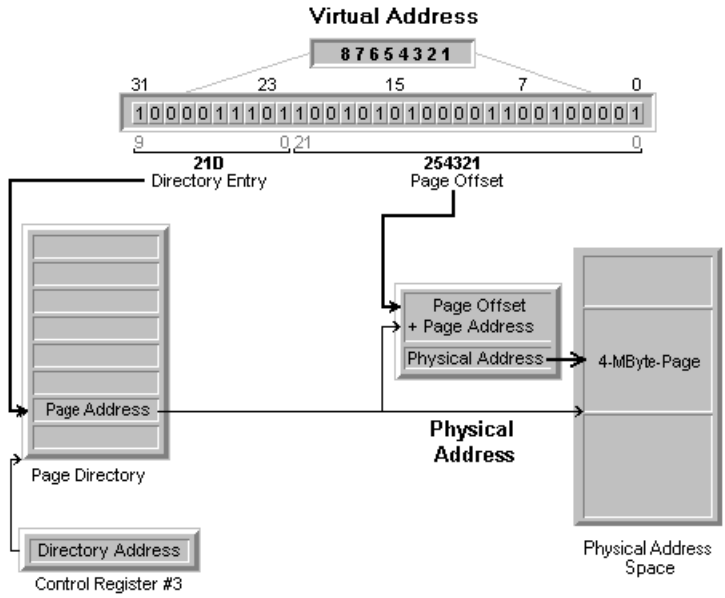


Abbildung 2.30: Berechnung einer physikalischen Adresse aus einer virtuellen unter Benutzung von 4-MByte-Pages im 32-Bit-Adressierungsmodus

PSE-36-Modus

Mit dem Pentium III wurde (nach der Einführung der PAE mit dem Pentium Pro, siehe weiter unten) eine modifizierte Nutzung von 4-MByte-Pages eingeführt, die die Begrenzung der Adressumsetzung auf physikalische Adressen mit 32 Bit aufhob. Der Pentium III verfügt über 36 Adress-Pins, sodass ein physikalischer Adressraum von 3^{36} = 68.719.476.736 Bytes = 64 GByte adressierbar ist.

Wird der PSE-Modus genutzt und somit 4-MByte-Pages ausgewählt (PG-Flag in CR0 gesetzt, PSE-Flag in CR4 ebenfalls gesetzt und PAE-Flag in CR4 gelöscht) und unterstützt der Prozessor 36-Bit-Adressen, so verwendet er den in Abbildung 2.31 gezeigten, modifizierten page directory entry. Ob er das allerdings tatsächlich tut, signalisiert er über ein feature flag des CPUID-Befehls: das PSE-36-Flag.

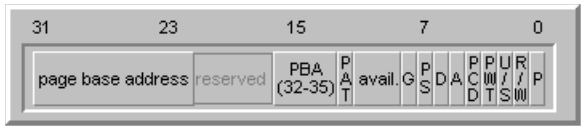


Abbildung 2.31: Speicherabbild eines Page Directory Entry im PSE-36-Adressierungsmodus

Wie man in der Abbildung sieht, werden die Bits 13 bis 16, die im »normalen« page directory entry für 4-MByte-Pages reserviert sind (vgl. Abbildung 2.29) benutzt, um die Bits 32 bis 35 der 36-Bit-Adresse aufzunehmen.

Das bedeutet, dass der PSE-36-Modus im Prinzip nichts anderes als ein »aufgebohrter« 32-Bit-Adressierungsmodus ist, der analog zum 32-Bit-Modus, jedoch mit 36-Bit-Adressen, für die zum Einsatz kommenden 4-MByte-Pages arbeitet. Sie können daher Abbildung 2.30 mit Fug und Recht zur Veranschaulichung auch dieser Adressierungsart heranziehen.

PAE-Modus

Eine etwas andere Art der Adressierung wird beim PAE-Adressierungsmodus verwendet, der mit dem Pentium Pro eingeführt wurde. Er ist aktiv, wenn das PG-Flag in Kontrollregister CR0 (enable paging) gesetzt, das PSE-Flag (page size extension) in CR4 gelöscht und das PAE-Flag (physical address extension) in CR4 gesetzt ist. Ihn kann man am besten damit umschreiben, dass in diesem Modus

- eine weitere Ebene der Adressumrechnung eingeführt wurde. So gibt es neben page tables und page directories nun noch eine »höhere« Ebene, die *page directory pointer tables*.

- tatsächlich auf allen Ebenen mit Ausnahme des Kontrollregisters CR3 mit echten 36-Bit-Adressen gearbeitet wird.

Ursprung PDPTR

Das hat als erste Konsequenz, dass Kontrollregister CR3 nun nicht mehr den Alias PDBR (page directory base register) hat, sondern PDPTR, *page directory pointer table register*. Dieser rein formale Unterschied (so bewirkt ja ein bloßer Wechsel der Benennung des Registers noch gar nichts!) hat einen tief greifenden Hintergrund: Nun enthält das Register nicht mehr die »oberen« 20 Bits einer 32-Bit-Adresse für das page directory, sondern die »oberen« 27 Bits einer 32-Bit-Adresse für eine page directory pointer table. Das bedeutet, diese Tabelle muss innerhalb der »unteren« 4 GByte des 64-GByte-Adressraums an 32-Byte-Adressen liegen. Abbildung 2.32 stellt das PDPTR dar. Weitere Veränderungen an diesem Register hat es nicht gegeben (vgl. Abbildung 2.24).

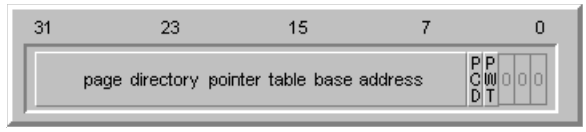


Abbildung 2.32: Speicherabbild des PDPTR im PAE-Adressierungsmodus

Erste Stufe: Page Directory Pointer Table

Doch nun wird es ernst mit den 36-Bit-Adressen! Anders als im PSE-Modus erfolgt die Speicherung einer 36-Bit-Adresse im PAE-Modus nicht mit Hilfe des im Tabelleneintrag reservierten Bereiches, sondern dadurch, dass ein Eintrag nun doppelt so breit ist wie im 32-Bit-Modus, nämlich 64 Bit = 8 Byte. Die Bits 32 bis 35 eines solchen Eintrags stellen die »oberen« vier Bits der 36-Bit-Adresse dar. Dies kann man in Abbildung 2.33 erkennen. Grund: Nachdem nun auf allen Ebenen mit 36-Bit-Adressen gearbeitet wird, muss auch in allen Entries eine 36-Bit-Adresse gespeichert werden können. Dies kann jedoch nicht in Form der vom PSE-Modus verwendeten Nutzung reservierter Bereiche eines Eintrags erfolgen, da im PAE-Modus die Einträge solche reservierten Bereiche im erforderlichen Ausmaß nicht kennen. Die Erweiterung eines Eintrags auf 8 Byte hat noch einen weiteren Vorteil: Mit 36 Bit ist nicht das Ende der Fahnenstange für physikalische Adressen erreicht. So gibt es, wie die Abbildung zeigt, genügend Platz, um auch 64-Bit-Adressen verwalten zu können. Ob das allerdings in 32-Bit-Rechnern jemals der Fall sein wird oder eine über 36-Adressleitungen hinausgehende Adressierung mit den »echten« 64-Bit-Rechnern vom Typ Itanium erfolgen wird, weiß Intel allein ...

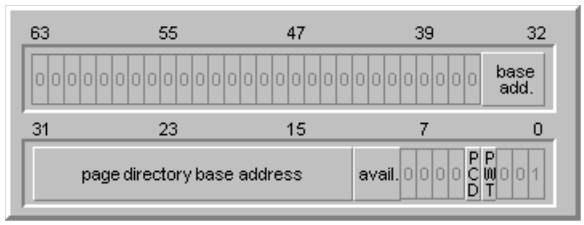


Abbildung 2.33: Speicherabbild eines Page Directory Pointer Table Entry im PAE-Adressierungsmodus

Wie auch immer. Das PDPTR zeigt nun in eine Tabelle, die maximal vier Einträge vom Typ page directory pointer table entry. Warum es ausgerechnet vier sind und nicht mehr, werden wir weiter unten erfahren. In jedem Eintrag dieser Tabelle steht nun die 36-Bit-Adresse einer page directory table, die wir aus dem 32-Bit-Modus bereits kennen. Die PDPT (page directory pointer table) ist somit eine zusätzliche Stufe in der Hierarchie der Adressumrechnungsstrukturen.

Ansonsten gibt es zum page directory pointer table entry wenig zu sagen. Ein Vergleich mit dem in Abbildung 2.24 gezeigten PDBR des 32-Bit-Modus zeigt, dass der grundsätzliche Aufbau der Einträge in beiden Fällen der gleiche ist, die PDPT also als vier PDBRs aufgefasst werden kann und das PDPTR somit lediglich angibt, welches dieser »Software-PDBRs« ausgewählt ist. Die Unterschiede beschränken sich auf die zusätzlichen 32 Bit des Eintrages aufgrund der 36-Bit-Adressen sowie auf die Tatsache, dass die Entries nun auch einen für den Paging-Mechanismus nutzbaren Bereich (»avail«) haben, den das PDBR nicht besitzt und besitzen muss.

Zweite Stufe: Page Directory Table

Hangeln wir uns anhand der Erkenntnisse aus dem 32-Bit-Adressierungsmodus weiter. Jeder PDPT-Eintrag zeigt nun mit einer 36-Bit-Adresse auf eine page directory. Bei dieser Tabelle handelt es sich um die gleiche Art Tabelle, die im 32-Bit-Modus auch verwendet wird, um die page tables zu referenzieren. Mit den nun mittlerweile bekannten Erweiterungen, die im Rahmen des PAE-Mechanismus erforderlich sind, stellt uns Abbildung 2.34 vor keine wirklichen Probleme.

Bis auf die Tatsache, dass das global flag G nicht verzeichnet ist, da es im PAE-Modus nicht benötigt wird, und der 36-Bit-Erweiterung gibt es keinen Unterschied zum page directory entry im 32-Bit-Modus (vgl. Abbildung 2.25).

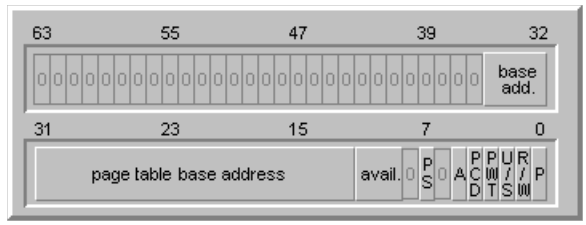


Abbildung 2.34: Speicherabbild eines Page Directory Entry im PAE-Adressierungsmodus

Dritte Stufe: Page Table

Und auch die vierte Stufe der Adressumsetzung kann schnell abgehandelt werden, zeigt doch der page table entry in Abbildung 2.35 verglichen mit dem aus dem 32-Bit-Modus (Abbildung 2.26) nur die inzwischen bereits erwarteten Unterschiede.

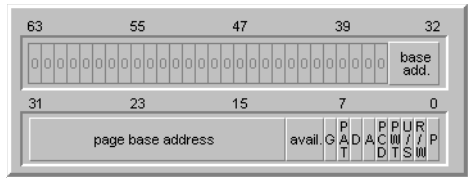


Abbildung 2.35: Speicherabbild eines Page-Table-Entry im PAE-Adressierungsmodus bei Verwendung von 4-kByte-Pages

Endpunkt: Offset

Auch im PAE-Modus bildet der Offset die letzte Stufe der Adressberechnung. Da auch in diesem Modus 4-kByte-Pages zum Einsatz kommen, sind die Offsets wie im 32-Bit-Modus 12 Bit breit (2^{12} = 4.096). Somit können wir für die Adressberechnung im PAE-Modus das in Abbildung 2.36 dargestellte Schema festhalten.

Indices

Doch bleiben wir noch einen Moment bei der virtuellen 32-Bit-Adresse, die Ausgangspunkt für jede Berechung der physikalischen ist. Weiter oben haben wir festgestellt, dass neben dem 12-Bit-Offset zweimal 10 Bit als Index für die beiden Tabellen (page directory und page table) verwendet werden. Das ist hier auch der Fall, jedoch mit einer kleinen Änderung. So wird auch hier für die beiden Tabellen eine einzelne Page mit 4 kByte verwendet. Da aber im PAE-Modus die Einträge in die Tabellen doppelt so breit sind wie im 32-Bit-Modus, können nur die Hälfte der Einträge aufgenommen werden, nämlich 512. Um diese Anzahl von Einträgen zu indizieren, werden allerdings »nur« 9 Bits benötigt (2^9 = 512). Das bedeutet, die beiden Tabellenindices im PAE-Modus besitzen jeweils nur 9 Bits, was sich zusammen mit dem 12-Bit-Offset zu 30 Bit

addiert. Was passiert mit den verbliebenen zwei Bits der virtuellen 32-Bit-Adresse?

PDPT-Index

Sie werden es erraten: Diese beiden Bits stellen den Index in die page directory pointer table dar, die als zusätzliche Hierarchiestufe eingeführt worden war. Mit ihm lassen sich somit $2^2 = 4$ Einträge indizieren. Und genau das ist auch der Grund, warum die PDPT »nur« vier Einträge hat. Das bedeutet aber, dass sich diese zusätzliche Hierarchiestufe zwangsläufig dadurch ergeben hat, dass Tabelleneinträge in die page directory oder page table im 36-Bit-PAE-Modus doppelt so breit sind wie im 32-Bit-Modus. Somit ist keinerlei »zusätzliche« Funktionalität damit verbunden!

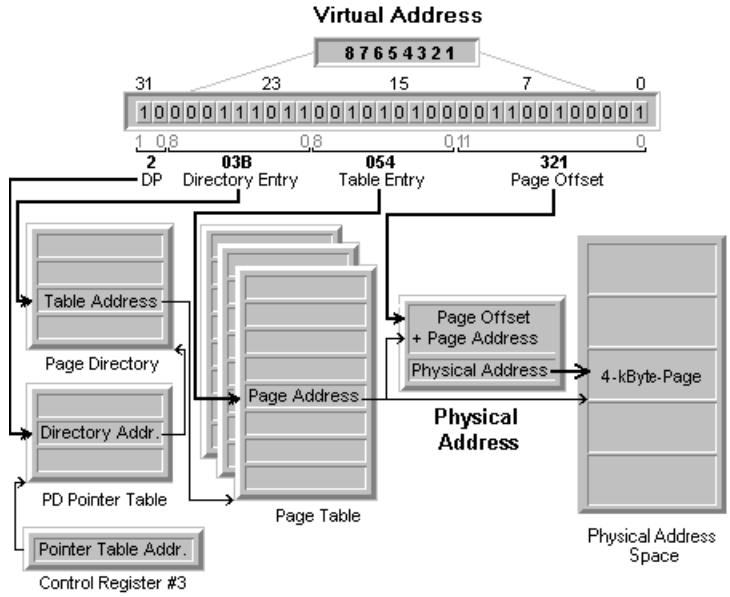


Abbildung 2.36: Berechnung einer physikalischen Adresse aus einer virtuellen unter Benutzung von 4-kByte-Pages im PAE-Adressierungsmodus

2-MByte-Pages

Wenn nun aber der PAE-Adressierungsmodus dem 32-Bit-Adressierungsmodus so ähnlich ist, kann man dann erwarten, dass auch 4-MByte-Pages anstelle der 4-kByte-Pages zum Einsatz kommen? Ja und nein! Ja: Es gibt größere Pages. Nein: Sie umfassen nicht 4 MByte, sondern »nur« 2 MByte. Und der Grund hierfür ist bei einigem Nachdenken auch klar: Analog dem 32-Bit-Modus entfällt hier die Ebene der page table. Das bedeutet, dass der page directory entry direkt auf die Page zeigt, somit ein etwas anderer Entry erforderlich wird. Er ist in Abbildung 2.37 dargestellt.

Damit bleibt jedoch die Struktur als solche bestehen: 2 Bits der virtuellen 32-Bit-Adresse als Index in die PDPT und 9 Bits als Index in das page directory. Macht 11 Bits. Bleiben 21 Bits für den Offset übrig. Und das resultiert in einer 2-GByte-Page: 2^{21} = 2.097.152 Bytes. Aus diesem – und nur aus diesem – Grunde gibt es im PAE-Modus nur 2-GByte-Pages neben der klassischen Form der 4-kByte-Pages. Abbildung 2.38 fasst den Weg der Adressumrechnung im 2-MByte-PAE-Modus schematisch zusammen.

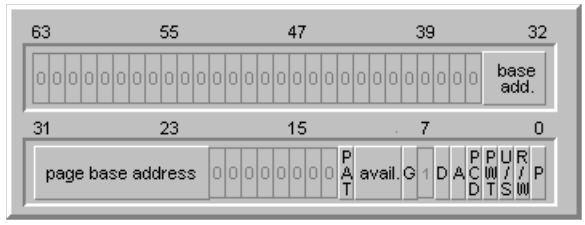


Abbildung 2.37: Speicherabbild eines Page Directory Entry im PAE-Adressierungsmodus bei Verwendung von 2-MByte-Pages

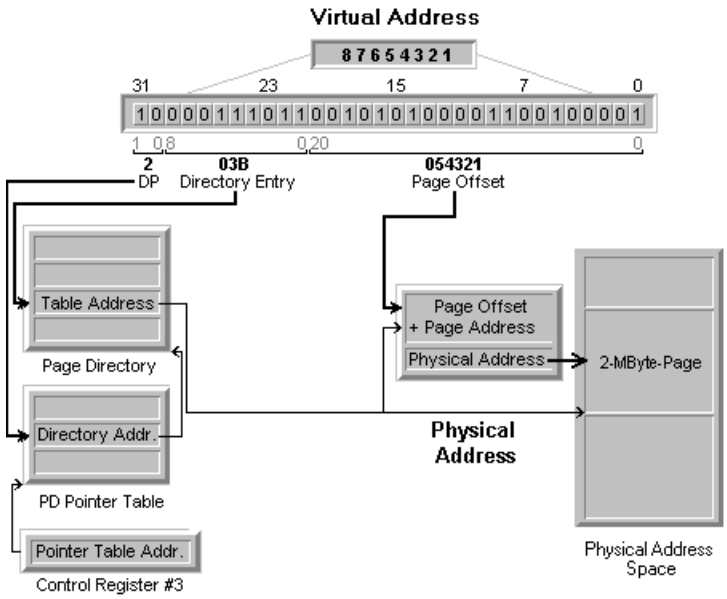


Abbildung 2.38: Berechnung einer physikalischen Adresse aus einer virtuellen unter Benutzung von 2-MByte-Pages im PAE-Adressierungsmodus

Anwahl des Adressierungsmodus

Wie kann man nun die unterschiedlichen Adressierungsmodi und Page-Größen einstellen? Hierbei spielen mehrere Flags eine Rolle, die wir jeweils kurz angesprochen haben. Sie bewirken in unterschiedlicher Kombination, welcher Modus und welche Page-Size eingestellt wurde.

Tabelle 2.1 zeigt die Zusammenhänge. Paging ist nur möglich, wenn das PG-Flag in CR0 gesetzt ist, andernfalls erfolgt eine direkte Übersetzung von virtuellen Adressen in physikalische, unabhängig von den Stellungen der restlichen Flags.

Ist PG gesetzt, so signalisiert PSE, page size extension, ob das page size flag (PS) im page directory entry (PDE) Bedeutung hat oder nicht. Ist PSE gelöscht, ist eine page size extension nicht vorgesehen und die Stellung von PS spielt keine Rolle: Physikalische Adressen sind 32-Bit breit und Pages grundsätzlich 4 kByte groß.

Ist PSE dagegen gesetzt, so werden 4-kByte-Adressen verwendet, wenn PS gelöscht ist, und 4-MByte-Adressen, wenn es gesetzt ist. Die Breite der physikalischen Adressen ist 32 Bit – es sei denn, der CPUID-Befehl signalisiert mit seinem gesetzten PSE-36-Flag, dass der Prozessor die Bits 13 bis 16 als zusätzliche Adressbits verwendet. In diesem Fall stellt ein gelöschtes PS-Flag den 32-Bit-4-kByte-Modus ein, ein gesetztes PS-Flag den 36-Bit-4-MByte-Modus.

PG (CR0)	PSE (CR4)	PAE (CR4)	PS (PDE)	address size	page size
0	0 / 1	0 / 1	0 / 1	32 Bits	kein paging
1	0	0	0 / 1	32 Bits	4 kByte
1	1	0	0	32 Bits	4 kByte
1	1	0	1 a)	32 Bits	4 MByte
1	1	0	1 a)	36 Bits	4 MByte
1	0	1	0	36 Bits	4 kByte
1	0	1	1	36 Bits	2 MByte
1	1	1	0 / 1	verboten	-

a) Ob 32- oder 36-Bit-Adressen verwendet werden, ist prozessorabhängig. Unterstützt ein Prozessor den PSE-36-Modus, was über das PSE-36 feature flag des CPUID-Befehls eruierbar ist, werden die Bits 13 bis 16 genutzt, was die Nutzung von 36-Bit-Adressen erlaubt. Andernfalls sind diese Bits reserviert.

Tabelle 2.1: Flags und Flagstellungen zur Definition des verwendeten Paging-Modus und der Größen der Pages

Wenn PAE gesetzt ist, wird die physical address extension verwendet, die 36-Bit-Adressen benutzt. Das PS-Flag im PDE entscheidet dann darüber, ob 4-kByte- oder 2-MByte-Pages zum Einsatz kommen. Die Kom-

bination gesetztes PG-, PSE- und PAE-Flag ist verboten. (Zumindest bislang, da sich ja physical address extension und page size extension gegenseitig ausschließen!)

page size mixing Besteht die Möglichkeit, Pages verschiedener Größe gleichzeitig einzusetzen? Ja – solange entweder das PSE- oder das PAE-Flag gesetzt ist. Da in jedem page directory entry das PS-Flag existiert, kann für jeden Entry bestimmt werden, ob die »kleinen« oder »großen« Pages eingesetzt werden – und ob somit die Adresse des PDE auf eine page table zeigt oder direkt auf eine »große« Page.

Mehr noch: page size mixing ist nicht nur möglich, sondern wird vom Betriebssystem auch heftig benutzt. So werden der Kernel und wichtige, häufig benötigte Systemteile in 4-MByte-Pages abgelegt.

Übersicht Erinnern Sie sich noch an unsere black box auf Seite 434? Mit den neu gewonnenen Erkenntnissen lässt sie sich transparent machen.

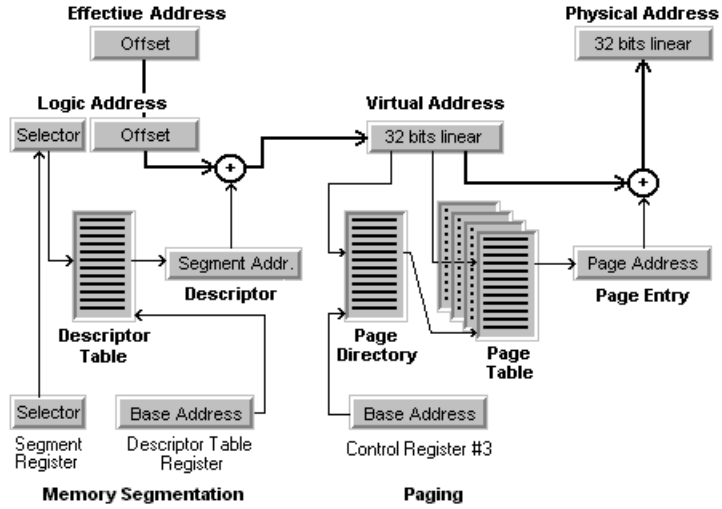


Abbildung 2.39: Umsetzung einer effektiven Adresse in eine physikalische durch Speichersegmentierung und Paging mit 4-kByte-Pages im 32-Bit-Adressierungsmodus

Ziehen wir kurz ein Resümee. Wie Sie gesehen haben, ist die Umsetzung einer virtuellen Adresse in eine physikalische im protected mode im Prinzip in allen zur Verfügung stehenden Adressierungsmodi die gleiche. In Abbildung 2.39 ist daher der lange Weg von der effektiven

Adresse zur physikalischen Adresse exemplarisch am Beispiel der Umsetzung im »nativen« 32-Bit-Modus gezeigt.

Zur Adressberechnung sind ein Segmentregister, ein Systemregister und ein Kontrollregister erforderlich, ferner spielen zwei bis vier Systemsegmente eine Rolle: eine Deskriptoren-Tabelle (GDT oder LDT), ggf. eine page directory pointer table, ein page directory und ggf. eine page table. Im Hintergrund spielen jedoch noch erheblich mehr Komponenten eine Rolle, auf die wir hier leider nicht näher eingehen können, da das den Rahmen dieses Buches sprengen würde. Zu nennen wären first und second level cache und die TLBs (translation lookaside buffer).

Page Attribute Table

Page attribute tables (PATs) sind eine Erweiterung der page tables, die wir ja hinreichend besprochen haben. Sie wurden mit dem Pentium III eingeführt, arbeiten mit den Flags PWT und PCD zusammen und haben als physikalischen Background spezielle MTRRs (*machine type range registers*).

Mit Hilfe der PATs ist es in Verbindung mit den betreffenden MTRRs möglich, bestimmte Speichertypen (»uncacheable memory«, »write-combining memory«, »write-through memory«, »write-protect memory« und »write-back memory«) im Rahmen der virtuellen Speicherverwaltung bestimmten physikalischen Bereichen des Speichers zuzuordnen. Die MTRRs übernehmen hierbei die physikalische Zuordnung: Sie ordnen bestimmte Speichertypen bestimmten Regionen des physikalischen Speichers zu; die PATs nun agieren wie die page tables, indem sie bei der Umrechnung der Adressen die entsprechenden pages zuordnen. Dies soll an Informationen im Rahmen dieses Buches genügen.

2.2.12 Auslagerungsdatei

Kommen wir noch einmal auf die Auslagerungsdatei zurück, die wir am Anfang dieses Kapitels angesprochen haben. Man kann sich die Art und Weise, wie der physikalisch verfügbare Speicher und die Datei im Rahmen der virtuellen Speicherverwaltung zusammenarbeiten, wie folgt vorstellen. RAM (also physikalisch verfügbarer Speicher) und Auslagerungsdatei bilden ein Kontinuum, das den benötigten Adressraum vollständig abbildet (was nicht notwendigerweise die gesamten 4 GByte sein müssen!). RAM und Datei sind in die Pages unterteilt. Und der Prozessor kann nun auf alle Pages, die er benötigt, direkt zugreifen. In dieser Art der Vorstellung ist somit die Auslagerungsdatei nichts anderes als eine spezielle Form von RAM! Es ist somit durchaus gestattet,

die Größe des RAMs und die Größe der von Windows verwalteten Auslagerungsdatei (z.B. WIN386.SWP bei Win98 oder PAGEFILE.SYS bei Win2000) zu addieren, um den verfügbaren Adressraum zu bestimmen. Ist beispielsweise der RAM 128 MByte groß und die Auslagerungsdatei weitere 128 MByte, so verfügt der Prozessor über 256 MByte »physikalischen« Speicher.

Hierbei bleibt natürlich unberücksichtigt, dass der Prozessor auf Daten in der Auslagerungsdatei nicht in der Weise zugreifen kann, wie er das beim RAM tun kann. Aber dazu existiert ja der Paging-Mechanismus, der zu diesem Zweck einfach Seiten zwischen RAM und Datei austauscht.

2.2.13 Das 32-Bit-Betriebssystem Windows

Nach den ausführlichen Betrachtungen zur Speicherverwaltung im protected mode möchte ich noch kurz etwas zu dem Betriebssystem sagen, das diese Möglichkeiten implementiert und wohl aufgrund des ausgewiesenen Marktanteils zu den wichtigsten gehört: Windows in seinen verschiedenen Versionen.

Windows – genauer: 32-Bit-Windows! – gibt es in zwei Versionen:

- dem »Customer-Windows« Windows 95, 98, 98 SE (»second edition«) und ME (»millenium edition«)
- dem »Professional-Windows« NT 3.x, NT 4.x, 2000, XP.

Neben den allseits bekannten Unterschieden zwischen diesen Versionen gibt es auch Unterschiede, was die Speicherverwaltung betrifft. Unterschiede gibt es sogar »innerhalb« einer bestimmten Version. So kann Windows 2000 beispielsweise in einer 2- oder 3-GByte-Benutzermodus-Version installiert werden. Gemeinsam haben jedoch alle diese Versionen einen virtuellen Adressraum von 4 GByte.

Win 98 Dieser Adressraum wird unter Windows 95, 98 (SE) und ME wie folgt partitioniert:

- $0000_0000 bis $0000_0FFF (4 kByte): Null-Selektor
- $0000_1000 bis $003F_FFFF (4 MByte – 4kByte): MS-DOS- und 16-Bit-Windows-Kompatibilitätspartition
- $0040_0000 bis $7FFF_FFFF (2 GByte minus 4 MByte): User-Partition
- $8000_0000 bis $BFFF_FFFF (1 GByte): gemeinsamer MMF-Bereich
- $C000_0000 bis $FFFF_FFFF (1 GByte): Kernel-Partition

Win 2000

Windows 2000 und NT sowie vermutlich auch die 32-Bit-Version von Windows XP (genaue Unterlagen zu XP lagen mir zur Zeit der Erstellung des Manuskriptes noch nicht vor!) verzichten auf den MS-DOS-Kompatibilitätsbereich und den gemeinsamen MMF-Bereich.

- $0000_0000 bis $0000_FFFF (64 kByte): Null-Selektor
- $0001_0000 bis $7FFE_FFFF (2 GByte minus 128 kByte): User-Partition
- $7FFF_0000 bis $7FFF_FFFF (64 kByte): Geschützter Bereich
- $8000_0000 bis $FFFF_FFFF (2 GByte): Kernel-Partition

Diese Partitionen sind in Abbildung 2.40 für die beiden Vertreter des »Consumer-« und »Professional-Windows« Windows 98 und Windows 2000 dargestellt. Bitte beachten Sie, dass die Größenverhältnisse der Partitionen nicht maßstabsgetreu sind und lediglich zur Veranschaulichung dienen.

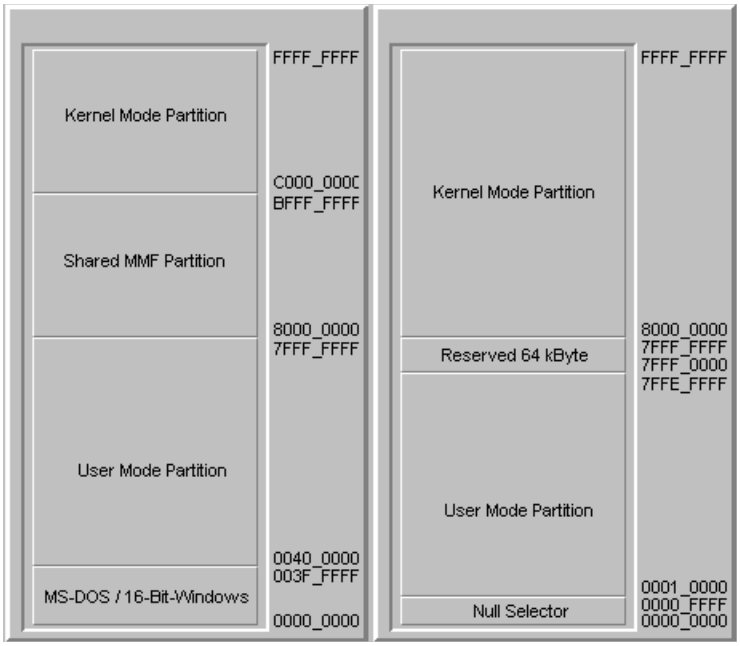


Abbildung 2.40: Aufteilung des virtuellen 4-GByte-Speichers unter Windows 98 (links) und Windows 2000 (rechts)

Was verbirgt sich nun hinter den einzelnen Partitionen?

Null-Selektor

Der Bereich des »Null-Selektors«, der die untersten Bytes der entsprechenden Partition umfasst, dient hauptsächlich zur Unterstützung des Programmierers. Dieser Bereich ist zugriffsgeschützt, weshalb ein Zugriff auf eine Position in diesem Bereich eine #GP auslöst. Auf diese Weise können nicht-initialisierte Zeiger leicht aufgefunden werden. Zum Thema Null-Selektor kommen wir auch weiter unten (vgl. Seite 430).

MS-DOS- und Win16-Kompatibilität

Das »Consumer-Windows« legt aufgrund der sehr beliebten Spiele im MS-Modus oder alten 16-Bit-Windows-Applikationen einen gewissen Wert auf MS-DOS- und Win16-Kompatibilität. Daher wurden die Bereiche, die diese beiden Betriebssysteme verwenden, in dieser Windows-Version berücksichtigt. Win32-Applikationen sollten daher diesen Speicherbereich meiden, vor allem, da es aus »technischen Gründen« nicht möglich war, diesen Adressbereich gegen den Zugriff aus dem User-Bereich zu schützen! Das bedeutet, dass jeder Prozess in diesem Bereich Unheil anrichten kann.

User-Partition

Hierbei handelt es sich um den Bereich, den Applikationen benutzen können, die im User-Modus arbeiten (also mit Privilegstufe 3, vgl. Seite 467). Anders ausgedrückt ist das der »private« Bereich eines jeden Prozesses. Er ist gegen den Zugriff aus anderen Prozessen geschützt.

In diesen Bereich kommen somit alle EXE- und DLL-Module, die den Prozess ausmachen. Unter Windows 2000 und den anderen »Professional-Windows«-Versionen werden auch die MMFs (memory mapped files) abgebildet, auf die dieser Prozess zugreifen kann. Die »Consumer-Windows«-Versionen benutzen hierfür eine eigene Partition (s. u.).

Reserved 64 kByte

Microsoft hat diesen Bereich geschützt, um die Implementierung seines Betriebssystems zu erleichtern. Für Einzelheiten hierzu verweise ich auf Sekundärliteratur, da dies hier nicht von Bedeutung ist.

Shared MMF Partition

In Win98 als Vertreter des »Consumer-Windows« werden in diesem Bereich neben den Prozessmodulen auch die wichtigsten Win32-System-DLLs abgelegt: KERNEL32.DLL, ADVAPI32.DLL, USER32.DLL und GDI32.DLL. Auf diese Weise hat jeder Prozess direkten Zugriff auf diese vier DLLs, die sogar bei allen Prozessen an der gleichen Adresse liegen. Zusätzlich werden alle speicherbasierten Dateien (memory mapped files, MMF) in diesem Bereich abgebildet. Das sind Dateien, die in der Regel große Datenmengen enthalten und über Puffer ausgelesen und beschrieben werden. Auf MMFs wird in der Regel über die Windows-Betriebssystemaufrufe *CreateFileMapping* (Definition eines Adressbereiches für MMFs) und *MapViewOfFile* (Bereitstellung des Speichers) zugegriffen.

Win2000 und alle anderen »Professional-Windows«-Versionen kennen diese MMF-Partition nicht!

Kernel Partition

In dieser Partition findet sich das Betriebssystem, vor allem die Teile, die im kernel mode (vgl. Seite 467) ablaufen. Insbesondere handelt es sich hierbei um die Funktionen zur Speicherverwaltung (vgl. Seite 434), zur Verwaltung von Tasks (vgl. Seite 462) und Threads, zur Verwaltung des Dateisystems sowie die verwendeten virtuellen Gerätetreiber. Hinzu kommen die Systemtabellen und I/O-Puffer.



Das »Consumer-Windows« hat dem »Professional-Windows« eines voraus: Die Größe der User-Partition steht dem Prozess vollständig und ausschließlich zur Verfügung, MMFs und Betriebssystem-DLLs werden in die *shared MMF partition* geladen, die es beim »Professional-Windows« nicht gibt. Fasst man somit *MMF partition* und *user partition* zusammen, stehen unter »Consumer-Windows« einem Prozess satte 3 GByte Adressraum zur Verfügung. Das »Professional-Windows« stellt hierzu nur die 2 GByte der *user partition* bereit. Sehr schnell wurde daher die Forderung laut, auch bei den High-End-Windows-Versionen 3 GByte »user partition« zur Verfügung zu stellen.

Microsoft hat dieser Forderung in der Windows-Version Windows 2000 Advanced Server und Windows 2000 Data Center Rechnung getragen. Die Belegung des Adressraums lautet in diesen Versionen:

- $0000_0000 bis $0000_FFFF (64 kByte): Null-Selektor
- $0001_0000 bis $BFFE_FFFF (3 GByte minus 128 kByte): User-Partition
- $BFFF_0000 bis $BFFF_FFFF (64 kByte): Geschützter Bereich
- $C000_0000 bis $FFFF_FFFF (1 GByte): Kernel-Partition



Dies hat aber Konsequenzen. Der Kernel in Windows 2000 ist unter der Maßgabe entwickelt worden, nicht mehr als 2 GByte Adressraum zu verwenden. Dies hat »gerade so« geklappt. Die Reduktion des Kernels auf 1 GByte muss somit mit Einschränkungen erkauft werden. Dies äußert sich in der Anzahl der Threads, Stacks und verschiedener Ressourcen, die verfügbar sind. Sie musste drastisch reduziert werden. Außerdem unterstützen diese Versionen von Windows 2000 nicht mehr 64 GByte RAM (siehe »PSE-36-Modus« und »PAE-Modus« ab Seite 449), sondern nur noch 16 GByte, da nicht mehr genügend virtueller Adressraum zur Verfügung steht, den zusätzlichen RAM zu verwalten.

64-Bit-Windows

Windows 2000 und Windows XP gibt es in einer 64-Bit-Version, die auf den neuen Intel-Prozessoren (IA-64: Itanium) und den Alpha-Prozessoren läuft. Mit den 64 Adress-Pins dieser Prozessoren lassen sich exakt 16 EByte (Exa-Byte = 18.446.744.073.709.551.616 Bytes) adressieren. Hier wird dieser gigantische Adressraum wie folgt aufgeteilt (wobei auch in diesem Fall die Angaben für 64-Bit-Windows XP nicht anhand öffentlich zugänglicher Dokumentationen verifiziert werden konnten):

- $0000_0000_0000_0000 bis $0000_0000_0000_FFFF (64 kByte): Null-Selektor.
- $0000_0000_0001_0000 bis $0000_03FF_FFFE_FFFF (4 TByte [Tera-Byte] = 70.368.744.177.663 Byte minus 64 kByte): User-Partition
- $0000_03FF_FFFF_0000 bis $0000_03FF_FFFF_FFFF (64 kByte): geschützter Bereich
- $0000_0400_0000_0000 bis $FFFF_FFFF_FFFF_FFFF (16 EByte minus 4 TByte): Kernel-Partition.

Bei der Belegung des Speichers wurde darauf geachtet, dass er möglichst kompatibel mit der Auslegung im 32-Bit-Modus ist, damit »alte« 32-Bit-Software einfacher in die 64-Bit-Welt portiert werden kann. Es ist daher durchaus möglich, dass sich in verschiedenen 64-Bit-Versionen der 64-Bit-Betriebssysteme einiges ändern und verschieben kann.

2.3 Multitasking

Als Multitasking bezeichnet man die Fähigkeit moderner Prozessoren, mehr als einen Task gleichzeitig ablaufen zu lassen.



Vorsicht! Echtes Multitasking benötigt somit mehr als einen Prozessor. Denn es ist schlechterdings unmöglich, einen einzelnen Prozessor gleichzeitig zwei oder mehrere verschiedene Dinge durchführen zu lassen. Das bedeutet, dass es echtes Multitasking nicht gibt – auch nicht auf Mehr-Prozessor-Systemen! Denn das würde bedeuten, dass pro Task ein Prozessor verfügbar ist. Und wer sich einmal im Task-Manager von Windows angeschaut hat, was alles für Tasks laufen, selbst wenn Sie nicht ein einziges Anwendungsprogramm gestartet haben, wird schnell feststellen, dass z.B. nur um die Oberfläche von Windows sichtbar zu machen, bereits ungefähr zwei Hände voll Tasks erforderlich sind.

Das bedeutet, dass Mehrprozessorsysteme lediglich die zu bewältigenden Aufgaben auf mehrere Prozessoren verteilt, die in anderen Systemen ein Prozessor erledigen muss. Und das wiederum bedeutet, dass Multitasking für einen Mechanismus steht, der dem Anwender lediglich vorgaukelt, dass mehrere Tasks gleichzeitig ablaufen.

Dies erfolgt, indem der Prozessor Zeitscheiben verteilt und jedem Task, der »aktiv« ist, eine solche Zeitscheibe zuordnet. In dieser Zeitscheibe, einem Zeitraum von Bruchteilen einer Sekunde, widmet die CPU ihre ganze Aufmerksamkeit dem zugeordneten Task. Ist der zugeordnete Zeitraum verstrichen, wendet sie sich einem anderen Task zu.

Task Switching

Diesen Vorgang nennt man task switching. Wählt man die Zeitscheibe klein genug (aber auch nicht so klein, dass in ihrem Verlauf keine sinnvolle Aktion mehr möglich ist!), findet häufig ein task switch statt. Und wenn nun nicht »zu viele« Tasks aktiv sind, entsteht tatsächlich der Eindruck, die CPU verarbeite die Tasks parallel – ganz so, wie der Eindruck einer kontinuierlichen Bewegung entsteht, wenn man geeignete »Einzelbilder« mit einer bestimmten Frequenz darstellt: Kino, Fernsehen und Video leben davon!

Was heißt nun »klein genug« und »zu viele«? Das ist nicht einfach vorherbestimmbar, da es viele Einflussparameter gibt. Die Leistungsfähigkeit der CPU zum Beispiel. Es ist eine Binsenweisheit, dass der Eindruck von parallel ablaufenden Tasks umso größer ist, je mehr Befehle die CPU in der Zeitscheibe ausführen kann und je kleiner die Zeitscheibe gewählt werden kann. Das bedeutet: je schneller die CPU, desto mehr »Parallelität« – man spricht von »Quasi-Parallelität«! Aber auch das Betriebssystem spielt eine bedeutende Rolle. Denn es muss ja diese task switches durchführen. Und dazu ist ein gewisser Verwaltungsaufwand und – logischerweise – ein Wasserkopf an Befehlen erforderlich. Je kleiner dieser Wasserkopf gehalten werden kann, desto schneller können task switches erfolgen. (Ganz wie im täglichen Leben: Je größer das Management eines Unternehmens, desto mehr halten sich für wichtig und wollen gefragt werden – mit der Konsequenz, dass das System immer unflexibler und träger wird!) Und auch die Art der laufenden Tasks spielt eine Rolle. Wenn alle Tasks gleiche Bedeutung haben, sind die Zeitscheiben für jeden Task gleich groß. Das muss aber nicht notwendigerweise so sein. So bräuchte beispielsweise ein Bildschirmschoner in seiner Zeitscheibe lediglich festzustellen, ob Aktivitäten vorhanden sind. Ist dies der Fall, muss er nicht aktiv werden und kann in den Hintergrund treten. Oder Druckaufgaben: Da der Drucker erheblich

langsamer ist als die CPU, braucht ein im Hintergrund arbeitender Druck-Task lange nicht die Rechenleistung, die z.B. ein wissenschaftliches Programm benötigt, das Messwerte eventuell sogar in »Echtzeit« auszuwerten hat.

Task

Um nun zwischen Tasks umschalten zu können, müssen alle relevanten Daten, die mit einem Task verbunden sind, gesichert (aktueller Task) bzw. restauriert (neuer Task) werden. Zu diesem Zweck kommen *task state segments* zum Einsatz, die aufgrund ihres Typs (Systemsegment) bereits auf Seite 420 ausführlich beschrieben wurden. Jedem Task ist ein solches task state segment zugeordnet, das in der *global descriptor table* (GDT) verzeichnet sein muss. Es enthält die Inhalte aller Register der CPU sowie verschiedene Felder, die die notwendigen Informationen aufnehmen (vgl. Abbildung 2.8 auf Seite 420). Dies sind zum einen die Selektoren für das Code- und die verschiedenen möglichen Datensegmente (DS, ES, FS, GS) sowie für das Stacksegment (SS) und, falls der task im protected mode abläuft, für die Stacksegmente in den verbleibenden drei übergeordneten Privilegstufen. Zum anderen finden sich Angaben zu einer eventuell vorhandenen, taskeigenen local descriptor table (LDT) sowie, falls der Paging-Mechanismus aktiv ist, auch die Basisadresse der *page directory table*.

Daneben besteht ein Task natürlich auch aus dem Adressraum, in dem die zur Ausführung erforderlichen Teile angesiedelt sind (*task execution space*), und den dort anzusiedelnden Daten: Codesegment, mindestens ein Datensegment und mindestens ein Stacksegment.

Task Selector

Es mag merkwürdig erscheinen – aber all diese Informationen lassen sich mit einem einzelnen 16-Bit-Wert abrufen: dem *task selector*. Dieser Selektor zeigt in die *global descriptor table* und dort auf einen Deskriptor, der das *task state segment* beschreibt, in dem die Informationen zu Code-, Daten-, Stacksegment sowie den anderen Informationen liegen ... Dieser Selektor residiert in einem eigenen Register, dem task register (TR), das bereits auf Seite 432 vorgestellt wurde.

Mechanismus des Task Switch

Das bedeutet, ein task switch ist recht einfach zu bewerkstelligen: Eintrag des neuen, gewünschten task selector in den sichtbaren Teil des task register mittels des Befehls LTR, der als privilegierter Befehl allerdings nur aus Privilegstufen < 3 zugänglich ist, und Auslesen der entsprechenden Informationen. Fertig!

LTR wird jedoch lediglich unmittelbar nach dem Start des Prozessors und Umschalten in den protected mode »alleine« benutzt, um einen

»Initialtask« zu starten. Läuft dieser, erfolgt ein task switch etwas komplizierter auf eine von vier Arten:

- Ausführen eines Far-CALL oder Far-JMP mit dem Selektor auf den Deskriptoren des neuen TSS als Operand,
- Ausführen eines Far-CALL oder Far-JMP mit dem Selektor auf einen in der GDT oder aktuellen LDT verzeichneten Deskriptor für ein task gate,
- Ausführen eines Interrupts bzw. einer Exception mit einem Selektor, der auf ein task gate descriptor in der IDT zeigt, oder
- Ausführen eines IRET des aktuellen tasks, wenn das NT-Flag im EFlags-Register gesetzt ist und damit anzeigt, dass der aktuelle task durch einen »übergeordneten« aufgerufen wurde. Der notwendige Selektor steht dann im Feld PTL (previous task link) des aktuellen TSS.

Selektor für das neue TSS

Wird somit ein JMP, CALL oder IRET ausgeführt, so bestimmt der Prozessor zunächst den Selektor für das neue TSS. Bei einem JMP und CALL »in ein TSS« ist es der Selektor des als Operanden übergebenen Far-Pointers. Bei einem Interrupt, JMP oder CALL »in ein task gate« ist es entweder der Selektor, der als Interruptvektor dem INT-Befehl übergeben wird, oder derjenige, der im Deskriptor steht, dessen Selektor als Teil des Far-Pointers dem JMP- oder CALL-Befehl als Operand übergeben wurde. Bei IRET ist es, wie gesagt, der Inhalt des Feldes PTL des aktuellen TSS.

Schutzkonzepte

Als Nächstes prüft der Prozessor im Rahmen des INTnn-, CALL- oder JMP-Befehls, ob der aktuelle task den switch überhaupt ausführen darf (Schutzkonzepte). Hierzu wird geprüft, ob der CPL des aktuellen tasks und der RPL des übergebenen Selektors kleiner oder gleich dem DPL des betreffenden Deskriptors für das TSS sind. Hardware-Interrupts und Exceptions dürfen einen task switch ebenso ungeprüft durchführen wie der IRET-Befehl.

Verfügbarkeit der Daten

Sind die Voraussetzungen erfüllt, prüft der Prozessor, ob das neue TSS »present« und sein Limit valide ist, was bedeutet, dass es größer als $67 Bytes sein muss. (Dies ist die Mindestgröße, die ein TSS benötigt, um alle taskrelevanten Daten außer einer eventuellen I/O permission bit map und weiteren, zusätzlichen Informationen zu speichern. Ist das nicht der Fall, wird eine #NP ausgelöst und dem Betriebssystem damit die Möglichkeit gegeben, das Segment nachzuladen.

Anschließend wird geprüft, ob der neue task selbst verfügbar ist (»present« bei CALL, JMP und INT nn bzw. »busy« bei IRET).

Ist das der Fall und ist der Paging-Mechanismus eingeschaltet, so wird geprüft, ob der aktuelle TSS, der neue TSS und alle am task switch beteiligten Deskriptoren im physikalischen Speicher vorliegen. Ist das nicht der Fall, wird mit einer #PF dem Betriebssystem die Möglichkeit gegeben, die entsprechenden pages zu laden.

Updaten des Deskriptors des aktuellen Tasks

Falls der task switch durch ein JMP oder IRET ausgelöst wird, löscht der Prozessor das busy flag B im Deskriptor des aktuellen tasks. Bei einem CALL, Interrupt oder einer Exception bleibt das Flag gesetzt. Wurde der switch durch ein IRET ausgelöst, wird das NT-Flag im EFlags-Register, genauer: einer temporären Kopie davon, gelöscht. Nach einem CALL, JMP oder Interrupt/Exception bleibt das NT-Flag unangetastet.

Sicherung der aktuellen Task-Umgebung

Nun sichert der Prozessor die aktuelle Task-Umgebung, bestehend aus den Inhalten der Allzweckregister der CPU, der Segmentregister, der temporären Kopie des EFlags-Registers sowie des instruction pointers (EIP) in die dafür vorgesehenen Felder des aktuellen TSS.

Updaten des Deskriptors des neuen Tasks

Falls der Task Switch durch ein CALL, eine Exception oder ein Interrupt ausgelöst wurde, wird im TSS des neuen Tasks das NT-Flag im EFlags-Register-Abbild gesetzt. Falls ein IRET der Initiator des Task Switch ist, wird das NT aus der auf dem Stack liegenden Kopie des EFlags-Registers restauriert. Bei einem Task Switch nach einem JMP bleibt NT unverändert.

Wurde der Task Switch durch ein CALL, ein JMP, einen Interrupt oder eine Exception ausgelöst, wird das busy flag im Deskriptor des neuen Tasks gesetzt. Nach einem IRET bleibt B gesetzt.

Einleitung des task switch

Nun wird das TS-Flag (task switched) in Kontrollregister CR0 gesetzt, um nach dem eigentlichen Switch dem Betriebssystem Gelegenheit zu geben, Verwaltungsaufgaben nach einem Task Switch durchzuführen (z.B. Sicherung der FPU- bzw. SIMD-Umgebung).

Jetzt – und erst jetzt! – wird mittels LTR der Selektor des neuen Tasks in das task register geschrieben.



Ab dieser Stelle »verpflichtet« sich der Prozessor zum Task Switch (»commitment to task switch«). Sind bis zu diesem Zeitpunkt Fehler aufgetreten, die nicht behandelt werden können (also keine #NP oder #PF!), so kann er alle Veränderungen rückgängig machen und somit auf den alten Task »zurückschalten«. Nach diesem Punkt ist das nicht mehr

möglich. In diesem Fall vollendet der Prozessor den Switch, ohne jedoch weitere Prüfungen (Schutzkonzepte, Verfügbarkeit der Segmente etc.) durchzuführen. Bevor er dann jedoch mit der Ausführung des neuen Tasks fortführt, löst er eine geeignete Exception aus. Es ist nun Aufgabe des Handlers der Exception, den Task Switch korrekt zu beenden (also alle Prüfungen durchzuführen), bevor er dem Prozessor die Ausführung des neuen Tasks erlaubt.

Vollendung des Task Switch

An dieser Stelle lädt der Prozessor die neue Task-Umgebung aus dem TSS des neuen, jetzt aktuellen Tasks. Dabei handelt es sich um die Inhalte des Kontrollregisters CR3 mit der neuen Basisadresse der page directory table, des LDTR mit der aktuellen local descriptor table, des EFlags- und Instruction-Pointer-Registers sowie der Segment- und Allzweckregister der CPU. Schließlich wird die Programmausführung an der neuen Stelle (CS:EIP) fortgesetzt – der Task Switch ist erfolgt.



Jeder Task hat einen eigenen Adressraum, in dem er abläuft, und ein eigenes TSS. Aufgrund der Tatsache, dass in diesem TSS auch ein Abbild des CS-Registers liegt, besitzt jeder Task auch einen eigenen CPL. Während eines Task Switch erfolgen ausführliche Prüfungen zur Rechtmäßigkeit des Switches. Die einzelnen Tasks sind somit streng voneinander isoliert und Software braucht nicht noch zusätzlich zu prüfen, ob die erforderlichen Privilegien vorliegen: Sie liegen vor, falls der Task Switch erfolgreich war.



Da Task Switching eine Funktion des Betriebssystems ist und im Rahmen dieses Buches, was das Betriebssystem betrifft, lediglich Hintergrundinformationen geliefert werden sollen, soll an dieser Stelle die Besprechung des Multitaskings beendet werden. Sie sollten nun genügend Erkenntnisse gewonnen haben, um zu verstehen, wie ein Task Switch abläuft und welcher Aufwand getrieben wird. Dieses Wissen sollte mehr als ausreichen, um Assembler in Ihren Hochsprachen einsetzen zu können. Falls Sie weitere Informationen benötigen, muss ich Sie auf Sekundärliteratur verweisen.

2.4 Schutzmechanismen

Der protected mode stellt einen Schutzmechanismus vor unberechtigtem Zugriff zur Verfügung. Kernpunkt des Schutzmechanismus ist die Definition von Ebenen, die mit unterschiedlichen Zugriffsattributen, den »Privilegien«, ausgestattet sind. Die Speichersegmentierung kennt vier solcher »Privilegstufen«, die mit den Nummern 0 bis 3 codiert wer-

den: Privilegstufe 0 stellt die höchste, Privilegstufe 3 die niedrigste Stufe dar. Der Paging-Mechanismus kennt zwei Privilegstufen.

Geschützt werden kann und muss »nur« der Zugriff auf Daten. Das bedeutet, dass die Schutzmechanismen nur dann eine Rolle spielen, wenn

- bei einem Befehl eine Adresse ins Spiel kommt oder
- ein Zugriff auf die Peripherie des Prozessors erfolgen soll, also seine »Ports« (vgl. Seite 827) involviert sind.

Im Falle der Adressberechnung gibt es somit die Möglichkeit, Schutzmechanismen im Rahmen der

- Speichersegmentierung und/oder
- des Paging-Mechanismus

zu implementieren. Beides erfolgt.

2.4.1 Schutzmechanismen im Rahmen der Speichersegmentierung

Wie in Kapitel »Segmenttypen, Gates und ihre Deskriptoren« auf Seite 407 beschrieben, gibt es für jedes Segment einen Deskriptor, der neben der Lage des Segmentes im virtuellen Adressraum des Prozessors und der Größe des Segmentes auch Felder enthält, die als Segmentattribute bezeichnet werden und der Unterstützung der Schutzkonzepte dienen.

Mit ihrer Hilfe sind verschiedene Prüfungen möglich, die im Rahmen der Verifizierung von Zugriffsbeschränkungen durchgeführt werden können:

- Prüfung, ob sich eine Adresse innerhalb des betreffenden Segments befindet (»limit checking«)
- Prüfung, ob der Segmenttyp bei einem Zugriff erlaubt ist (»type checking«)
- Überprüfung der Zugriffsrechte (»privileg level checking«)
- Beschränkung auf Einsprungpunkte
- Beschränkung des Befehlssatzes

In diesem Zusammenhang sind einige Begriffe zu sehen, die im Folgenden detaillierter dargestellt werden.

Privilegstufen

Die CPU unterstützt einen Schutzmechanismus, der auf vier Privilegstufen basiert. Die Stufe mit den weitestgehenden Privilegien, also mit der höchsten Privilegstufe, ist die Ebene 0.

Kernelmodus

Auf dieser Ebene sind alle Komponenten angesiedelt, die (fast) uneingeschränkten Zugang zu allem haben (müssen), was möglich ist. Diese Stufe ist naturgemäß die wesentliche Stufe und wird daher vom Betriebssystem, genauer: Teilen des Betriebssystems, dem Betriebssystemkern oder »kernel« benutzt. Code, der mit dieser Privilegstufe abläuft, läuft im »Kernelmodus«, wie man sagt.

Auf den beiden nächsten Ebenen, level 1 und 2, können Komponenten angesiedelt werden, die abgestuft einen ebenfalls hohen Zugriffsschutz benötigen, jedoch nicht notwendigerweise alle Privilegien haben müssen (sollen), die der Kernel hat. Intel schlägt hierfür Service-Routinen des Betriebssystems vor.



Tatsache jedoch ist, dass Microsoft weder in seinen Consumer-Versionen des Betriebssystems Windows (Windows 95, 98, 98SE, ME) noch in den Professional-Versionen (Windows NT, 2000, XP) diese beiden Privilegstufen nutzt: Das Betriebssystem teilt sich auf die Ebenen 0 und 3 auf! Grund: Windows ist bzw. soll ein Betriebssystem sein, das auf mehreren Plattformen läuft. Und da es neben Intel und den x86-Prozessoren auch andere Hersteller mit anderen Prozessoren gibt bzw. gab (z.B. Alpha), musste den unterschiedlichen Hardwarevoraussetzungen Rechnung getragen und der kleinste gemeinsame Nenner gefunden werden. Dieser aber ist, da einige der »Exoten«, die berücksichtigt werden soll(t)en, nur zwei Privilegstufen kennen, eben diese Zwei-Ebenen-Architektur. Schade eigentlich!

Usermodus

Die letzte Ebene, Privilegstufe 3, ist die Ebene mit den geringsten Zugriffsrechten und somit den höchsten Zugriffsbeschränkungen. Diese Ebene ist für Anwendungsprogramme oder Tools des Betriebssystems vorgesehen, die vom Kernel und ggf. auf anderen Ebenen angesiedelten Betriebssystemteilen abgeschottet werden sollen. Code, der mit dieser Privilegstufe abläuft, läuft im so genannten Usermodus ab.



Beachten Sie bitte, dass die »Privilegiertheit« numerisch umgekehrt zur Privilegstufe zu interpretieren ist: Je niedriger der numerische Wert, desto höher die attestierten Privilegien!

Diese Privilegstufen spielen in Verbindung mit den folgenden Feldern eine wesentliche Rolle, die CPU und ihre Schutzkonzepte zur Verfügung stellen:

CPL Einer der wesentlichsten Begriffe in Verbindung mit Schutzmechanismen ist der Begriff »current privileg level«, abgekürzt: CPL. Der CPL ist die Privilegstufe, mit der der augenblicklich ausgeführte Code ausgestattet ist. Er signalisiert, welche Zugriffe den aktuell bearbeiteten Instruktionen erlaubt sind und welche nicht.

Der CPL wird vom Betriebssystem nach verschiedenen Prüfungen vergeben, wann immer Code in einem »neuen« Segment ausgeführt werden soll. Dies ist nach einem Task Switch der Fall, aber auch nach einem Far-JMP oder Far-Call sowie im Rahmen von Interrupts. Der vom Betriebssystem festgelegte CPL wird in den Bits 0 und 1 des Segmentregisters CS gespeichert. Hier steht üblicherweise der RPL (s. u.) aus dem übergebenen Selektor; im Falle des CS-Registers ist dies aber identisch mit dem CPL.

DPL Wenn das Betriebssystem eine Privilegstufe vergibt, die die Zugriffsrechte des jeweils aktuell ausgeführten Codes betrifft, so muss es auch Privilegstufen für die Programmteile geben, auf die zugegriffen werden soll und die geschützt werden sollen. Diese Zugriffsanforderungen nennt man »descriptor privileg level«, kurz DPL. Der DPL wird für das gesamte Segment vergeben, das vom Deskriptor beschrieben wird. Er beschreibt die Privilegstufe, die zugreifender Code mindestens haben muss, damit der Zugriff gestattet wird.

Das bedeutet, dass allein durch diese beiden Felder bereits ein Zugriffsschutz realisiert werden kann und wird: Das Betriebssystem entscheidet, welche Privilegien zugreifender Code hat und welche Privilegien Segmente erfordern, auf die zugegriffen wird. Stimmen beide überein, wird der Zugriff erlaubt, andernfalls untersagt.

Der DPL wird je nach Segmenttyp etwas unterschiedlich interpretiert:

- In Datensegmenten non-conforming codes segments, auf die *nicht* über ein call gate zugegriffen wird, call gates und bei task state segments ist der DPL der numerisch höchste Wert, den ein Programm oder Task haben darf, um Zugriff zu bekommen. So muss z.B. Code, der auf ein solches Segment mit DPL = 1 zugreifen will, einen CPL ≤ 1 haben. Code mit CPL > 1 hat keine Chance!
- In conforming code segments oder non-conforming code segments, auf die via call gate zugegriffen wird, bezeichnet DPL den nume-

risch kleinsten Wert, den ein Programm oder Task haben darf, um auf das Segment zugreifen zu können. Das bedeutet, dass bei Zugriffen auf ein solches Segment mit DPL = 2 Code keine Chance hat, dessen CPL < 2 ist. Nur Code mit CPL = 2 oder 3 hat die Erlaubnis.

RPL

Eine weitere Stufe in diesem Zusammenspiel ist der »requestor privileg level«, RPL. Er wird manchmal auch »request*ed* privileg level« bezeichnet und findet sich als Bits 0 und 1 im Selektor, der mit einem Segment in Verbindung steht. Er wurde kreiert, um die Zugriffsprivilegien des zugreifenden Codes an die des Segments anzupassen, auf das zugegriffen werden soll. Ursprünglich war der RPL dazu gedacht, im Rahmen von Routinen anzugeben, welchen CPL der Rufer der Routine, der »requestor« hat.

Der RPL kann dazu benutzt werden, den CPL »zu schwächen«. Ein Beispiel: Gegeben sei eine Systemroutine, die von einem Benutzerprogramm aufgerufen werden darf. Diese Routine sitzt in einem Segment, das als Systembestandteil die höchste Privilegstufe besitzt: DPL = 0. Wenn nun (z.B. über ein call gate) die Programmausführung vom Benutzerprogramm in dieses Segment übertragen wird, würde zwangsläufig ein CPL = 0 eingestellt. Auf diese Weise wäre es jedem nicht-privilegierten Programm oder Task möglich, die Schutzkonzepte dadurch zu untergraben, dass Systemroutinen aufgerufen werden.

RPL macht hier einen Strich durch die Rechnung. Da jedem aufgerufenen Programm in irgendeiner Weise eine Rücksprungadresse übergeben wird – sei es als tatsächliche Rücksprungadresse auf dem Stack bei einem CALL oder INT, sei es in Form eines Task-Selektors bei nested tasks –, hat aufgerufener Code Zugriff auf den Selektor, der auf das Codesegment zeigt, das gerufen hat. In diesem Selektor steht aber ein RPL, der identisch mit dem CPL des rufenden Codes ist und nicht manipuliert werden kann. (Bei einem Far-CALL wird der Inhalt des CS-Registers auf den Stack gelegt. Und Bit 0 und 1 dieses Selektors sind der CPL des rufenden Codes, der nun lediglich RPL heißt.)

Das gerufene Programm nun kann den eigenen CPL anhand dieses RPLs anpassen (siehe ARPL auf Seite 174). Auf diese Weise läuft es mit den Privilegien, die der rufende Teil hat, selbst wenn es sich ursprünglich um ein höherprivilegiertes Segment gehandelt haben sollte. Ergebnis: Das Schutzkonzept greift weiterhin.

Limit Checking

Jedes Segment hat in seinem Deskriptor das Feld *segment limit*, das die Größe des Segments angibt. Mit diesem Feld ist es möglich, zu prüfen,

ob sich ein Offset innerhalb der Grenzen des Segments befindet oder nicht. Auf diese Weise werden beabsichtigte oder unbeabsichtigte Fehler entdeckt, die darauf beruhen, dass auf Adressen außerhalb des aktuellen Segments zugegriffen wird. Limit checking dient somit der physischen Isolierung einzelner Segmente voneinander.

Die Nutzung des Feldes ist allerdings nicht so einfach, wie das zunächst den Anschein hat: *IF Offset > SegmentLimit THEN Fehler* funktioniert *so* nicht! Denn der 20-Bit-Inhalt dieses Feldes muss anhand einiger Flags der Segmentattribute »interpretiert« werden. Bei den zu berücksichtigenden Feldern handelt es sich um

- das *granularity flag* G. Ist es gesetzt, so bezieht sich der Inhalt von segment limit auf die Anzahl von 4-kByte-Pages, was bedeutet, dass er mit 2^{12} = 4.096, der Größe einer page, multipliziert werden muss, um die tatsächliche Größe des Segments in Bytes zu erhalten. Auf diese Weise kann das Segment die maximale Größe im virtuellen Adressraum annehmen (2^{20} (segment limit) · 2^{12} (page size) = 2^{32} = 4 GByte). Ist es dagegen gelöscht, spiegelt segment limit die tatsächliche Größe in Bytes wider. Damit lassen sich Segmentgrößen bis 1 MByte (= 2^{20} Byte) erreichen. (Hinweis: der Befehl LSL, vgl. Seite 176, hilft bei der Berechnung der tatsächlichen Segmentgröße.)
- Bei Datensegmenten spielt noch das Flag E, *expand down*, eine Rolle. Die Segmentgrößen-Berechnung erfolgt zwar wie eben geschildert, doch wird der so berechnete Wert für die Segmentgröße in Abhängigkeit des Flags E anders interpretiert. Ist es gelöscht, so handelt es sich um ein *expand-up segment* und alles ist »normal«: Die Segmentgröße bezeichnet den Offset des letzten adressierbaren Bytes des Segmentes. Ist es dagegen gesetzt, so handelt es sich um ein *expand-down segment* und die Segmentgröße bezeichnet den Offset des letzten Bytes des Segments, das *nicht adressiert werden darf*. Bei Expand-Up-Segmenten liegt somit der »gültige« Bereich zwischen Offset *0* und Offset *Segmentgröße*, bei Expand-Down-Segmenten zwischen Offset *Segmentgröße* und Offset ... ja welchem eigentlich?
- Der Maximalgröße des Segmentes, natürlich. Und die wird durch das Flag B, *big*, angegeben. Ist B gesetzt, haben Segmente, weil 32-bittig adressiert, eine Maximalgröße von 2^{32} Byte = 4 GByte, andernfalls aufgrund der 16-Bit-Adressierung von 2^{16} Byte = 64 kByte. Das bedeutet: Bei Expand-Down-Segmenten liegt der »gültige« Bereich, der durch das limit checking geprüft wird, zwischen Offset *Segment-Größe* und Offset *4.294.967.295* bzw. Offset *65.535*.

Durch limit checking werden Fehler entdeckt, die zu einem Überschreiben oder Auslesen von Bereichen führten, die nicht zum Segment gehören. Solche Fehler können vor allem bei der Berechnung von Adressen auftreten. Sie werden immer dann entdeckt, wenn sie auftreten (nämlich bei Zugriff auf die betreffende Adresse).

Limit checking wird auch bei Zugriffen auf Systemtabellen eingesetzt. So besitzen GDTR und IDTR einen 16-Bit-Limit, LDTR und TR gar einen 20-Bit-Limit, der aus dem jeweiligen korrespondierenden Deskriptor ausgelesen wird. Bei Zugriffen auf diese Systemtabellen wird geprüft, ob der Selektor auf einen Eintrag außerhalb der durch den Limit angegebenen Wert zeigt.



Type Checking

Ein *type checking* erfolgt, um das inkorrekte oder unbeabsichtigte Verwenden von Segmenten für Zwecke zu verhindern, für die sie nicht zuständig sind. Ein Codesegment beispielsweise enthält Code, in der Regel aber nicht Daten. Und wenn, dann nur lesbare Daten, wie z.B. in ROMs. Der schreibende Zugriff auf ein Codesegment ist somit entweder ein Versehen oder, wenn beabsichtigt, nur aus ganz bestimmten, selten erforderlichen Gründen (»self modifying code«) erwünscht.

Zuständig für die Typprüfung sind das Flag S und das Feld Type im Deskriptor des Segmentes. S gibt an, ob das Segment ein Systemsegment ist oder aber ein Code- bzw. Datensegment. Um was für Untertypen bei System-, Code- und Datensegmenten es sich handelt, codiert das Feld Type (vgl. »Segmenttypen, Gates und ihre Deskriptoren« auf Seite 407ff).

Ein type checking erfolgt zu verschiedenen Zeiten und aus verschiedenen Anlässen, so z.B. wenn:

- ein Segment-Selektor in ein Segmentregister geladen wird. So dürfen in das Register CS nur Code- und in die Register DS, ES, FS und GS nur Datensegmente geladen werden. Das Segmentregister SS fordert noch einige Restriktionen, die bei Stackregistern notwendig sind (z.B.: beschreibbar!)
- ein Segment-Selektor in das local descriptor table register (LDTR) oder das task register (TR) geladen wird. In das LDTR dürfen nur Selektoren auf Systemsegmente vom (Unter-)Typ LDT-Segment geladen werden, in das TR nur solche für Systemsegmente vom Typ TSS (task state segment).
- Befehle auf Segmente zugreifen, deren Deskriptoren bereits in den nicht sichtbaren Teil des Segmentregisters geladen sind. So dürfen z.B. schreibende Instruktionen (z.B. MOV) nicht in read-only (data)

segments oder execution-only (code) segments schreiben. Analoges gilt für lesende Instruktionen (z.B. MOV) und execution-only (code) segments.

- der Operand einer Instruktion einen Selektoren für ein Segment enthält. Hier wären zu nennen: CALL und JMP, die nur Zugriff auf Codesegmente, call oder task gates oder TSS haben, INT, der nur Zugriffe auf interrupt, trap oder task gates hat, oder »spezialisierte« Instruktionen wie LLDT, LTR, LAR und LSL, die ebenfalls nur auf bestimmte Segmente zugreifen dürfen.
- verschiedene interne Operationen erfolgen, wie z.B. im Rahmen von CALLs, JMPs, INTs und IRETs.



Der Versuch, einen »Nullselektor« (vgl. Seite 430) in das CS- oder SS-Register zu laden, endet in einer general protection exception #GP. Ein solcher Selektor kann zwar ohne Reue in eines der Datenregister DS, ES, FS oder GS geladen werden. Allerdings führt dann jeglicher Zugriff auf Adressen, die das entsprechende Segmentregister explizit oder implizit einbeziehen, unweigerlich zu einer general protection exception #GP.

Privilege Level Checking

Die Überprüfung der weiter oben eingeführten »Privilegstufen« erfolgt immer dann, wenn der Selektor eines Segmentes in »sein« Register eingetragen wird, also entweder in die Segmentregister CS, DS, ES, FS, GS und SS oder in die Systemregister LDTR und TR. Je nach Typ des zum Einsatz kommenden Segments unterscheidet man hierbei die Prüfung der Privilegstufen bei

- Zugriff auf Datensegmente
- Zugriff auf das Stack-Segment und
- Zugriff auf das Code-Segment.

Datensegmente

Um Daten in einem Datensegment ansprechen zu können, muss der zum Datensegment gehörende Segment-Selektor in eines der Daten-Segmentregister DS, ES, FS oder GS geladen werden. Dies kann mit den Instruktionen MOV, POP, LDS, LES, LFS oder LGS erfolgen. Bevor der Prozessor diesen Selektor jedoch in das Segmentregister lädt, führt er eine Privileg-Prüfung durch. Hierzu vergleicht er zunächst den CPL des aktuellen Codesegments (in Register CS) mit dem RPL, der im Selektor des Datensegments verzeichnet ist. Das Codesegment kommt deshalb ins Spiel, da ja der Datenzugriff durch eine Instruktion erfolgt. Und diese Instruktion wird im Rahmen Code ausgeführt, der bestimmte Privilegien hat, die durch den CPL repräsentiert werden.

Von diesen beiden Werten, CPL bzw. RPL, wählt er den höheren Wert, gleichbedeutend mit niedrigerer Privilegstufe. Er lädt nun den Selektor dann und nur dann in das Segmentregister, wenn der DPL, also die für einen Zugriff geforderten Privilegien, einen numerisch höheren oder gleichen Wert hat als/wie der zu prüfende PL (CPL oder RPL). Anders ausgedrückt: Nur dann, wenn das Datensegment weniger oder die gleichen Privilegien fordert (gleicher oder höherer DPL), die das aktuelle Codesegment bzw. der Requestor haben, wird der Zugriff gestattet und das Datensegmentregister mit dem Selektoren beladen. Andernfalls wird eine general protection exception #GP ausgelöst.

```
CPL := CS[CPL]
RPL := Selector[RPL]
DPL := Descriptor[DPL]
IF (CPL < RPL) THEN
   PL := RPL
ELSE
   PL := CPL
IF (PL <= DPL) THEN
   SegmentRegister ← Selektor
ELSE
   #GP
```



Was heißt das? Schlicht und einfach: Code im Kernelmodus (CPL = 0) hat, so es der RPL zulässt (was man steuern kann, wie man im nächsten Absatz sehen wird), grundsätzlich Zugriff auf alle Datensegmente, unabhängig von den geforderten Privilegien. Code im Usermodus (CPL = 3) dagegen kann nur auf Datensegmente zugreifen, die einen DPL von 3 haben. Systemdaten mit DPL < 3 sind somit vor dem Zugriff durch Anwender geschützt.



Wichtiger Hinweis: Der RPL eines Datensegments ist unter Ihrer Kontrolle! Es hindert Sie niemand daran, das RPL-Feld im Selektore, den Sie in ein Segmentregister laden wollen, vorher auf »0«, also höchste Privilegstufe, zu setzen. Es erfolgt dann lediglich die Prüfung CPL – DPL. (Was dann in der Regel immer noch bedeutet, dass Sie keinen Zugriff haben, weil »Ihr CPL nicht stimmt«. Dennoch ist der RPL nicht sinnlos! Wann immer nämlich einem Codefragment (z.B. einer Routine) ein Selektor auf ein Datensegment übergeben wird (z.B. über einen Far-Pointer im Rahmen von Parameterübergaben), kann dieses Codefragment den RPL so setzen, dass Datensegmente vor einem Zugriff geschützt sind, selbst wenn der Zugriff aus einem Codesegment mit Privilegstufe

0 erfolgen sollte. So können Sie z.B. Ihre Datensegmente vor Zugriff durch das Betriebssystem (oder Debugger?) schützen! Unterstützt werden Sie dabei durch den Befehl ARPL (vgl. Seite 174).

Daten in Codesegmenten

Es kann vorkommen, dass Daten in Codesegmenten gespeichert werden. Häufig ist dies der Fall, wenn Code in ROMs abgelegt wird. Anschauliches Beispiel sind die BIOS-Funktionen im EPROM. Sie greifen ebenfalls auf Daten zurück, die eben auch im EPROM gespeichert sind.

Nun ist ein Zugriff auf ein Codesegment etwas anderes als einer auf ein Datensegment. Datensegmente haben ein Attribut, das sie zwar vor schreibendem Zugriffen schützt, niemals aber vor lesendem (wozu sollte ansonsten ein Datensegment auch gut sein?). Codesegmente dagegen sind nicht dafür gedacht, Daten aufzunehmen, weshalb sie in der Regel das Attribut execute-only besitzen. Um jedoch Daten auch im Codesegment zumindest lesen zu können – die Legitimität wurde ja mit dem ROM-Beispiel eben dargelegt –, kann ein Codesegment auch das Attribut readable haben. Dann kann es ausgelesen werden. Beschrieben werden kann es allerdings nie! Ade du schöner selbst modifizierender Code aus den guten, alten DOS-Tagen :-(



Ein Tipp von mir als bekennendem James-Bond-Freak: »Never say never«! Es gibt sehr wohl die Möglichkeit, auch schreibend auf Codesegmente zuzugreifen. Allerdings nur mit dem Trick, dem Codesegment das Hemd eines Datensegments überzustreifen. Soll heißen: Basteln Sie den Deskriptor für ein Datensegment mit den Eckdaten (Basisadresse, Limit) des Codesegments, tragen Sie diesen Deskriptor in die Deskriptoren-Tabelle ein und nutzen Sie den neuen Selektor. Einfach ist dies allerdings nicht, fehlen Ihnen doch vermutlich die erforderlichen Privilegien ...

Wie nun kann ein Zugriff auf Daten im Codesegment erfolgen? Das kommt darauf an, von welchem Typ das Codesegment ist:

- non-conforming, readable code segment: Laden Sie einfach den Selektor auf dieses Segment in ein Datensegment-Register. Dann haben Sie die gleichen Prüfmechanismen wie bei »normalen« Datensegmenten.
- conforming, readable code segment: Laden Sie ebenfalls den Selektor auf das segment in das gewünschte Datensegment-Register. Zugriffe auf ein solches Codesegment sind immer erlaubt, da diese

Segmente unabhängig von dem im DPL-Feld eingetragenen Wert immer die gleichen Privilegien haben wie der aktuelle CPL.

- readable code segment, dessen Selektor sich bereits im CS-Register befindet: Benutzen Sie den segment override prefix CS:, um auf die Daten »ganz normal« zuzugreifen. Denn der DPL dieses Codesegments ist der gleiche wie der aktuelle CPL (ansonsten wäre der Selektor nicht in das CS-Register geladen worden!).

Stacksegmente

Stacksegmente sind »besondere« Datensegmente. Bei solchen Segmenten muss CPL, RPL und DPL identisch sein, damit Sie auf die Daten auf dem Stack zugreifen können. Falls RPL oder DPL nicht gleich dem CPL sind, wird eine general protection exception #GP ausgelöst.

Codesegmente

Der Zugriff auf ein Codesegment kann, mit der eben besprochenen Ausnahme des Zugriffs auf Daten im Codesegment, nur dadurch erfolgen, dass die Programmkontrolle einem anderen Segment übertragen wird und sich somit CS und (E)IP ändern. Daher findet eine Privilegprüfung statt, wann immer ein neuer Selektor in das CS-Register eingetragen werden soll. Das ist bei allen CALLs und JMPs der Fall, bei denen ein vollständiger 48-Bit-Pointer bestehend aus 16-Bit-Selektor und 32-Bit-Offset übergeben wird (bei 16-Bit-Systemen: 32-Bit-Pointer bestehend aus 16-Bit-Selektor und 16-Bit-Offset), der noch interpretiert werden muss (s. u.). Das ist auch bei INT nn der Fall, wo ein Index in die IDT angegeben wird, aus deren indiziertem Deskriptor der Selektor für das zu verwendende Codesegment geholt wird. Und das ist bei RET und IRET der Fall, wo der Selektor auf dem Stack liegt, sowie bei SYSENTER und SYSEXIT, wo die Selektoren in Registern übergeben werden.

Bei CALLs und JMPs sind folgende Möglichkeiten gegeben:

- Direkte Übergabe des Selektoren des Zielsegments
- Indirekte Übergabe des Zielsegments durch Verwendung eines Selektors für ein call gate, das seinerseits den Selektor für das Zielsegment enthält.
- Indirekte Übergabe des Zielsegments durch Verwendung eines Selektors für ein task state segment, das seinerseits den Selektor für das Zielsegment im Feld CS enthält.
- »Doppelt indirekte« Übergabe des Zielsegments durch Verwendung eines Selektors für ein task gate, das einen Selektor für ein TSS enthält, das seinerseits den Selektor für das Zielsegment im Feld CS enthält.



Die im Rahmen eines Task Switches (vorletzter und letzter Punkt) ablaufenden Privilegprüfungen werden im Kapitel »Multitasking« auf Seite 462ff. besprochen.

Direkte Zugriffe auf Code-Segmente

Bei CALLs und JMPs unterliegen natürlich nur die Far-Versionen einer Privilegprüfung, da die NEAR- und Short-Versionen ja innerhalb des Segmentes bleiben. Wir beim Zugriff auf Datensegmente auch, kommen hier die Felder CPL, RPL und DPL zum Einsatz. Darüber hinaus spielt auch das conforming flag C im Deskriptoren des Codesegments eine Rolle.

Non-conforming Code Segments

Falls C im Deskriptor des Zielsegments gelöscht ist, liegt ein non-conforming code segment vor. Bei solchen Segmenten spielt RPL eine geringere Rolle als bei Datensegmenten. Hier muss RPL lediglich kleiner oder gleich CPL sein, damit eine Prüfung durchgeführt wird. Ist es größer als der CPL, wird eine #GP ausgelöst. Der restliche Prüfmechanismus ist einfach: CPL muss gleich DPL sein:

```
CPL := CS[CPL]
RPL := Selctor[RPL]
DPL := Descriptor[DPL]
IF (RPL > CPL) THEN
   #GP
ELSEIF (CPL = DPL) THEN
   SegmentRegister CS ← Selektor
ELSE
   #GP
```



Das heißt aber: Wenn der Selektor nach einer erfolgreichen Prüfung der Privilegien in das CS-Register geladen wird, bleibt der CPL erhalten, der gerufene Code wird mit dem CPL des rufenden Codes ausgeführt! Und das heißt auch: direkte Far-CALLs oder -JMPs kann es nur innerhalb einer Privilegebene geben. Soll die Privilegebene gewechselt werden, hat das über gates zu erfolgen!

Conforming Code segments

Ist das Flag C im Ziel-Deskriptor gesetzt, liegt ein conforming code segment vor. Bei diesen Segmenten spielt das Feld RPL überhaupt keine Rolle und wird nicht in die Privilegprüfungen einbezogen.

Ein Zugriff auf ein conforming code segment ist nur Code möglich, der *weniger oder gleich* privilegiert ist als das Zielsegment! Das bedeutet:

```
CPL := CS[CPL]
DPL := Descriptor[DPL]
IF (CPL · DPL) THEN
   SegmentRegister CS · Selektor
ELSE
   #GP
```



Beim Zugriff auf ein conforming code segment bleibt der CPL des rufenden Codes erhalten. Das bedeutet, dass hier der einzige Fall gegeben ist, in dem der DPL wertmäßig verschieden vom CPL sein kann. Da sich der CPL nicht ändert, erfolgt auch kein stack switch!



Conforming codes segments sind in der Regel Segmente des Betriebssystems, die für den ausschließlichen Gebrauch durch Anwendungen im User-Modus gedacht sind, die zwar Betriebssystem-Unterstützung benötigen, nicht aber direkten Zugriff auf Betriebssystem-Module. So sind häufig Mathematik-Bibliotheken oder gewisse Exception-Handler in solchen Segmenten untergebracht. Dass in solchen Segmenten der CPL des rufenden Codes erhalten bleibt, verhindert, dass über conforming code segments vom User-Modus aus auf höher privilegierte Teile zugegriffen werden kann.

Beschränkung auf Einsprungpunkte

Soll zwischen Codesegmenten »umgeschaltet« werden, die auf unterschiedlichen Privilegstufen liegen, muss dies über gates erfolgen. Solche Zugriffe nennt man daher auch Interprivileg-CALLs bzw. -JMPs. Der Grund für die Existenz eines gate ist, dass z.B. einem niedriger privilegierten Code nicht wahlloser Zugriff (über beliebige Offsets als Teil des Far-Pointers) auf das gesamte Segment gegeben werden soll, sondern nur spezielle, »erlaubte« Einsprungpunkte (die im gate definiert sind). Hier kommt der Zugriff über ein call gate in Betracht. Solche Zugriffe nennt man indirekte Zugriffe auf Code-Segmente.

Der Zugriff via call gate unterscheidet sich von einem direkten Zugriff somit in dem Punkt, dass ein weiterer Deskriptor und ein weiterer Selektor ins Spiel kommen: die des call gates. Dem CALL- bzw. JMP-Befehl wird im Rahmen des Far-Pointers der Selektor des call gates übergeben. Der Offset dieses Pointers interessiert nicht, es ist ein Dummy-Offset, da der Ziel-Offset bei Zugriffen über gates immer im Deskriptor des gates verzeichnet ist (vgl. Abbildung 2.10 auf Seite 422).

Diese zusätzliche Stufe bedingt, dass nun fünf statt vier Felder in die Privilegprüfung einbezogen werden: der CPL, der RPL des Gate-Selektors, die beiden DPLs des Gate- und Ziel-Codesegment-Deskriptors sowie das C-Flag des Zielsegment-Deskriptors. Der RPL des Selektors, der im call gate verzeichnet ist, interessiert nicht! Ferner unterscheidet sich die Privilegprüfung bei CALLs und JMPs.

```
CPL   := CS[CPL]
RPL   := Gate-Selector[RPL]
G-DPL := Gate-Descriptor[DPL]
S-DPL := Code-Segment-Descriptor[DPL]
C     := Code-Segment-Descriptor[C]
IF (CPL > G-DPL) OR (RPL > G-DPL) THEN
   #GP
ELSE
   IF ((C = 1) AND (C-DPL > CPL))     ; conforming
   OR ((C = 0) AND (Instruction = CALL) AND (C-DPL > CPL))
   OR ((C = 0) AND (Instruction = JMP) AND (C-DPL ≠ CPL) THEN
      #GP
   ELSE
      SegmentRegister CS ← Selektor
```

Erste Hürde: call gate. Ein Zugriff auf das call gate ist nur dann gestattet, wenn RPL und CPL kleiner oder gleich dem DPL des call gates sind. Das bedeutet konkret, dass call gates, die aus dem User-Modus (CPL = 3) angesprungen werden sollen, einen DPL von 3 haben müssen! Zweite Hürde: Code-Segment. Hier spielt der RPL keine Rolle mehr. An dieser Hürde trennen sich die Wege in zweifacher Hinsicht: beim Zugriff auf conforming code segments haben, wie bei direktem Zugriff, nur Segmente Erfolg, deren Privilegstufe (CPL) größer oder kleiner als die Privilegien des Codesegments (DPL) sind. Ist das nicht der Fall, scheitert der Zugriff. CALLs und JMPs machen hier keinen Unterschied. Bei non-conforming code segments dagegen gibt es einen Unterschied zwischen CALLs und JMPs. CALLs werden so interpretiert, dass eine Routine aufgerufen wird. Das bedeutet, sie können nur von Privilegstufen mit niedrigerem oder gleichem numerischen Wert (höher oder gleich privilegiert!) aufgerufen werden. JMPs dagegen werden so interpretiert, dass der Code »auf gleicher Stufe« fortgeführt wird. CPL und DPL müssen dann gleich sein.



Wie beim direkten Zugriff auf ein conforming code segment bleibt der CPL des rufenden Codes auch beim Inter-Privileg-Zugriff erhalten. Beim Inter-Privileg-Zugriff auf ein non-conforming segment wird der CPL auf den DPL des Zielsegmentes gesetzt.

Beschränkung der Instruktionen

Schließlich ist Teil des Schutzmechanismus die beschränkte Benutzung von Instruktionen. Diese »privilegierten« Instruktionen werden nur dann ausgeführt, wenn der ausführende Code Privilegstufe 0 hat.

Bei diesen Instruktionen handelt es sich um LGDT, LLDT, LTR, LIDT, MOV (mit Kontroll- und Debugregistern als Operanden), LMSW, CLTS, INVD, WBINVD, INVLPG, HLT, RDMSR, WRMSR, RDMPC und RDTSC-Schutzmechanismen im Rahmen des Paging-Mechanismus.

Eine zweite Möglichkeit, Schutzmechanismen zu benutzen, findet sich auf der Ebene des Pagings. Diese Mechanismen sind unabhängig von denen der Speichersegmentierung, komplementieren sie jedoch hervorragend.



Hierbei ist wichtig, festzuhalten, dass Schutzkonzepte im Rahmen der Speichersegmentierung gemäß der Reihenfolge der Adressberechnung *vor* den Schutzkonzepten beim Paging greifen. Das bedeutet, dass erst gar nicht die Ebene des Pagings mit der entsprechenden Zugriffskontrolle erreicht wird, falls sich bereits auf der Segmentierungsebene eine Zugriffsverletzung ergibt. Somit kann beispielsweise nicht das generelle Schreibverbot auf Code-Segmente dadurch umgangen werden, dass das R/W-Flag der Page, die das Segment enthält, auf writeable gesetzt und schreibend angesprochen wird. Das bedeutet also, dass eine Zugriffsbeschränkung auf Paging-Ebene immer nur als zusätzliche Maßnahme aufgefasst werden kann, die Mechanismen bei der Speichersegmentierung zu »verfeinern« und zu erweitern.

Auf Paging-Ebene kommen zwei Prüfungen zum Tragen:

- Prüfung des Page-Typs
- Prüfung der Privilegien

Fällt eine der beiden Prüfungen negativ aus, so führt das in jedem Fall zu einer page fault exception #PF.

Prüfung der Privilegien

Die Prüfung der Privilegien ist durchaus mit derjenigen vergleichbar, die bei der Speichersegmentierung eingesetzt wird. Es gibt zwei Betriebsmodi:

- Supervisor mode; der Prozessor ist in diesem Modus, sobald er mit einem CPL von 0, 1 oder 2 läuft.
- User mode; dieser Modus liegt vor, wenn der Prozessor mit CPL = 3 arbeitet.

Zur Angabe von Schutzattributen gibt es in den page directory entries und page table entries ein Flag, das U/S-Flag (Bit 2) (vgl. »Paging: Von der virtuellen zur physikalischen Adresse« auf Seite 441ff.), das darüber entscheidet, welcher Modus erforderlich ist, um auf eine Page zugreifen zu können. Ist das U/S-Flag gesetzt (User-Mode-Zugriffe erlaubt), so kann die betreffende Page im user mode angesprochen werden. Ist es dagegen gelöscht (Supervisor-Mode-Beschränkung), so kann ein Zugriff nur im Rahmen des supervisor mode erfolgen.

Prüfung des Page-Typs

Ein zusätzlicher Schutzmechanismus ist die Prüfung des Page-Typs. Sie basiert auf der Tatsache, dass es zwei Arten von Pages gibt:

- Pages, die für einen schreibenden und lesenden Zugriff gedacht sind (read/write access), und
- Pages, die lediglich für einen lesenden Zugriff vorgesehen sind (read-only access).

Gemäß diesen beiden Möglichkeiten gibt es analog dem U/S-Flag ein Flag, das die Unterscheidung ermöglicht. Dieses R/W-Flag findet sich ebenfalls sowohl in den page table entries der page tables als Bit 1 als auch in den page directory entries der page directory ebenfalls als Bit 1 (vgl. »Paging: Von der virtuellen zur physikalischen Adresse« auf Seite 441ff.). Auf diese Weise kann die Page geschützt werden, die die page table enthält wie auch die betreffende Seite, auf die der page table entry zeigt.



Einfluss auf das Geschehen hat auch das mit dem 80486 eingeführte WP-Flag (write protect) im Kontrollregister CR0. Es wurde eingeführt, um die »copy-on-write«-Strategie mancher Betriebssysteme (z.B. Unix) im Rahmen der Erzeugung von tasks zu unterstützen. Wird das WP-Flag gesetzt, so wird dem Prozessor der schreibende Zugriff auf User-Mode-Pages untersagt, wenn er sich im supervisor mode befindet. Versucht er es dennoch, kann das Betriebssystem im Rahmen der dann ausgelösten #PF eine Kopie der geschützten Seite herstellen.

Nach einem Prozessor-Reset ist WP gelöscht. Befindet sich nun der Prozessor im supervisor mode (CPL < 3), so hat er uneingeschränkten Zugriff auf alle pages, unabhängig davon, ob das U/S- und/oder R/W-Attribut gesetzt ist. Im user mode (CPL = 3) dagegen kann er ungehindert nur auf Pages zugreifen, in denen das U/S-Flag gesetzt ist und die betreffende Seite dadurch für einen Zugriff auch im user mode freigibt. Ist dann R/W-Flag gesetzt und die Page damit als read/write access-

able ausgewiesen, so ist auch ein schreibender Zugriff erlaubt, andernfalls kann die Seite nur gelesen werden.

Kombinierte Effekte

Wie Sie gesehen haben, haben sowohl page directory entries als auch page table entries je ein U/S- und R/W-Flag. Das bedeutet, dass auf beiden Ebenen der Adressumsetzung geprüft wird. So ist es vollkommen unerheblich, welche Position das R/W- und/oder U/S-Flag eines page table entries besitzt, sobald bereits auf der Ebene der page tables eine Verletzung der Zugriffsrechte festgestellt wird. Falls also bereits der Zugriff auf die page table untersagt wird, läuft gar nichts mehr.



Und noch ein Hinweis: Der Zugriff auf die Systemtabellen GDT, LDT und IDT erfolgt auf Paging-Ebene unabhängig vom CPL grundsätzlich in der Weise, als handele es sich um einen Zugriff im supervisor mode. Analoges gilt für Zugriffe auf den für eine bestimmte Privilegstufe reservierten Stack während eines Inter-Privileg-CALLs oder einem Interrupt oder einer Exception mit Privilegstufen-Wechsel.

2.4.2 Schutzmechanismen bei Zugriff auf die Peripherie

Der Zugriffsschutz auf Ports (vgl. »Ports« auf Seite 827) erfolgt über zwei Mechanismen:

- generelle Zugriffskontrolle auf alle Ports durch das Feld IOPL im EFlags-Register und
- individuelle Zugriffskontrolle auf einzelne Ports durch die *I/O permission bit map* im *task state segment* des aktuellen Tasks.

IOPL

Das Feld IOPL (»I/O privileg level«) im EFlags-Register der CPU kann dazu benutzt werden, die Privilegstufe zu definieren, die ablaufender Code mindestens haben muss, um Zugriff auf den gesamten I/O-Adressraum zu haben. Die CPU unterstützt dies, indem die Befehle, mit denen ein solcher Zugriff möglich ist, ihre Funktion verweigern, falls die Zugriffsprüfung fehlschlägt, und eine Exception auslösen. Zu diesen Funktionen gehören IN, INS (mit seinen Vettern INSB, INSW und INSD), OUT, OUTS (OUTSB, OUTSW, OUTSD), CLI und STI. (Diese Befehle nennt man auch IOPL-sensibel, da sie sensibel für das IOPL-Feld sind.)

Wie Sie bereits aus dem vorangehenden Kapitel wissen, ist die Privilegstufe, mit der jeder Code in jedem beliebigen Codesegment zum je-

weiligen Zeitpunkt läuft, im Feld CPL (current privileg level) im »verborgenen« Teil des Codesegmentregisters CS (und, redundant, im Stacksegment-Register SS) eingetragen. Die Erlaubnis zum Zugriff auf I/O-Adressen kann daher sehr einfach überprüft werden:

- Wenn CPL ≤ IOPL, besitzt der Code eine höhere oder gleiche Privilegstufe wie gefordert, der Zugriff wird gestattet und die Befehle arbeiten wie erwartet.
- Wenn CPL > IOPL, besitzt der Code nicht die erforderlichen Privilegien, der Zugriff wird verweigert und eine general protection exception #GP ausgelöst.



Jeder Task hat in seinem task state segment ein Feld für den Inhalt des EFlags-Register. Das bedeutet, dass jeder Task »sein eigenes« IOPL-Feld besitzt. Auf diese Weise ist es dem Betriebssystem möglich, Port-Zugriffe Task-abhängig zu ermöglichen oder zu verbieten.

Wer an Schutzkonzepte denkt, fragt sich natürlich gleich auch, wie man sie umgehen kann (entweder um sie zu umgehen oder um zu prüfen, wie man ggf. ein Umgehen verhindern kann!). Das Feld IOPL ist Teil des EFlags-Registers und damit, anders als andere Register wie die Kontrollregister, für jedermann frei zugänglich. Meint man. Somit bliebe es ja jedem freigestellt, das IOPL-Feld so zu modifizieren, dass ein Zugriff erlaubt ist: Man müsste ja nur den aktuellen CPL in das IOPL-Feld kopieren.

Irrtum! Dies ist nur mit wenigen Befehlen möglich, nämlich mit POPF und IRET, die ja in das EFlags-Register zurückschreiben. Diese beiden Befehle jedoch gehören zu den »privilegierten« Befehlen. Das sind Befehle, die eine Privilegstufe »0« benötigen, um ihre Funktion (vollständig) zu entfalten. Somit braucht man einen CPL von 0, um das IOPL-Feld verändern zu können. Und den gibt das Betriebssystem Ihnen in der Regel nicht!



Obwohl POPF und IRET privilegierte Befehle sind, können Sie mit ihnen arbeiten. Denn wie Sie in Band 2, Die Assembler-Referenz, bei der Besprechung der beiden Befehle sehen können, sind nur bestimmte Aktionen im Rahmen ihrer Tätigkeit privilegiert, so z.B. das Verändern des IOPL-Feldes, nicht aber z.B. des Kontrollflags oder der Statusflags.

Der Versuch, das IOPL-Feld zu verändern, ist nicht »strafbar«. Das bedeutet, es wird nicht »zur Strafe« eine Exception ausgelöst, wenn man mittels IRET oder POPF eine Änderung herbeiführen will. Der Feldinhalt bleibt schlichtweg unverändert.



I/O permission bit map

Alternativ zum Schutzmechanismus via IOPL, der generell *alle* Ports sperrt oder freigibt, gibt es auch die Möglichkeit, Ports individuell vor einem Zugriff zu schützen. Diese individuelle Prüfung erfolgt immer dann, wenn der CPL größer als der IOPL ist oder man sich im virtual 8086 mode befindet. In diesem Fall konsultiert die CPU die *I/O permission bit map*. Hierbei handelt es sich um ein Feld von Bits, das jeder I/O-Adresse genau ein Bit zuordnet. Da es max. 65.536 I/O-Adressen gibt, kann diese Bitmap auch max. 8.192 Byte groß sein. Meistens ist sie jedoch erheblich kleiner.

Ist nun das Bit, das zu der Port-Adresse gehört, gesetzt oder ist es gar überhaupt nicht vorhanden, da die I/O permission bit map kleiner ist als die Position des Ports in ihr es verlangt, wird der Zugriff verweigert und eine general protection exception #GP ausgelöst. Nur dann, wenn das Bit existent und gelöscht ist, wird der Zugriff erlaubt.



I/O-Adressen sind wie Speicheradressen Byte-orientiert. Das bedeutet, dass jede I/O-Adresse ein Byte im I/O-Adressraum anspricht. Nun gibt es analog Zugriffen auf »normalen« Speicher auch I/O-Zugriffe, die Word- oder DoubleWord-weise erfolgen. Das bedeutet, dass einem solchen Port zwei bzw. vier konsekutive Byte-Adressen zugeordnet sind – zumindest was die Zugriffsprüfung betrifft. Falls Sie somit via INSW einen Word-Zugriff auf eine Adresse $xxxx durchführen wollen (I/O-Adressen sind 16-Bit-Adressen!), müssen alle Bits existent und gelöscht sein, die dieser Port besitzt: Bit Nummer $xxxx und ($xxxx + 1). Bei einem DoubleWord-Zugriff auf Adresse $yyyy sind es die vier Bits $yyyy bis ($yyyy + 3). Ist nur eines der betroffenen Bits gesetzt oder nicht existent, wird gnadenlos der Zugriff verweigert und eine #GP ausgelöst.

Wo nun befindet sich diese I/O permission bitmap? Und: Kann man sie manipulieren? Sie befindet sich im task state segment (TSS; vgl. Seite 417). Und um die letzte Frage gleich zu beantworten: Ein TSS ist ein Systemsegment und damit dem schreibenden Zugriff und somit jeder Art von Manipulation entzogen, falls man nicht die Privilegien einer CPL = 0 hat!

Innerhalb des TSS liegt sie nicht an einer konstanten Stelle (vgl. Abbildung 2.8 auf Seite 420), sondern flexibel in Position und Größe irgendwo in diesem Segment. Das Feld *I/O map* am (konstanten!) Offset $66 ($102_d$) des Segmentes nennt den Offset dieser bit map innerhalb des Segments. Sie wird abgeschlossen durch ein Byte mit gesetzten Bits.



Falls also der im Feld *I/O map* stehende Wert größer als das oder gleich dem Segmentlimit des TSS ist, verfügt es über keine I/O permission bit map und alle Zugriffe auf Ports sind verboten – es sei denn, Ihr CPL ist kleiner oder gleich dem IOPL ...



Jeder Task hat in seinem task state segment eine eigene I/O permission bit map, zumindest aber die Möglichkeit dazu, eine zu realisieren. Das bedeutet, dass es analog IOPL dem Betriebssystem möglich ist, Port-Zugriffe auch via I/O permission bit map Task-abhängig zu ermöglichen oder zu verbieten.

2.5 Exceptions und Interrupts

Was sind Interrupts und worin besteht eigentlich der Unterschied zwischen Exceptions und Interrupts? Die letzte Frage ist einfach zu klären: In der Definition des Begriffes »Exception«. Denn technisch gesehen *sind* Exceptions Interrupts, wie alle anderen Interrupts auch! Sie heißen nur deshalb anders, weil die Ursache für den Interrupt in einem Fehler liegt, der bei der Bearbeitung eines CPU-Befehls erfolgte und damit zu einer *Ausnahmesituation*, einer *»exception«*, führte. Oder anders ausgedrückt: Interrupts erfolgen, soweit sie wie Exceptions von der Hardware ausgelöst werden, unvorhersehbar und ohne Beziehung zum ablaufenden Programm (»asynchron«), Exceptions vorhersehbar (wenn man akzeptiert, dass jeder Befehl auch zu Fehlern führen kann; und spätestens wenn Sie in Band 2, Die Assembler-Referenz, die Befehle genauer studieren, werden Sie wissen, dass Befehle immer zu Fehlern führen können!) und immer im Zusammenhang mit einem laufenden Programm (»synchron«). Behandelt werden Exceptions und Interrupts aber absolut identisch.

2.5.1 Interrupts

Als Interrupt bezeichnet man den Vorgang, dass die CPU von irgendeiner »Interruptquelle« ein Signal erhält, das sie den derzeitigen Programmablauf möglichst schnell unterbrechen soll, um sich dem Grund

für den Interrupt zu widmen. Daher auch der Name *Interrupt*: (Programm-)Unterbrechung.

Gründe für Interrupts gibt es viele: Eine Speicherstelle ist defekt; auf der seriellen Schnittstelle klopft ein Byte an; der Anwender glaubt, gerade in diesem Moment die Maus bewegen oder die Tastatur bearbeiten zu müssen; das gerade im Debugger ablaufende Programm kommt an einen Haltepunkt (»*break point*«). Je nach Quelle und Ursache unterscheidet man daher

- Hardware-Interrupts, bei denen eine Komponente des Rechners oder die CPU selbst den Interrupt auslöst, und
- Software-Interrupts, bei denen die Software Anlass für die Programmunterbrechung ist.

Hardware-Interrupts

Die Hardware-Interrupts können von verschiedenen Hardwarekomponenten des Systems ausgelöst werden. Hierbei gibt es zum Teil erhebliche Unterschiede, die sich auch in der Behandlung der Interrupts auswirken können. Gemäß der Quelle des Interrupts möchte ich drei Gruppen definieren:

- Nicht maskierbare Interrupts; solche »non-maskable interrupts« (NMIs) sind Interrupts, die sich, wie der Name bereits suggeriert, nicht »maskieren« lassen. Das bedeutet, es gibt keine Möglichkeit, sie »abzuschalten« und damit die Interrupt-Behandlung für diesen Fall auszuschalten. NMIs signalisieren in der Regel gravierende Fehler des Systems: Fehler im Speicher, in Kontrollern, in der Hardware insgesamt. Da solche Fehler in der Regel nicht (ohne technische Hilfe von außen) reparierbar sind, würden sie – unbehandelt – zu einem falschen Verhalten der laufenden Programme führen (z.B. könnten Daten nicht gespeichert werden) oder gar Schäden anrichten. NMIs sind daher in der Regel dazu da, dem Anwender eine Hiobsbotschaft zu überbringen – und den Prozessor dann herunterzufahren.

  Eine weitere Form der nicht maskierbaren Interrupts ist der SMI, *system management interrupt*, der entweder über eine direkte Verbindung zur CPU oder via APIC (*advanced programmable interrupt controller*) durch einen spezialisierten Baustein ausgelöst wird. Auf diese Interruptquelle gehe ich aber nicht weiter ein, da der SMM (*system management mode*), in dessen Dunstkreis der SMI anzusiedeln ist, nicht Gegenstand dieses Buches ist.

- Maskierbare Interrupts; das sind die Interrupts, die üblicherweise von Hardwarekomponenten generiert werden, die aus welchen Gründen auch immer, die Aufmerksamkeit des Prozessors benötigen. Hierzu gehören Tastatur, Maus, Festplattenkontroller, Timer, serielle Schnittstelle(n) und ggf. auch die parallele(n) etc. Diese Komponenten melden in der Regel ihre Interrupt-Anforderung (*interrupt request*; INTR) an einen darauf spezialisierten Baustein (externer oder interner PIC bzw. APIC), der die Reihenfolge der einlaufenden Anforderungen anhand ihrer Priorität sortiert und so an die CPU weitergibt. Solche Interruptquellen sind maskierbar, d.h. es lassen sich einzelne oder alle dieser Quellen »abschalten«: Die Komponenten haben dann zwar immer noch den Wunsch, dass die CPU sich ihrer annimmt. Wie ein(e) gute(r) Assistent(in) blockt jedoch der PIC (*programmable interrupt controller*) bzw. APIC (*advanced PIC*) die nicht erwünschten Anfragen ab, indem er sie nicht weiterleitet. Das Löschen des IF-Flag im EFlags-Register beispielsweise maskiert alle maskierbaren Interrupts, sodass nur noch NMIs und Softwareinterrupts erfolgreich sind. Sollen bestimmte Interruptquellen selektiv maskiert werden, muss via I/O der programmierbare Interruptkontroller direkt programmiert werden. Wenn Sie weitere Details hierzu benötigen, muss ich Sie auf Sekundärliteratur verweisen, die es haufenweise zu diesem Thema gibt.
- CPU-generierte Interrupts; schließlich kann auch die CPU selbst als Hardwarebaustein Interrupts auslösen. Sie tut das auch sehr ausgiebig. Die von der CPU ausgelösten Interrupts nun nennt man »*exceptions*«, da sie bis auf ganz wenige Ausnahmen aufgrund der bereits angesprochenen Ausnahmesituationen in einem regulären Programmablauf (Division durch Null, Verletzung der Schutzkonzepte, falsche Datenausrichtung, Debuggen, etc.) ausgelöst werden.

Software-Interrupts

Als Software-Interrupts bezeichnet man Interrupt-Anforderungen an die CPU, die nicht via PIC/APIC und INTR von Hardwarekomponenten bzw. von der CPU selbst kommen, sondern von dem laufenden Programm. Hierzu stellt der Prozessor den INT-Befehl zur Verfügung. Prinzipiell unterscheidet, bis auf die Quelle, Software-Interrupts nichts von Hardware-Interrupts – die Behandlung durch die CPU ist stets die gleiche: Einfrieren des derzeitigen Status, Aufruf des Handlers, der für die Bearbeitung des entsprechenden Interrupts zuständig ist, und Auftauen des eingefrorenen Programms, sobald der Handler seine Tätigkeit beendet hat.

2.5.2 Exceptions

Als Exceptions bezeichnet man also, wie gesagt, Interrupts, die die CPU aufgrund von Ausnahmesituationen beim regulären Programmablauf auslöst. Auch bei den Exceptions gibt es zwei Quellen:

- Hardware-Exceptions, wenn die CPU die Exception aufgrund eines Fehlers vor oder bei der Bearbeitung eines Befehles selbst auslöst, und
- Software-Exceptions, die durch bestimmte Befehle ausgelöst werden, wenn ein »Fehler« auftritt oder aufgetreten ist (z.B. löst INTO eine #OF [overflow exception] aus, wenn das overflow flag nach einer Operation gesetzt ist. Ohne das Einstreuen des INTO-Befehls in den Programmcode würde das Setzen des OF keine Exception auslösen!).

Die Software-Exception unterscheidet sich vom Software-Interrupt nur aufgrund der Randbedingung: Bei der Exception liegt ein zu behandelnder Fehler vor, beim Interrupt eben nicht – er wird benutzt, um z.B. bestimmte Systemfunktionen abzurufen.

2.5.3 Interrupt-Behandlung



Interrupts und Exceptions haben keine Namen. Sie werden anhand einer für sie reservierten, unveränderliche und eindeutigen Nummer identifiziert. Die Zuordnung der Interrupts und Exceptions zu »Ihren« Nummern entnehmen Sie bitte Tabelle 2.2 auf Seite 500.

Die Interrupt-Behandlung ist vom Ablauf her dieselbe, egal, ob die Hardware, Software oder CPU die Quelle war. In allen Fällen legt der Prozessor den Inhalt des EFlags-Registers auf den Stack und sichert somit den aktuellen Status (»Condition Code«) des Programms. Dann legt er eine Rücksprungadresse ebenfalls auf den Stack. Diese Adresse bezeichnet die Stelle, an der der Prozessor das unterbrochene Programm wieder aufnehmen soll, wenn der Interrupthandler seine Aufgabe erledigt hat.



Bei Exceptions legt die CPU ggf. zusätzlich einen Fehlercode auf den Stack. Dieser Code wird nach den Flags und der Rücksprungadresse abgelegt. Es liegt damit in der Verantwortung des Exception-Handlers, diesen Code vor dem abschließenden IRET-Befehl vom stack zu holen – IRET aus verständlichen Gründen nicht. Unterbleibt diese Stackbereini-

gung, verwendet IRET den Fehlercode als EIP-Anteil der Rücksprungadresse, den tatsächlichen EIP-Teil als Segment-Selektor und diesen als EFlags-Inhalt. Es ist offensichtlich, dass damit ernsthafte Probleme auftreten werden.

Das weitere Vorgehen hängt nun vom Betriebsmodus ab.

Real Mode

Im real mode existiert eine Tabelle, die auf den schönen Namen *interrupt vector table* (IVT) hört. Wie der Name bereits ahnen lässt, enthält diese Tabelle »Interruptvektoren«. Das sind Zeiger auf eine Routine, die den Interrupthandler darstellt. Der Zeiger besteht, wie im real mode üblich, aus einer Segmentadresse und einem Offset. Nach der im real mode üblichen Adressberechnung (real address = 16 · segment address + offset) berechnet die CPU nun die Adresse, an der der Handler für den betreffenden Interrupt steht, und lädt diese Adresse in CS:IP, was gleichbedeutend mit einem unbedingten Sprung an die Einstiegsadresse des Handlers ist. Dadurch wird der Handler aufgerufen. Dieser beendet seine Aktivitäten nicht mit einem üblichen RET-Befehl, sondern mit einem IRET. Der Unterschied zwischen beiden Befehlen ist der, dass IRET, nachdem es die Rücksprungadresse vom Stack gelesen und in CS:IP eingetragen hat, die ebenfalls auf dem Stack liegenden Flags holt und in das EFlags-Register zurück einträgt. Somit ist wieder der Zustand erreicht, der vor Auslösung des Interrupts herrschte, und die CPU kann mit dem unterbrochenen Programm weiterarbeiten, als wäre nichts passiert. RET dagegen lädt nur die Rücksprungadresse zurück. Der Ablauf der Interrupt-Behandlung im real mode ist in Abbildung 2.41 dargestellt.

Der Mechanismus der Interrupt-Behandlung setzt zwei Dinge voraus:

- Die CPU muss wissen, wo die Tabelle ist, und
- die Tabelle muss einen genau definierten, einheitlichen Aufbau haben.

Beide Anforderungen sind leicht zu erreichen: Die Tabelle liegt im Speicher grundsätzlich an Adresse $0000:$0000 (Segment: Offset), also bei $0_0000. Sie ist nicht verschiebbar! Sie kann maximal 256 Einträge aufnehmen, was bedeutet, dass im real mode maximal 256 Interrupts möglich sind. Um einen einheitlichen Aufbau und definierte Bedingungen zu schaffen, *muss* diese Tabelle auch 256 Einträge aufweisen. Werden nicht alle Interrupts benötigt, so werden an die Positionen der nicht benötigten Interruptvektoren die Adressen $0000:$0000 eingetragen. (Würde ein Interruptvektor mit einer solchen Adresse verwendet, pas-

sierten schlimme Dinge: An $0000:$0000 steht kein Interrupthandler, sondern – die Interrupt-Tabelle. Und die enthält mit Sicherheit keinen ausführbaren Code!

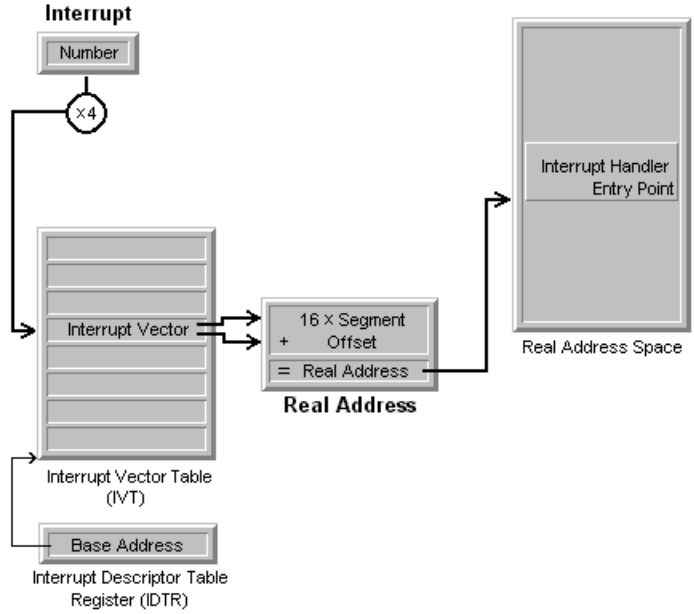


Abbildung 2.41: Berechnung der Adresse eines Interrupt-Handlers im real mode



So ganz stimmt die Behauptung, dass die Tabelle grundsätzlich an Adresse $0000:$0000 liegt und nicht verschiebbar ist, nicht! Spätestens seit dem 80286, der auch über ein IDTR (interrupt descriptor table register) verfügt, wird dieses Register Modus-unabhängig zur Feststellung der Basisadresse der IVT verwendet. Die CPU trägt nach einem Start oder Reset als Adresse $0000:$0000 und als Limit $03FF ein, sodass das eben Geschilderte korrekt ist. Danach kann auch im real mode der Inhalt des IDTR verändert werden, die CPU wird die IVT dann an entsprechender Stelle suchen. Dennoch ist es guter Stil, im real mode auf die Verschiebung der Tabelle im Speicher zu verzichten, da er praktisch nur noch zwecks DOS-Kompatibilität verwendet wird. In diesem Fall sollte auf 8086-Kompatibilität geachtet werden, auch wenn es diesen CPU-Dinosaurier schon lange nicht mehr gibt, weil unter DOS praktisch jeder am Betriebssystem vorbei direkt auf die BIOS- und sonstigen Strukturen zugegriffen hat – eben auch auf die IVT, um die Interrupts

auf eigene Routinen »zu verbiegen«. Und kein Mensch hat zu DOS-Zeiten an das IDTR gedacht!

Da die IVT Adressen des Typs Segment:Offset enthält, ist jeder Eintrag in die Tabelle vier Bytes (je zwei für die Words Segment und Offset) und die Gesamttabelle 1 KByte (= 256 · 4 Bytes) lang. Somit braucht die CPU zum Auffinden des korrekten Handlers nur die Interrupt-Nummer, die ihr der PIC/APIC übergibt (Hardware-Interrupts) oder die im INT-Befehl kodiert ist (Softwareinterrupts) bzw. die sich aus dem Typ des Fehlers ergibt (Exceptions) mit vier zu multiplizieren, um den Offset in die Tabelle zu erhalten. Diesen Offset zur Basisadresse $0_0000 der Tabelle addiert, liefert die Adresse des gewünschten Tabelleninhaltes. Dieser Adresse entnimmt sie dann Segment- und Offset-Anteil der Adresse des Handlers, der zum entsprechenden Interrupt gehört.

Protected Mode

Obwohl vom Grundsatz her vergleichbar, gestaltet sich die Auffindung des Interrupthandlers im protected mode etwas anders. In diesem Betriebsmodus haben wir ja Segmente und ihre Deskriptoren, in denen Adressen verwaltet werden. Die Adressberechnung zum Aufruf von Handlern im protected mode muss also in irgendeiner Weise mit Segmenten und ihren Deskriptoren umgehen.

Doch auch in diesem Fall haben wir eine Tabelle, sie heißt hier *interrupt descriptor table*, IDT. Auch diese Tabelle ist einheitlich aufgebaut, sodass es eine einfache Möglichkeit gibt, anhand der vom PIC/APIC (Hardware-Interrupts) übergebenen, im INT-Befehl codierten (Softwareinterrupts) oder sich aus dem Exception-Grund ergebenden Interrupt-Nummer einen Zeiger in diese Tabelle zu berechnen.

Die Tabelle besteht, wie ihr Name sagt, aus Deskriptoren und ist damit grundsätzlich so aufgebaut wie die GDT, die *global descriptor table*. Jeder Eintrag besteht aus zwei DoubleWords und somit 8 Bytes. Auch die IDT hat »nur« 256 Einträge und damit 256 mögliche Interruptquellen, was aber im Allgemeinen ausreicht! Die IDT ist damit 2 KByte groß (256 · 8 Byte). Anders als im real mode, wo die Tabelle (aus Gründen der Abwärtskompatibilität) grundsätzlich an Adresse $0_0000 liegen muss (sollte!), ist die Lage der IDT im Adressraum frei wählbar. Ihre Adresse wird analog zur GDT in ein spezielles Register der CPU eingetragen, dem IDTR (*interrupt descriptor table register*). Die Befehle LIDT und SIDT sind dafür verantwortlich, die Adresse der IDT in dieses Register zu schreiben oder aus ihm auszulesen.



Soweit die gute Nachricht. Und hier die schlechte: LIDT ist ein privilegierter Befehl, was bedeutet, dass er nur dann aufgerufen werden kann, wenn das Programm die Privilegstufe 0 und somit Betriebssystemfunktion hat. Das bedeutet, Sie werden vermutlich nicht an der IDT herumspielen können ...

Nicht alle Segmente sind Interrupt-tauglich. Segmente, die Interrupthandler beherbergen, müssen eine bestimmte Bedingung erfüllen: Die Handler müssen über *gates* anspringbar sein, also einen genau definierten Einsprungspunkt haben. Somit sind in der IDT Deskriptoren auf diese Art von Systemsegmenten verzeichnet:

- task gates
- trap gates
- interrupt gates

Die Behandlung von Interrupts erfolgt nun in der Weise, dass die CPU die Interrupt-Nummer mit 8 multipliziert, um den Selektor in die IDT zu erhalten. Dann entnimmt sie dem IDTR die Basisadresse der IDT, addiert den Selektor und entnimmt dem an dieser Adresse verzeichneten Deskriptor die erforderlichen Informationen. Im Prinzip ist somit der Ablauf eines protected mode interrupts der gleiche wie im real mode. Nur liegt eben im protected mode die Tabelle nicht an einer bestimmten Stelle, sondern muss erst via IDTR »gefunden« werden, und sie enthält nicht direkt die Adresse des Handlers, sondern einen Deskriptor, der die erforderlichen Informationen (Adresse) zum Anspringen des Handlers beinhaltet. Wenn dieser Weg auch komplizierter erscheint und ist, so ist er dennoch äußerst flexibel – und aufgrund der auch hier greifenden Schutzkonzepte sehr sicher.

interrupt bzw. trap gate

Findet die CPU im selektierten IDT-Eintrag einen Deskriptor für ein interrupt oder trap gate, muss ein Segment existieren, das den Handler beinhaltet. Dieser Handler kann als »Unterprogramm« oder »Bibliotheksroutine« aufgefasst werden und hat somit eine genau definierte Einsprungadresse. Um also den Interrupthandler über ein interrupt oder trap gate ansprechen zu können, benötigt die CPU diese Einsprungadresse, die sie direkt dem interrupt oder trap gate descriptor entnimmt. Es fehlt nur noch die Basisadresse des dazugehörigen Segmentes. Auch diese erhält sie aus dem gate descriptor: Hier ist der Selektor in die GDT/LDT verzeichnet, der auf den Deskriptor des entsprechenden Segmentes zeigt. Und dieser Segment-Deskriptor enthält

natürlich die Basisadresse des Segmentes und seine Größe. Einem Ansprung der Interrupt-Routine steht nun nichts mehr im Wege. Abbildung 2.42 zeigt die Interrupt-Behandlung via interrupt oder trap gate.

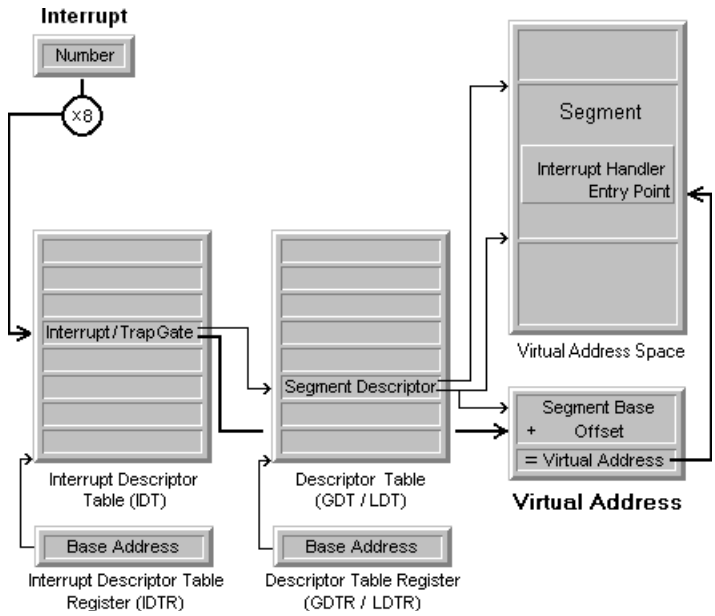


Abbildung 2.42: Berechnung der Adresse eines Interrupt-Handlers im Protected Mode

Die CPU legt daher den Inhalt des EFlags-Registers und die Rücksprungadresse auf den Stack. Und an dieser Stelle tritt der einzige Unterschied zwischen trap gate und interrupt gate zu Tage: Im Falle des Interrupt-Handlings durch ein interrupt gate löscht der Prozessor das IF-Flag im EFlags-Register und unterbindet damit die weitere Auslösung von Interrupts. Dies unterbleibt bei trap gates. Somit unterscheiden sich ein interrupt gate und ein trap gate voneinander nur dadurch, dass letzteres durch andere Interrupts unterbrochen werden kann.

task gates Im Falle eines task gates ist die Sache klar: Es muss einen Task geben, der den Interrupt handeln kann. Dieser Task muss zwar derzeit nicht aktiv sein; aber es muss ein task state segment, TSS, geben, das den Task beschreibt, und er muss per task switch aktivierbar sein. Und dieses TSS benötigt einen Eintrag in der GDT in Form eines Deskriptors. Das bedeutet: Der im task gate descriptor angegebene Selektor muss auf diesen TSS descriptor zeigen. Und damit ist recht einfach ein task

switch zu dem Task möglich, der den Interrupt handeln kann. In Abbildung 2.43 ist der Weg der Interrupt-Auslösung via task gate dargestellt:

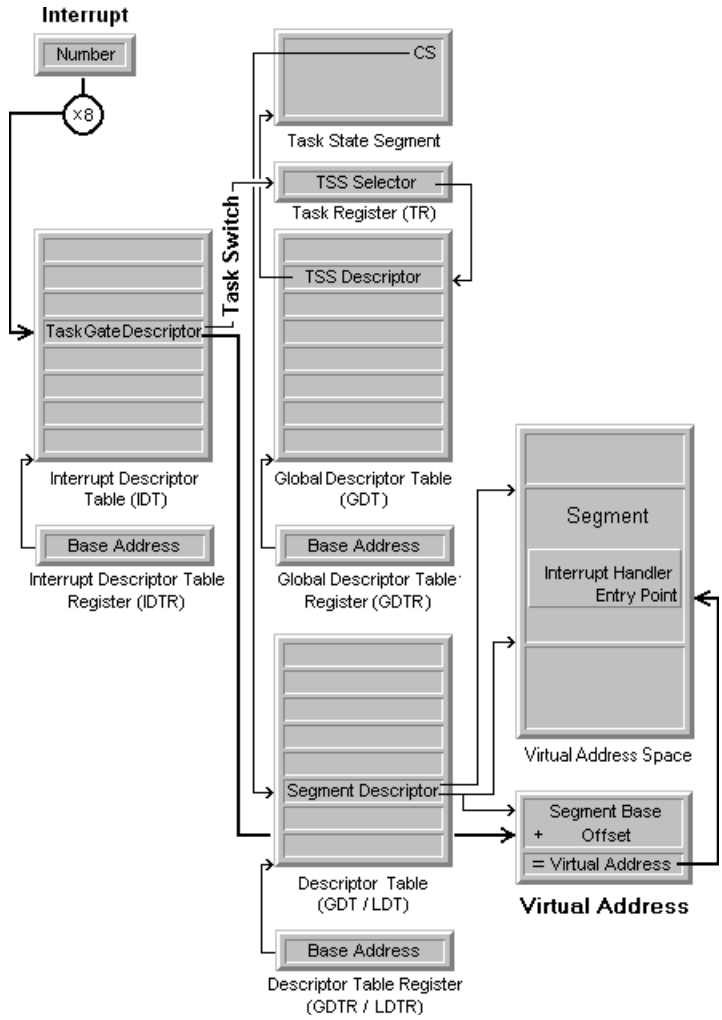


Abbildung 2.43: Interrupt-Behandlung über ein Task Gate

Aus dem task gate descriptor wird die Einsprungadresse des Interrupthandlers entnommen und mit der Basisadresse seines Segmentes zu einer virtuellen Adresse addiert, die dann beim task switch angesprungen wird. Der Weg, an diese Basisadresse zu gelangen, erscheint ein wenig umständlich. Aber dieser Aufwand ist nötig, da ja der Handler nur im Rahmen eines task switches angesprungen werden kann.

Dem task gate descriptor wird daher neben der Einsprungadresse auch ein Selektor in die GDT entnommen, der auf einen task state segment descriptor zeigt. Dieser TSS descriptor wiederum zeigt auf das zum gewünschten Task gehörige task state segment, das den eingefrorenen Zustand des Tasks beinhaltet. Hier liegt auch der Grund dafür, dass der Task bereits gestartet worden sein muss, wenn er auch inaktiv sein darf. Im TSS nun gibt es ein Speicherabbild des Codesegment-Registers (CS). Und in diesem hatte ja der Prozessor den Selektor in die GDT/LDT gespeichert, der auf den zum Task gehörigen Segment-Deskriptor zeigt. Also muss nur über diesen Selektor der Segment-Deskriptor ausgelesen werden, der ja die Basisadresse des Segmentes enthält, der den Interrupthandler beherbergt.



Wenn Sie Abbildung 2.42 und Abbildung 2.43 vergleichen, stellen Sie fest, dass der Aufwand, einen Interrupt via task switch zu behandeln, erheblich höher ist als über trap oder interrupt gates. Das ist auch so, und es sei nicht verheimlicht, dass der zum zweimaligen task switch notwendige Overhead ziemlich groß ist. Dennoch gibt es Gründe, weshalb man diesen Weg gehen kann und geht. Leider führte es jedoch im Rahmen dieses Buches zu weit, hierauf näher einzugehen: Da Sie selbst wohl kaum in die Verlegenheit kommen werden, Betriebssystemkomponenten zu schreiben, die das Interrupt-Handling betreffen, und da das Betriebssystem ja sowieso eigene Vorstellung davon hat, wer hier wie viel spielen darf, ist jedes weitere Wort in diesem Zusammenhang zu viel!

Fehlercodes

Wie bereits geschildert, legt die CPU bei der Auslösung von Exceptions teilweise vor der Ablage des EFlags-Registerinhaltes und der Rücksprungadresse einen Fehlercode auf den Stack. Bei welcher Exception welcher Code verwendet wird, wird bei der Besprechung der einzelnen Exceptions weiter unten angegeben. Dieser Fehlercode hat den in Abbildung 2.44 gezeigten allgemeinen Aufbau.

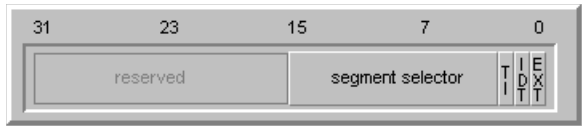


Abbildung 2.44: Speicherabbild der bei Exceptions verwendeten Fehlercodes

Das Flag EXT (external event) signalisiert im gesetzten Zustand, dass die Exceptionquelle »außerhalb des Programms«, also z.B. von der Hardware oder von Exception-Handlern, die nichts mit dem aktuellen Programm zu tun haben, ausgelöst wurde. Andernfalls handelt es sich um eine Software-Exception im aktuellen Kontext. Ist das Flag IDT, *descriptor location* (»*interrupt descriptor table*«), gesetzt, so zeigt *segment selector* auf einen Eintrag in der IDT, also ein task gate, ein trap gate oder ein interrupt gate. Andernfalls wird durch segment selector ein gate oder segment descriptor in der GDT oder LDT selektiert. TI, *table index*, entscheidet in diesem Fall, ob die GDT (TI = 0) oder LDT (TI = 1) heranzuziehen ist. Segment selector enthält den Selektor auf den entsprechenden Deskriptoren.

Bei page fault exceptions #PF hat der ErrorCode einen etwas anderen Aufbau, wie Abbildung 2.45 zeigt. In diesem Fall werden lediglich vier Bits übergeben, da die restlichen Informationen aus entsprechenden Registern gewonnen werden können. So gibt das *page not present flag* P, wenn gelöscht, an, dass die #PF aufgrund einer nicht vorhandenen Page erfolgte. Aufgabe des Handlers ist dann, diese Page nachzuladen. Ist P dagegen gesetzt, so war die Exception aufgrund einer Zugriffsverletzung auf Page-Ebene (RSVD, *reserved*, = 0) oder aufgrund der unerlaubten Nutzung eines reservierten Bits (RSVD = 1) in einem *page directory entry* oder *page table entry* ausgelöst worden. In diesem Fall signalisiert ein gesetztes flag U/S, *user/supervisor mode*, dass die Exception im user mode (CPL = 3) bzw., im gelöschten Zustand, im supervisor mode (CPL < 3) aufgrund eines lesenden (R/W, *read/write*, = 0 oder schreibenden (R/W = 1) Zugriffs generiert wurde.



Abbildung 2.45: Speicherabbild des bei einer page fault exception #PF verwendeten Fehlercodes

Interrupt-Ende

Erfolgte die Behandlung des Interrupts durch ein task gate, so erfolgt durch den abschließenden IRET-Befehl des Interrupthandlers ein erneuter task switch zu dem task, der durch den Interrupt unterbrochen worden ist. Nach Rückkehr aus dem Interrupthandler wird in allen Fällen, also sowohl im real mode wie auch im protected mode mit via trap, interrupt oder task gates ausgelösten Unterbrechungen der gleiche Ori-

ginalzustand wiederhergestellt, indem die ursprüngliche Stellung des IF-Flags aus dem auf den Stack geretteten EFlags-Registerinhalt restauriert wird. Der Prozessor nimmt dann die Arbeit an der Stelle des unterbrochenen Programms wieder auf, die er vor Eintritt in den Handler auf den Stack gerettet hat.



Interrupthandler müssen immer durch ein IRET abgeschlossen werden! Da der Prozessor beim Aufruf einer Interrupt-Routine grundsätzlich den Inhalt des EFlags-Registers vor der Rücksprungadresse auf den Stack sichert, hinterließe der Abschluss mittels RET die Flags auf dem Stack. Auch wenn das vielleicht nicht tragisch, wenn auch fast vorsätzlich schlampig programmiert wäre – die Gefahr geht von etwas anderem aus: Der Prozessor löscht bei Eintritt in den Interrupthandler das IF-Flag, um zu verhindern, dass ein weiterer Interrupt während der Behandlung eines Interrupts ausgelöst werden kann. Durch das Rückspeichern des gesicherten EFlags-Registers mittels IRET wird dieses IF-Flag auf seinen ursprünglichen Inhalt zurückgesetzt. Schließt man mittels RET ab, bleibt IF gelöscht – und es findet bis zum Sankt-Nimmerleins-Tag kein Interrupt mehr statt.

2.5.4 Emulation von Exceptions

Mittels des INT-Befehls ist praktisch jeder Interrupt auslösbar, der in der IDT (bzw. in der IVT des real mode) verzeichnet ist. Dies bedeutet, dass auch Exceptions, die von der CPU ausgelöst werden, mittels INT emuliert werden können.



Achtung! Hierbei ist im protected mode absolute Vorsicht angebracht! Während Hardware- und Software-Interrupts keinerlei Informationen an den Interrupthandler übergeben, bei Eintritt in den Handler also nur der Inhalt von EFlags und die Rücksprungadresse auf dem Stack liegen, ist das bei Exceptions anders. Hier legt die CPU häufig einen *error code* auf den Stack, der dem Handler weitere Informationen über die Ursache der Exception geben soll. Beispielsweise übergibt die CPU bei #NP via Error-Code den Selector des nicht vorhandenen Segmentes oder bei #PF Informationen über die nachzuladende Page. Der Handler erwartet somit bei bestimmten Exceptions einen zusätzlichen Code, den er von Stack holen kann. Liegt dort keiner, da die Exception mittels eines INT-Befehles emuliert und nicht auf die korrekte Stackbelegung geachtet wurde, verwendet er hierzu das an der Stackspitze liegende Double-

Word – und das ist der EIP-Inhalt, also ein Teil der Rücksprungadresse. Nun liegen nur noch der CS-Inhalt (Segment-Selektor) und EFlags auf dem Stack, was der Prozessor als Rücksprungadresse missinterpretiert und den Selektoren als Offset und EFlags als Selektor betrachtet. Resultat: Der den Handler abschließende IRET-Befehl wird nun mit Sicherheit ins Nirwana zurückspringen – was im harmlosesten Fall zu einer #GP (*general protection exception*) führen wird.

Aus diesen Gründen ist es nicht einfach, eine Exception zu emulieren. Denn aus bekannten Gründen muss der INT-Befehl verwendet werden, der jedoch keinen Error-Code als Parameter akzeptiert. Somit kann nur dann ein Fehlercode *hinter* die vom INT-Befehl gesicherte Rücksprungadresse auf den Stack geschoben werden, wenn – INT nicht verwendet wird! So könnte man z.B. den Stack auch »von Hand« aufbauen: EFlags pushen, eine Rücksprungadresse pushen, den Fehlercode pushen und dann – ein unbedingter Sprung zur gewünschten Adresse. Aber dies macht noch mehr Schwierigkeiten: Woher die Adresse nehmen? Den ganzen Aufwand treiben, den der Prozessor selbst im Rahmen des INT-Befehls erledigen würde, als da wäre: Interrupt-Nummer mal 8, Suche der Tabellenadresse (IDTR), Berechnung der Adresse des Eintrags, Auslesen des Deskriptors, Prüfung auf task gate oder interrupt bzw. trap gate ... Lassen Sie es lieber!

2.5.5 CPU-Exceptions

Tabelle 2.2 zeigt die Belegung der 256 möglichen Interrupts, die in der IDT verzeichnet sein können. Die Interrupt-Nummern 0 bis 31 hat Intel für eigene Zwecke reserviert, die Nummern 32 bis 255 gelten als »frei verfügbar«.

Der Begriff »frei verfügbar« (»user defined«) ist hierbei missverständlich. Als »user« sieht Intel hier nicht den Anwender, sondern den Hersteller des Betriebssystems. Da die IDT ein vom Betriebssystem verwendetes und verwaltetes Segment ist, haben Sie in Ihren Programmen vermutlich nicht die Privilegien, die IDT zu verändern. Dies müssten Sie aber, um eigene Interrupthandler für die »user defined« Interrupts installieren zu können. Gehen Sie daher davon aus, dass Sie im protected mode Interrupts und Exceptions nur auslösen können – was dann nach dem Auslösen passiert, ist Sache des Betriebssystems.

Die als reserviert markierten Exceptions und Interrupts sollten Sie niemals verwenden. Sie werden von den Intel-Prozessoren intern genutzt.

Beispielsweise wurde Interrupt #9 beim 80387 verwendet, wenn während der Datenübertragung zwischen FPU-Registern und Speicher eine Segment- oder Page-Verletzung auftrat. Außer dem Gespann 80386/80387 verwendet keine andere FPU/CPU INT 09. Die reservierten Interrupts 20 bis 31 hat Intel für künftige Erweiterungen reserviert. So ist z.B. INT 19, #XF, erst mit den SSE-/SSE2-Fließkommabefehlen und damit seit dem Pentium III implementiert. Es bleibt also noch Raum genug für neue Entwicklungen.

#		Beschreibung	Exception Interrupt	Ursache	Klasse, Typ	Fehlercode
0	#DE	Divide Error	E	CPU	f, c	n
1	#DB	Debug	E	CPU/S	f/t, b	n
2	-	Non-maskable Interrupt	I	H	-, b	n
3	#BP	Break Point	E	S	t, b	n
4	#OF	Overflow	E	S	t, b	n
5	#BR	Bound Range Exceeded	E	S	f, b	n
6	#UD	Invalid Opcode	E	CPU/S	f, b	n
7	#NM	Device Not Available	E	CPU	f, b	n
8	#DF	Double Fault	E	CPU	a, b	j
9	-	FPU Segment Overflow	E	-	a, -	n
10	#TS	Invalid TSS	E	CPU	f, c	j
11	#NP	Segment Not Present	E	CPU	f, c	j
12	#SS	Stack Segment Fault	E	CPU	f, c	j
13	#GP	General Protection	E	CPU	f, c	j
14	#PF	Page Fault	E	CPU	f, pf	j
15	-	reserviert	E	-	-	n
16	#MF	Math Fault (FPU-Exception)	E	CPU	f, b	n
17	#AC	Alignment Check	E	CPU	f, b	j
18	#MC	Machine Check	E	CPU	a, b	n
19	#XF	SIMD-Exception (floating point)	E	CPU	f, b	n
20-31	-	reserviert	-	-	-, b	-
32-255	-	»frei verfügbare« Interrupts	I	S	-, b	-

Es bedeuten: E Exception, I Interrupt; H Hardware, S Software; f fault, t trap, a abort; b benign, c contributory, pf page fault; n nein, j ja. Die grau unterlegten Interrupts/Exceptions gelten als reserviert und sollten nicht benutzt werden

Tabelle 2.2: Liste der möglichen Exceptions und Interrupts im protected mode

Klassen Wie Sie Tabelle 2.2 entnehmen können, werden die Exceptions in bestimmte Klassen und Typen eingeteilt:

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. 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

Traps 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, *aborts*, 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 in einem Hardwarebaustein.

Typen

Neben den Exception-Klassen gibt es auch Exceptiontypen, mit denen Exceptions kategorisiert werden können:

benign

Benigne Fehler sind »gutmütige« Fehler, die außer den zu der spezifischen Ausnahmesituation gehörenden keine weiteren Probleme erzeugen und in der Regel leicht zu beheben sind. Beispiel: Überlauf. Die Tatsache, dass bei einer arithmetischen Berechnung der Wertebereich des Zieloperanden überschritten wurde, hat keine weiteren Auswirkungen, vor allem auf den Programmstatus.

contributory

Contributory exceptions sind Fehlerzustände, die an einem durch den Fehler veränderten Programmverlauf »mitwirken«, ihn »mit verursa-

chen«. Beispiel: Stack fault. Wenn der Stack überläuft, gibt es ein Problem, das auf den weiteren Verlauf des Programms Auswirkungen hat. In diesem Fall kann z.B. keine Rücksprungadresse mehr auf den Stack geschrieben werden, also die aufzurufende Routine auch nicht aufgerufen werden.

page fault Der Seitenfehler, *page fault*, signalisiert, dass versucht wurde, auf eine Seite zurückzugreifen, die derzeit nicht im Speicher verfügbar ist. Zwar ist dies eine Ausnahmesituation und damit eine Exception. Doch ist dies Teil des Paging-Mechanismus und damit ein Typ von Exceptions, der mit den Typen *benign* oder *contributory* nicht vergleichbar ist.

Prioritäten Falls im Kontext eines Befehls mehrere Exceptions auftreten, bedient sie die CPU anhand einer Prioritätenliste. Sie wird in Tabelle 2.3 angegeben.

Priorität	Beschreibung
1 (höchste)	Hardware-Reset und #MC (in dieser Reihenfolge)
2	Task switch: Flag T (trap) ist gesetzt
3	Externe Hardware-Intervention: FLUSH, STOPCLK, SMI, INIT
4	#BR, #DB
5	Externe Interrupts: NMI, maskierbare Hardware-Interrupts
6	Fehler beim Holen des nächsten Befehls: code breakpoint fault, code segment limit violation, code page fault (#PF)
7	Fehler beim Decodieren des nächsten Befehls: Länge der Befehlssequenz > 15 Bytes, #UD, #NM
8 (niedrigste)	Fehler bei der Ausführung eines Befehls: #OF, #BR, #TS, #NP, #SS, #GP, data page fault (#PF), #AC, #MF, #XF

Tabelle 2.3: Prioritätenliste für die Bearbeitung von Exceptions

Divide Error Eine #DE zeigt an, dass der Divisor bei DIV oder IDIV Null ist oder dass das Ergebnis der Division nicht mit der Anzahl von Bits dargestellt werden kann, die im Zieloperanden verfügbar sind.

Interrupt Interrupt-Nummer: 0

Quelle Interruptquelle: CPU

Klasse Klasse: fault

Typ Typ: contributory

ErrorCode ErrorCode: keiner

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf den Befehl, der die Exception ausgelöst hat. *Rücksprung*

Ein Wechsel im Programmstatus erfolgt nicht, da die Exception auftritt, bevor die die Exception verursachende Instruktion ausgeführt wird. Nach Exception-Behandlung kann die Programmausführung normal wieder aufgenommen werden. *Statuswechsel*

Eine #DB zeigt an, dass eine oder mehrere Bedingungen für eine Exception vorgefunden wurden. Es gibt verschiedene Gründe, die zu einer #DB führen können, und damit auch verschiedene Exception-Klassen: **Debug**

- ein Breakpoint wurde gefunden (fault)
- ein überwachtes Datum wurde verändert (trap)
- es erfolgte eine Ein- oder Ausgabe (trap)
- es besteht eine »general detection condition« (fault)
- Einzelschrittausführung (trap)
- es erfolgte ein task switch (trap)

Interrupt-Nummer:	1	*Interrupt*
Interruptquelle:	CPU	*Quelle*
Klasse:	fault oder trap; die Unterscheidung erfolgt anhand der Analyse der Debugregister, vor allem DR6.	*Klasse*
Typ:	benign	*Typ*
ErrorCode:	keiner; der Handler kann anhand der Debugregister feststellen, welche Ursache die Exception hatte.	*ErrorCode*

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt im Falle einer Exception der Klasse *fault* auf den Befehl, der die Exception ausgelöst hat, bei einer Exception der Klasse *trap* auf den Befehl, der dem die Exception verursachenden folgt. *Rücksprung*

Im Falle einer Exception der Klasse *fault* erfolgt ein Wechsel im Programmstatus nicht, da die Exception auftritt, bevor die die Exception verursachende Instruktion ausgeführt wird. Nach Exception-Behandlung kann die Programmausführung normal wieder aufgenommen werden. *Statuswechsel*

Im Falle einer Exception der Klasse trap dagegen ändert sich der Status des Programms, da die Instruktion (Einzelschrittausführung) oder der erfolgte task switch (überwachtes Datum verändert, Ein-/Ausgabe, task switch) abgeschlossen werden muss, bevor die Programmausführung wieder aufgenommen werden kann. Allerdings ist danach der Programmstatus nicht korrumpiert, sodass das Programm problemlos fortgeführt werden kann.

NMI Der NMI ist ein nicht maskierbarer Interrupt (*non-maskable interrupt*), der durch externe Quellen ausgelöst wird, indem ein Signal am Pin NMI# angelegt wird oder durch den I/O-APIC (advanced programmable interrupt controller) im lokalen APIC ein NMI request gesetzt wird. Hierdurch wird der NMI-Handler aufgerufen.

Interrupt Interrupt-Nummer: 2

Quelle Interruptquelle: extern

Klasse Klasse: nicht zutreffend

Typ Typ: benign

ErrorCode ErrorCode: nicht zutreffend

Rücksprung Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf den Befehl, der dem die Exception verursachenden folgt.

Statuswechsel Der Befehl, der durch den NMI unterbrochen wird, wird in jedem Falle vollständig beendet, bevor der NMI ausgelöst wird. Das bedeutet, der Zustand des Programms ändert sich unter der Voraussetzung nicht, dass der NMI-Handler den CPU-Status vor der NMI-Behandlung sichert und vor der Rückkehr ins unterbrochene Programm wieder restauriert.

Breakpoint Eine #BP zeigt an, dass die CPU einen INT3-Befehl ausgeführt hat. Üblicherweise setzt ein Debugger einen Breakpoint, indem er das erste Byte aus der Befehlssequenz einer Instruktion durch den Opcode für den INT3-Befehl ersetzt.

Interrupt Interrupt-Nummer: 3

Quelle Interruptquelle: Software (INT3)

Klasse Klasse: trap

Typ Typ: benign

ErrorCode ErrorCode: keiner

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf den Befehl hinter dem INT3-Befehl. *Rücksprung*

Obschon die Rücksprungadresse auf den Befehl hinter dem INT3-Befehl zeigt, ändert sich am Zustand des Programms nichts, da der INT3-Befehl weder Register verändert noch auf Speicher zugreift. Das bedeutet, dass der Debugger die Bearbeitung des unterbrochenen Programms dadurch fortsetzen kann, dass er das erste Byte der Bytesequenz, das der vorab durch den INT3-Befehl substituiert hat, restauriert und die auf dem Stack liegende Rücksprungadresse dekrementiert. Nach Rückladen der so modifizierten Rücksprungadresse durch IRET wird die Programmausführung an der Stelle wieder aufgenommen, an der sich der Breakpoint befunden hat. *Statuswechsel*

Bei Prozessoren ab dem 80386 empfiehlt es sich, die leistungsfähigeren Möglichkeiten der Breakpoint-Verwaltung mit Hilfe der Debugregister zu nutzen. *Kompatibilität*

#BP kann auf zwei Arten ausgelöst werden: durch den Ein-Byte-Befehl INT3 sowie durch den Zwei-Byte-Befehl INT nn, wobei dem »normalen« INT-Befehl die Konstante $03 übergeben wird. Der Ablauf ist in beiden Fällen leicht unterschiedlich. So erfolgt bei INT3 keine interrupt redirection im VME Modus: Der Interrupt wird durch einen protected mode handler bearbeitet. Auch erfolgt im virtual 8086 mode keine IOPL-Prüfung, sodass der Interrupt auf jeder Privilegstufe behandelt wird. Bei INT mit Argument $03 ist das nicht der Fall. *Bemerkungen*

Hinweis: Alle mir bekannten Assembler übersetzen INT mit Argument $03 in die Ein-Byte-Version INT3. Die Zwei-Byte-Version müsste »von Hand« oder durch selbst-modifizierenden Code erzeugt werden.

Overflow

Eine #OF zeigt an, dass die CPU einen INTO-Befehl ausgeführt hat, der ein gesetztes overflow flag vorgefunden hat.

Interrupt-Nummer:	4	*Interrupt*
Interruptquelle:	Software (INTO)	*Quelle*
Klasse:	trap	*Klasse*
Typ:	benign	*Typ*
ErrorCode:	keiner	*ErrorCode*

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf den Befehl hinter dem INTO-Befehl. *Rücksprung*

Statuswechsel Obschon die Rücksprungadresse auf den Befehl hinter dem INT3-Befehl zeigt, ändert sich am Zustand des Programms nichts, da der INT3-Befehl weder Register verändert noch auf Speicher zugreift. Das bedeutet, dass die Programmausführung korrekt an der Stelle aufgenommen werden kann, auf die der auf den Stack gesicherte instruction pointer zeigt.

Bemerkungen Der Interrupt kann auf zwei Arten ausgelöst werden: durch den Ein-Byte-Opcode $CE (»INTO«) oder durch den Zwei-Byte-Opcode $CD04 (»INT 04«).

Bound Range Exceeded Eine #BR zeigt an, dass bei der Ausführung des BOUND-Befehls ein vorzeichenbehafteter Index in ein als Operand übergebenes Array dessen untere Grenze unter- bzw. die obere Grenze überschritten hat.

Interrupt Interrupt-Nummer: 5

Quelle Interruptquelle: Software (BOUND)

Klasse Klasse: fault

Typ Typ: benign

ErrorCode ErrorCode: keiner

Rücksprung Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf den BOUND-Befehl.

Statuswechsel Der Zustand des Programms wird nicht verändert, da der BOUND-Befehl keine Veränderungen an den Operanden vornimmt. Nach Rückkehr aus dem Handler wird somit der BOUND-Befehl ein weiteres Mal ausgeführt.

Bemerkungen ACHTUNG: Der Handler muss die Ursache der Exception beseitigen! Da der Handler immer an den BOUND-Befehl zurückkehrt, würde andernfalls eine Endlosschleife entstehen.

Invalid Opcode Eine #UD (undefined opcode, Synonym zu invalid opcode) wird in folgenden Fällen ausgelöst:

- Versuch der Ausführung eines ungültigen oder reservierten Opcodes
- Versuch der Ausführung einer Instruktion mit einem Operanden, der im Befehlskontext ungültig ist. Dies ist zum Beispiel der Fall, wenn der LES-Befehl ausgeführt werden soll, der Operand jedoch nicht auf eine Speicherstelle zeigt.

- Versuch der Ausführung eines SIMD-Befehls (MMX, SSE, SSE2) auf einem Prozessor, der diese Erweiterungen nicht besitzt.
- Versuch der Ausführung eines SIMD-Befehls (mit Ausnahme von PAUSE, PRETECHx, SFENCE, LFENCE, MFENCE oder CLFLUSH) bei gesetzten Flag EM in Kontrollregister CR0.
- Versuch der Ausführung eines SSE- oder SSE2-Befehls (mit Ausnahme von MASKMOV(D)Q, MOVNT(D)Q, MOVNTPD, MOVNTI, PREFETCHx, SFENCE, LFENCE, MFENCE oder den 64-Bit-MMX-Versionen, die durch SSE bzw. SSE2 eingeführt wurden, also PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULUHW, PSADBW, PSHUFW, PADDQ, PSUBQ) bei gelöschtem OSFXSR-Flag in Kontrollregister CR4, das Betriebssystem also die z.B. bei einem task switch erforderliche Sicherung/Restaurierung der SIMD-Umgebung mittels FXSAVE/FXRSTOR nicht unterstützt.
- Versuch der Ausführung einer SSE- oder SSE2-Instruktion, die eine SIMD-Fließkomma-Exception #XF auslöst, wenn das Flag OSXMMEXCEPT in Kontrollregister CR4 gelöscht ist, das Betriebssystem also keinen Exception-Handler zur Verfügung stellt.
- Ausführung des Befehls UD2
- Existenz des Präfixes LOCK als Teil einer Befehlssequenz, die nicht geLOCKt werden kann oder bei der der Operand kein Speicheroperand ist, wenn LOCK erlaubt ist.
- Versuch, LLDT, SLDT, LTR, STR, LAR, VERR, VERW oder ARPL im real oder virtual 8086 mode auszuführen.
- Versuch, RSM außerhalb des SMM-Modus auszuführen.

Interrupt-Nummer:	6	*Interrupt*
Interruptquelle:	CPU / Software (UD2)	*Quelle*
Klasse:	fault	*Klasse*
Typ:	benign	*Typ*
ErrorCode:	keiner	*ErrorCode*

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf den Befehl, der die Exception ausgelöst hat. *Rücksprung*

Der Programmzustand wird nicht geändert, da die ungültige Instruktion nicht ausgeführt wird. *Statuswechsel*

Kompatibilität Der Pentium 4, die P6-Familie und der Pentium verfügen über verschiedene Arten der Befehlsdecodierung, die sich erheblich von der der vorangegangenen Art unterscheiden. So verfügt der Pentium 4 über einen Decodierungsmechanismus, der mit micro-ops arbeitet, die nach der Bearbeitung restauriert werden (vgl. Seite 874). Die P6-Familie arbeitet mit »spekulativer Befehlsausführung« im Rahmen der branch prediction (vgl. Seite 875). Und der Pentium und nachfolgende Prozessoren benutzen instruction prefetching (vgl. Seite 875). Bei diesen Mechanismen ist der Befehl teilweise bereits lange decodiert, bevor er tatsächlich zur Ausführung kommen kann. In diesen Fällen unterbleibt die Auslösung der Exception, trotzdem bereits zu einem sehr frühen Zeitpunkt bekannt ist, dass es zu einer Exception kommen wird. Sie wird erst dann tatsächlich ausgelöst, wenn die Ausführung des Befehls erfolgen soll (also beim Pentium 4 die micro-ops »retired« werden, bei der P6-Familie tatsächlich zu dem betreffenden Befehl verzweigt wird oder beim Pentium der Befehl von der decoding unit tatsächlich an die execution unit übergeben wird). Zeitpunkt der Exception-Generierung ist somit nicht notwendigerweise der Zeitpunkt der Dekodierung (Prozessoren bis einschließlich 80486), sondern der des Beginns der Ausführung.

Bemerkungen Die Opcodes $D6 und $F1 sind ungültige Opcodes, die für die IA-32-Architektur reserviert sind. Obwohl undefiniert, erzeugen sie eine #UD.

Device Not Available Die #NM (*no math unit*) wurde ursprünglich geschaffen, um eine Software-Emulation der FPU-Befehle zu ermöglichen, wenn keine interne FPU oder eine externe NPX verfügbar war. Das Fehlen der FPU wird im Flag EM des Kontrollregisters CR0 signalisiert. Ist es gesetzt, so führt jede Ausführung eines FPU-Befehls zu dieser Exception. Der Exception-Handler kann daraufhin die FPU-Befehle emulieren.

Mit der Einführung von MMX als erster Stufe der SIMD haben die FPU-Register aber zusätzliche Aufgaben bekommen. Daher ist es nicht ausgeblieben, #NM an diese Situation anzupassen. eine #NM wird somit in folgenden Fällen ausgelöst:

- Ein FPU-Befehl soll ausgeführt werden, jedoch ist das EM-Flag gesetzt und zeigt an, dass die Hardwarevoraussetzungen (FPU) fehlen. Der FPU-Software-Emulator kann aufgerufen werden.
- Die CPU führte eine WAIT/FWAT-Instruktion aus und die Flags MP und TS in Kontrollregister CR0 sind unabhängig vom Status des EM-Flags gesetzt. So zeigt ein gesetztes TS-Flag an, dass nach dem

letzten FPU-, MMX-, SSE- oder SSE2-Befehl ein task switch erfolgte, jedoch entsprechende »Umgebung« (FPU-Register, status word, control word, XMM-Register, MXCSR) nicht gesichert wurde. Ist nun gleichzeitig EM gelöscht (d. h. die Hardware vorhanden), so kann der Handler die Sicherung der Umgebung vornehmen, da in diesem Fall nach jeder FPU- oder SIMD-Instruktion eine #NM ausgelöst wird (siehe nächster Punkt). Das Flag MP hat übrigens die Aufgabe, dieses Verhalten auch für WAIT/FWAIT freizuschalten, weshalb nur bei gesetztem MP und TS-Flag die Ausführung eines WAIT/FWAIT zu einer #NM führt. Ist es dagegen gelöscht, löst WAIT/FWAIT keine #NM aus.

- Es wurde eine FPU- oder SIMD-Instruktion (mit Ausnahme von PAUSE, PREFETCHx, SFENCE, LFENCE, MFENCE und CLFLUSH) ausgeführt, das Flag EM im Kontrollregister CR0 ist gelöscht (= keine Emulation!) und das Flag TS in CR0 zeigt durch seinen gesetzten Zustand an, dass ein task switch erfolgte. In diesem Fall hat der Handler, der #NM behandelt, für die Sicherung der FPU- bzw. SIMD-Umgebung zu sorgen.

Interrupt-Nummer:	7	*Interrupt*
Interruptquelle:	CPU	*Quelle*
Klasse:	fault	*Klasse*
Typ:	benign	*Typ*
ErrorCode:	keiner	*ErrorCode*

Rücksprung

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf die Instruktion, die die Exception ausgelöst hat.

Statuswechsel

Eine Änderung des Programmstatus erfolgt nicht, da die Instruktion, die die Exception ausgelöst hat, nicht ausgeführt wurde. Falls das EM-Flag in CR0 gesetzt ist, kann der Exception-Handler den Zeiger auf die FPU-Instruktion auf dem Stack benutzen, um die betreffende Instruktion festzustellen und zu emulieren. Falls TS gesetzt sein sollte, kann der Handler die FPU- bzw. SIMD-Umgebung sichern, das TS-Flag löschen und seine Aktion dann beenden. Danach wird die die Exception auslösende Instruktion nochmals ausgeführt.

Bemerkungen

Das MP-Flag ist hauptsächlich zur Benutzung mit dem 80286 und 80386DX implementiert worden. Beim 80486SX sollte es immer gelöscht sein, beim 80486DX und dem 80487SX sowie allen folgenden Prozessoren sollte es immer gesetzt sein!

Double Fault Die #DF zeigt an, dass die CPU eine Ausnahmesituation vorgefunden hat, während sie eine andere Exception bearbeitet. Zwar kann sie üblicherweise hintereinander auftretende Exceptions auch hintereinander (»seriell«) behandeln. Doch gibt es Ausnahmen. So erzeugt sie immer dann eine #DF, wenn beide Exceptions vom Typ *page fault* sind (vgl. Seite 501), beide vom Typ *contributory* oder die erste vom Typ *contributory* und die zweite vom Typ *page fault* (aber nicht umgekehrt!):

erste Exception	zweite Exception		
	benign	contributory	page fault
benign	serielle Behandlung	serielle Behandlung	serielle Behandlung
contributory	serielle Behandlung	#DF	serielle Behandlung
page fault	serielle Behandlung	#DF	#DF

Interrupt Interrupt-Nummer: 8

Quelle Interruptquelle: CPU

Klasse Klasse: abort

Typ Typ: keiner

ErrorCode ErrorCode: Es wird ein ErrorCode mit dem Wert »0« auf des Stack gelegt.

Rücksprung Der auf den Stack gelegte *instruction pointer* (CS:(E)IP) (»Rücksprungadresse«) ist undefiniert.

Statuswechsel Der Programmstatus nach einer #DF ist undefiniert. Das bedeutet: die Programmausführung kann nicht wieder aufgenommen werden. Die einzige Aufgabe des Exception-Handlers für double fault exceptions ist, so viele Informationen wie möglich zu sammeln und sie ggf. Diagnosetools zur Verfügung zu stellen und danach die Applikation zu schließen und/oder den Prozessor herunterzufahren oder zurückzusetzen.

Bemerkungen Falls eine weitere Exception auftritt, während die CPU den *double fault exception handler* ausführt, wird der Prozessor in den *shut-down mode* gefahren, der dem HLT-Zustand ähnelt. Fall der *shut down* erfolgt, während die CPU einen NMI handelt, wird ein *hardware reset* erforderlich!

Coprocessor Segment Overrun Diese Exception ist durch Intel reserviert und sollte nicht aufgerufen werden. Sie hatte nur bei der Kombination 80386/80387 eine Bedeutung. So musste bei einigen NPX-Instruktionen, die mit Speicheroperanden umgingen, die Datenübertragung in drei Teilen erfolgen.

Erfolgte nun während der Übertragung des mittleren Teils eine Page- oder Segmentverletzung, wurde diese Exception ausgelöst. Sie hat seit dem 80486 keine Bedeutung mehr, da diese Aufgabe durch eine general protection exception #GP übernommen wurde.

Interrupt-Nummer:	9	*Interrupt*
Interruptquelle:	CPU	*Quelle*
Klasse:	abort	*Klasse*
Typ:	benign	*Typ*
ErrorCode:	keiner	*ErrorCode*

Rücksprung

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP zeigt die Instruktion, die die Exception auslöste.

Statuswechsel

Der Programmstatus nach einer #DF ist undefiniert. Das bedeutet: die Programmausführung kann nicht wieder aufgenommen werden, das Programm wird beendet.

Bemerkungen

Bei 80486ern, Prozessoren der Pentium- oder P6-Familie oder beim Pentium 4 tritt, wie gesagt, die co-processor segment overrun exception nicht mehr auf. Diese Prozessoren brechen die Instruktion einfach dort ab, wo bei 80387ern die Exception ausgelöst würde. Um unentdeckte segment overruns zu erkennen, sollte daher die Umgebung in der gleichen Seite abgelegt werden wie das TSS. Dies verhindert, dass sie bei einem task switch verloren geht, wenn während eines FLDENV, FSTOR oder FXRSTOR ein page fault auftritt.

Invalid TSS

Die Exception #TS signalisiert, dass bei einem versuchten task switch ungültige Informationen im task state segment (TSS) vorgefunden wurden. Dies kann verschiedene Gründe haben, nämlich Verletzung der Zugriffsbeschränkungen für das TSS oder der im TSS gesicherten, neu zu ladenden Inhalte für LDT, CS, DS oder SS (vgl. auch »ErrorCode« weiter unten).

Die Exception kann entweder im Umfeld des alten oder des neuen Tasks auftreten. Solange, bis die CPU die vollständige Gültigkeit und Verfügbarkeit des TSS festgestellt hat, erfolgt die Exceptionauslösung im Kontext des alten Tasks. Danach gilt der task switch (mit dem Laden des validen Selektors in das TR) als erfolgt, und die Exception wird im Kontext des neuen Tasks ausgelöst. Zu diesem Zeitpunkt sind die Register CS, DS und SS sowie LDTR noch nicht neu belegt.

Interrupt Interrupt-Nummer: 10 ($0A)

Quelle Interruptquelle: CPU

Klasse Klasse: fault

Typ Typ: contributory

ErrorCode ErrorCode: Entsprechend der Ursache der Exception wird folgender ErrorCode auf den Stack gelegt:

ErrorCode	Exception-Grund
TSS segment selector	Segmentlimit des TSS kleiner $67 Bytes bei 32-Bit- oder kleiner $2C Bytes bei 16-Bit-TSSs
LDT segment selector	Ungültige LDT oder LDT nicht verfügbar
stack segment selector	Selektor ist größer als descriptor table limit
stack segment selector	Stacksegment nicht beschreibbar
stack segment selector	CPL ≠ selector RPL
stack segment selector	CPL ≠ segment DPL
code segment selector	Selektor ist größer als descriptor table limit
code segment selector	Codesegment nicht ausführbar
code segment selector	CPL ≠ non-conforming segment DPL
code segment selector	CPL < conforming segment DPL
data segment selector	Selektor ist größer als descriptor table limit
data segment selector	Datensegment nicht lesbar

Zusätzlich wird Bits 0 bis 2 des ErrorCodes gesetzt: Das EXT-Flag wird gesetzt, wenn die Ursache der Exception außerhalb des aktuelle Programms lag, also z.B., wenn ein externer Interrupthandler über ein task gate versuchte, einen task switch mit einer ungültigen TSS durchzuführen. Andernfalls wird EXT gelöscht.

Rücksprung Wenn der Task-Switch bereits erfolgt ist, der zu dem Ausnahmezustand führte, zeigt der *instruction pointer* CS:(E)IP auf dem Stack (»Rücksprungadresse«) auf die erste Instruktion des neuen Tasks, andernfalls auf die Instruktion im alten Task, die den Task-Switch verursachte.

Statuswechsel Der Zustand des Programms 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. (Vorher hätte der kaum die Möglichkeit, das TSS auszulesen!) 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 kann sich somit *nicht* darauf verlassen, dass die Segmentregister-Inhalte valide sind, er 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

Eine #NP wird ausgelöst, wenn das Flag P (present) in einem Descriptor gelöscht ist, der gerade verwendet werden soll. Dadurch wird signalisiert, dass das betreffende Segment sich zurzeit nicht im Speicher befindet, sondern nachgeladen werden muss. Gründe für das Auslösen einer #NP können sein:

- Der Versuch, eines der Segmentregister CS, DS, ES, FS oder GS im Rahmen eines task switches zu laden. Ein nicht vorhandenes Stacksegment (Laden von SS) wird durch eine #SS signalisiert.

- Der Versuch, via LLDT das local descriptor table register LDTR mit einer neuen local descriptor table zu laden. Eine nicht vorhandene LDT bei einem task switch wird dagegen durch eine #TS signalisiert.
- Der Versuch, ein nicht vorhandenes task state segment (TSS) in das task register TR zu laden.
- Der Versuch, ein zwar gültiges, aber eben nicht vorhandenes task state segment (TSS) oder einen gate descriptor zu benutzen.

Interrupt Interrupt-Nummer: 11 ($0B)

1. Quelle

Interruptquelle: CPU

Klasse Klasse: fault

Typ Typ: contributory

ErrorCode ErrorCode: Es wird ein ErrorCode gemäß Abbildung 2.44 auf Seite 496) auf den Stack gelegt. EXT ist gesetzt, wenn ein externes Ereignis wie ein NMI oder ein INTR (interrupt request, erzeugt durch den programmable interrupt controller PIC) zur Exception führte. IDT ist gesetzt, wenn sich der Fehlercode auf einen Eintrag in der interrupt descriptor table (IDT) bezieht, weil z.B. ein Interrupt auf ein nicht vorhandenes Segment zugreifen will.

Rücksprung Üblicherweise zeigt der *instruction pointer* CS:(E)IP auf dem Stack (»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, zeigt er auf die erste Instruktion des neuen Tasks. Falls die Exception ausgelöst wurde, als auf einen Gate-Deskriptor zugegriffen wurde, zeigt er auf den Befehl, der diesen Zugriff veranlasste (z.B. auf ein CALL).

Statuswechsel 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 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. Vergleiche hierzu die Informationen zum »Statuswechsel« bei der *invalid task segment exception* #TS auf Seite 511.

Kompatibilität

Wenn auf 80486ern eine #NP im Verlauf einer FLDENV-Instruktion auftritt, kann es sein, dass nur ein Teil der Umgebung restauriert wird. Dann besitzt das control word den Inhalt $007F. Die Pentium-, P6- und Pentium-4-Familie umgeht dieses Problem, indem sie versucht, das erste und letzte Byte der Umgebung zu lesen, bevor die Umgebung als Ganzes restauriert wird. Eine #NP im Verlauf der Instruktion ist damit nicht mehr möglich.

Stack Segment Fault

Eine #SS zeigt an, dass folgende Ausnahmebedingungen entdeckt worden sind:

- 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.

Interrupt-Nummer:	12 ($0C)	*Interrupt*
Interruptquelle:	CPU	*Quelle*
Klasse:	fault	*Klasse*
Typ:	contributory	*Typ*

ErrorCode:Der ErrorCode auf dem Stack ist 0, wenn eine »normale« Verletzung der Grenzen eines bereits im Gebrauch befindlichen Stack-Segments stattgefunden hat. Wenn jedoch diese Exception ausgelöst *ErrorCode*

wird durch ein nicht vorhandenes Stacksegment oder den Überlauf des »neuen« Stacks nach einem inter-privileg-level call (CALL, bei dem die Privilegstufe geändert wird), enthält der ErrorCode gemäß Abbildung 2.44 auf Seite 496 den Selektor für das Segment, das Ursache für die Exception war. 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 nur das Limit des Stacks verändert zu werden.

Rücksprung Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt üblicherweise auf die Instruktion, die die Exception auslöste. Falls aber die Exception im Rahmen eines task switch erfolgte, zeigt die Rücksprungadresse auf den ersten Befehl des neuen Tasks.

Statuswechsel 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. Vergleiche hierzu die Informationen zum »Statuswechsel« bei der *invalid task segment exception* #TS auf Seite 511.

General Protection Die #GP ist die bei allen Usern »beliebteste« Exception, die hinter der lapidaren Meldung »Allgemeine Schutzverletzung in Modul ...« steht und so »schön aussagekräftig« ist. Der Grund hierfür ist, dass für eine #GP tatsächlich eine große Anzahl Ursachen aus den unterschiedlichsten Themenbereichen verantwortlich zeichnen können:

- 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 Datensegment, das als »read-only« markiert ist, oder lesender Zugriff auf ein Codesegment, das als »execute-only« deklariert wurde
- Das Laden des SS-Registers mit einem Selektor für ein Segment mit Attribut »read-only« oder »execute-only« oder mit einem Nullsegment

- Das Laden von DS, ES, FS, GS oder SS mit Selektoren für Systemsegmente, das Laden von DS, ES, FS oder GS mit Selektoren für Segmente mit Attribut »read-only« oder das Laden von CS mit Selektoren auf Datensegmente oder ein Nullsegment
- Der Zugriff auf Speicher, wenn DS, ES, FS oder GS einen Nullselektor beinhalten
- Das Umschalten auf einen als »busy« markierten Task im Rahmen eines CALLs oder JMPs mit einem TSS als Ziel oder während der Rückkehr zu einem als nicht »busy« markierten Task im Rahmen eines IRETs.
- Das Benutzen eines Segment-Selektors bei einem Task-Switch, der auf einen TSS-Deskriptor in der aktuellen LDT zeigt. (TSS müssen global verfügbar sein und ihre Deskriptoren können daher nur in der GDT angesiedelt werden!)
- Jegliche Verletzung der Schutzkonzepte (vgl. Seite 467).
- 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 (im Rahmen eines Interrupts), 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 reserviertes Bit im Kontrollregister CR4 oder einem machine specific register (MSR) zu setzen
- Der Versuch, einen privilegierten Befehl auszuführen, wenn der CPL nicht 0 ist
- Zugriff auf ein Gate, das einen Nullselektor enthält
- Aufruf eines Interrupts, wenn der CPL größer als der DPL des benutzten interrupt, trag oder task gate ist
- Wenn der Segment-Selektor in einem call, interrupt oder trap gate nicht auf ein Codesegment zeigt

- Wenn der einem LLDT- oder LTR-Befehl übergebene Selektor auf eine lokale Deskriptoren-Tabelle zeigt (TI-Flag gesetzt; LDT- und TSS-Deskriptoren müssen global verfügbar sein und daher in der GDT stehen!) oder nicht auf ein Segment vom Typ LDT bzw. verfügbares TSS zeigt
- Wenn bei einem FAR CALL, FAR JMP oder FAR RETURN ein Nullselektor als Operand übergeben wird
- Wenn das PAE- und/oder PSE-Flag in Kontrollregister CR4 gesetzt ist (36-Bit-Adressierung!) und der Prozessor irgendein reserviertes Bit in einem *page directory pointer table entry* gesetzt vorfindet
- Der Versuch, reservierte Bits im MSCX-Register zu setzen
- Die Ausführung eines SSE- oder SSE2-Befehls, der die Ausrichtung eines Speicheroperanden an 16-Bit-Grenzen fordert, mit einem Speicheroperanden, der eine nicht ausgerichtete 128-Byte-Speicherstelle adressiert.

Interrupt Interrupt-Nummer: 13 ($0D)

Quelle Interruptquelle: CPU

Klasse Klasse: fault

Typ Typ: contributory

ErrorCode ErrorCode:Es wird in jedem Fall ein ErrorCode gemäß Abbildung 2.44 auf Seite 496 auf dem Stack abgelegt. Wurde die Exception im Rahmen des Ladens eines Segment-Deskriptors generiert, wird der dazugehörige Selektor in den ErrorCode eingetragen. Andernfalls ist der ErrorCode »0«. Als Selektoren kommen in Betracht: der Selektor auf das Codesegment mit der fehlerhaften Instruktion, der Selektor eines gates, der als Operand einer Instruktion übergeben wurde, der Selektor für ein task state segment im Rahmen eines task switches oder eine Interrupt-Nummer (Zeiger in die IDT).

Rücksprung Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt üblicherweise auf die Instruktion, die die Exception auslöste. Falls aber die Exception im Rahmen eines task switch erfolgte, zeigt die Rücksprungadresse auf den ersten Befehl des neuen Tasks.

Statuswechsel 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. Vergleiche hierzu die Informationen zum »Statuswechsel« bei der *invalid task segment exception* #TS auf Seite 511.

Bemerkungen

Wenn die Startadresse (das ist die Adresse des ersten, niedrigstwertigen Bytes) eines Operanden für Fließkomma-Berechnungen außerhalb der Grenzen des Segmentes liegt, wird keine Fließkomma-Exception #MF oder #XF ausgelöst, sondern eine #GP. Ein Handler hat dies zu berücksichtigen.

Page Fault

Bei eingeschaltetem Paging-Mechanismus (PG in Kontrollregister CR0 ist gesetzt) zeigt das Auftreten einer #PF an, dass eine physikalische Adresse aus einer virtuellen Adresse nicht berechnet werden konnte. Als Ursachen kommen in Betracht:

- Das present flag P in einem zur Berechnung erforderlichen page directory entry oder einem page table entry ist gelöscht und signalisiert damit, dass die betreffende page table oder page derzeit nicht im Speicher vorliegt.
- Der ausgeführte Code hat nicht die zu einem Zugriff auf die betreffende page erforderlichen Zugriffsrechte. Dies ist der Fall, wenn aus einem im user mode (CPL = 3) laufenden Code auf eine supervisor mode page zugegriffen werden soll.
- Der im user mode ausgeführte Code versucht, auf eine »read-only« page zu schreiben.
- Ein im supervisor mode laufender Code versucht, auf eine »read-only« page im user mode zu schreiben, während das Flag WP in Kontrollregister CR4 gesetzt ist.
- Ein oder mehrere reservierte Bits in einem page directory entry ist gesetzt.

Interrupt-Nummer:	14 ($0E)	*Interrupt*
Interruptquelle:	CPU	*Quelle*
Klasse:	fault	*Klasse*
Typ:	page fault	*Typ*

ErrorCode ErrorCode: Die CPU versorgt den page fault handler mit zwei Informationsquellen:

- Es wird ein ErrorCode gemäß Abbildung 2.45 auf Seite 497 auf dem Stack abgelegt. Einzelheiten zu den Flags siehe dort.
- Kontrollregister CR2. Die CPU lädt die virtuelle Adresse, die zur Auslösung der Exception führte, in dieses Register. Der Handler ist somit in der Lage, die physikalische zu berechnen und die erforderlichen pages nachzuladen.

Rücksprung Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt üblicherweise auf die Instruktion, die die Exception auslöste. Falls aber die Exception im Rahmen eines task switch erfolgte, zeigt die Rücksprungadresse auf den ersten Befehl des neuen Tasks.

Statuswechsel Üblicherweise verursacht die Exception keine Veränderung des Programmzustandes, da der auslösende Befehl nicht ausgeführt wird. Daher kann ein Handler die Ursachen der Exception beseitigen (z.B. die nicht vorhandene page nachladen) und die Programmausführung fortsetzen.

Hat sich die Exception während eines Task-Switchs ereignet, kann sich der Zustand des Programms wie folgt ändern. Ursache für eine #PF während eines task switches kann sein:

- Schreiben des task state des aktuellen tasks in sein TSS, das aufgrund einer ausgelagerten page jedoch nicht verfügbar ist.
- Auslesen der GDT, um das TSS des neuen tasks zu eruieren. Der zur TSS gehörige Deskriptor liegt in einer page, die derzeit nicht verfügbar ist.
- Das neue TSS selbst liegt auf einer ausgelagerten page und ist daher nicht verfügbar.
- Auslesen des neuen TSS. Einer oder mehrere Selektoren in diesem TSS zeigen auf Deskriptoren in pages, die nicht verfügbar sind.
- Auslesen der LDT des neuen tasks zur Verifizierung der Segmente in den Segmentregistern des neuen TSS.

Die letzten beiden Fälle spielen sich bereits im Kontext des neuen tasks ab. Vergleiche hierzu die Informationen zum »Statuswechsel« bei der *invalid task segment exception* #TS auf Seite 511.

Kompatibilität

Beim 80286 und 80386 wurde keine #PF ausgelöst, wenn supervisor-mode code auf »read-only« markierte user-mode pages schreibend zugreifen wollte.

Das Flag RSVD im ErrorCode existiert erst ab dem Pentium, da das Flag PSE in Kontrollregister CR4 mit dem Pentium eingeführt wurde, das PAE-Flag mit der P6-Familie. Diese beiden Flags steuern die Adressberechnung mit mehr als 32 Bits, die eine etwas modifizierte Art der Interpretation einer virtuellen Adresse erforderlich machen (vgl. »Paging: Von der virtuellen zur physikalischen Adresse« auf Seite 441). Im Rahmen dieser Umstellung gelten bei gesetztem PAE- oder PSE-Flag einige Bits in page directory oder page table entries als reserviert, weshalb eine Prüfung auf die unberechtigte Veränderung solcher Bits und die daraus resultierende Auslösung einer #PF auch erst mit der Einführung dieser Flags, also ab dem Pentium erforderlich war. Zuvor galt das Flag RSVD im ErrorCode als reserviert und war auf 0 gesetzt.

Interrupt $0F

Der Interrupt mit der Nummer $15 ist reserviert und nicht dokumentiert. Er sollte nicht benutzt werden.

Interrupt

Interrupt-Nummer: 15 ($0F)

Floating-Point Error

Eine #MF (math fault) zeigt an, dass die FPU einen Fehler bei der Bearbeitung von Fließkomma-Zahlen festgestellt hat. Es können sechs verschiedene Arten von FPU-Exceptions auftreten:

- invalid operation (#I)
- divide by zero (#Z)
- denormalized operand (#D)
- numeric overflow (#O)
- numeric underflow (#U)
- inexact result (#P, precision)

Einzelheiten zu diesen Interrupts werden im nächsten Kapitel dargestellt. Wenn die FPU eine Ausnahmesituation feststellt, gibt es zwei Möglichkeiten:

- Das korrespondierende Maskenbit im control word der FPU ist gelöscht, die entsprechende numerische Exception also unmaskiert. Nur in diesem Fall wird der CPU eine Ausnahmesituation berichtet und die #MF ausgelöst.
- Das Maskenbit ist gesetzt, die numerische Exception somit maskiert. In diesem Fall behandelt die FPU die Exception selbst, indem sie

eine für diesen Fall vorgesehene Standardbehandlung durchführt. Die CPU erhält keine Meldung, eine #MF wird nicht ausgelöst.

Interrupt Interrupt-Nummer: 16 ($10)

Quelle Interruptquelle: CPU in Verbindung mit FPU

Klasse Klasse: fault

Typ Typ: benign

ErrorCode ErrorCode: keiner. Der Grund für die Exception kann aus dem status word der FPU festgestellt werden.

Rücksprung Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf die FPU- oder WAIT/FWAT-Instruktion, die für die Auslösung der #MF zuständig ist. Dies ist somit *nicht* notwendigerweise der Befehl, der zu der Exception führte. Dieser kann, da FPU und CPU durchaus unabhängig voneinander arbeiten, im Befehlsstrom »sehr weit hinten« liegen! Er kann jedoch der FPU-Umgebung entnommen werden, da diese die aktuellen FPU-Register-Inhalte im Falle einer Exception nicht verändert, insbesondere nicht den last instruction pointer, last data pointer und das Register Opcode. Durch diese Register lässt sich der die Exception auslösende Befehl identifizieren.

Statuswechsel Da FPU und CPU weitgehend unabhängig voneinander agieren können und lediglich über WAITs/FWAITs synchronisiert werden, hat zum Zeitpunkt der Exception mit großer Wahrscheinlichkeit eine Programmzustandsänderung stattgefunden. Dies ist jedoch nicht relevant, solange eine Bedingung erfüllt ist: Die bereits abgearbeiteten CPU-Befehle dürfen nicht abhängig sein vom Ergebnis der zur Exception führenden FPU-Instruktion. (Was bei genauerer Betrachtung ja auch logisch ist. Ist der Wechsel im Programmstatus im Exceptionfall tatsächlich ein Problem, so wäre das Problem in dem Fall, dass die Exception nicht aufgetreten wäre, weitaus gravierender: Dann nämlich hätte die CPU mit Daten gearbeitet, die zu diesem Zeitpunkt von der FPU noch nicht berechnet worden sind. Und dies würde bedeuten: An dieser Stelle fehlt eine CPU-FPU-Synchronisierung durch ein WAIT/FWAIT!)

Ist die Bedingung also erfüllt, kann trotz der Änderung des Programmstatus der Programmablauf ohne Bedenken wieder aufgenommen werden, sobald die Ursache der FPU-Exception beseitigt wurde. Dies erfolgt durch den MF-Exceptionhandler, der die erforderlichen Informationen aus den FPU-Registern entnehmen kann. Diese waren ja durch die Exception »eingefroren« worden.

Ist die Bedingung jedoch nicht erfüllt, so ist dies Anlass dazu, vor der Instruktion, die vom FPU-Ergebnis abhängig ist, in den Befehlsstrom ein WAIT/FWAIT einzustreuen.

Bemerkungen

Solange eine #MF anhängig (nicht bearbeitet) ist, wird keinerlei weitere FPU-Instruktion durchgeführt, selbst wenn die CPU in der Befehlsverarbeitung fortfährt. Dies erfolgt solange, bis mittels WAIT/FWAIT oder einer »wartenden« FPU-Instruktion eine Synchronisation CPU – FPU durchgeführt und damit die #MF behandelt wurde.

Damit eine #MF ausgelöst werden kann, sind folgende Voraussetzungen erforderlich:

- das Flag NE (numeric exceptions) in Kontrollregister CR0 muss gesetzt sein
- das zum Exceptiongrund gehörende »Maskenbit« im control word der FPU muss gelöscht sein, die FPU-Exception, die die #MF veranlasst, also nicht maskiert.

Ist das der Fall, und tritt dann eine numerische Ausnahmesituation auf, so agiert die FPU wie folgt:

- das korrespondierende exception flag wird im status word der FPU gesetzt
- sie wartet, bis eine WAIT/FWAIT- oder eine »wartende« FPU-Instruktion (FCLEX, FINIT, FSAVE, FSTCW, FSTENV, FSTSW, nicht aber FXSAVE) im Befehlsstrom aufgefunden wird.
- sie erzeugt ein internes Interruptsignal, das die CPU veranlasst, eine #MF auszulösen.

Die Prüfung auf anhängige Exceptions unterbleibt bei allen FPU-Befehlen, die keine »wartende« Version besitzen, sowie bei dem »nicht wartenden« Teil der in beiden Versionen vorkommenden Instruktionen (FNCLEX, FNINIT, FNSAVE, FNSTCW, FNSTENV, FNSTSW und FXSAVE).

Im virtual 8086 mode kann der V86-Monitor dahingehend programmiert werden, verschiedene Nummern für den Interruptvektor für Fließkomma-Exceptions zu akzeptieren. Im real mode und protected mode dagegen muss es die Nummer 16 sein, die für den Aufruf des Exception-Handlers verantwortlich ist.

Alignment Check

Eine #AC wird ausgelöst, wenn der *alignment check aktiviert* wurde (s.u.) und ein nicht ausgerichtetes (»aligned«) Datum vorgefunden wurde. Ein Datum gilt als ausgerichtet, wenn das erste (»least significant«) Byte, also das Byte mit dem niedrigstwertigen Bit, an einer Adresse liegt, die restlos durch einen Divisor teilbar ist, der identisch mit der Größe des Datums in Bytes ist oder, falls ein Datum eine »Struktur« aus zusammengesetzten Daten ist wie z.B. bei Far-Pointern, mit der Größe des Datums der kleinsten Strukturkomponente. Dies wird in Tabelle 2.4 zusammengefasst.

Datum vom Typ	Größe	Divisor
Byte (ShortInt)	1	1
Word (SmallInt)	2	2
DoubleWord (LongInt)	4	4
QuadWord (QuadInt)	8	8
OctelWord (DoubleQuadWord, OctelInt)	16	16
SingleReal	4	4
DoubleReal	8	8
ExtendedReal	10	8 *)
Segment-Selector	2	2
32-Bit-Far-Pointer (16-Bit-Selektor und 16-Bit-Offset = 2 · Word)	4	2
48-Bit-Far-Pointer (16-Bit-Selektor und 32-Bit-Offset = 3 · Word)	6	2
Deskriptoren (Inhalte der nicht sichtbaren Teile des GDTR, IDTR, LDTR, TR, der Deskriptortabellen oder der Segmentregister bestehend aus zwei DoubleWords)	8	4
Speicherort für F(N)STENV/FLDENV (abhängig von operand size)	28/14	4/2
Speicherort für F(N)SAVE/FRSTOR (abhängig von operand size)	108/94	4/2
Bit Strings (abhängig von operand size)		4/2

*) Obwohl eine ExtendedReal nicht aus einzelnen Komponenten aufgebaut ist und 10 Bytes Umfang hat, gilt sie als ausgerichtet, wenn ihre Adresse an QuadWord-Grenzen ausgerichtet ist. Das bedeutet aber auch, dass z.B. vier ausgerichtete, aufeinander folgende ExtendedReals 3 · 6 = 18 Byte Speicherplatz (31%) verschwenden.

Tabelle 2.4: Ausrichtung von Daten

Ein *alignment check* wird aktiviert, indem

- das AM Flag in Kontrollregister CR0 gesetzt wird
- das AC Flag in EFlags gesetzt wird
- der CPL auf 3 gesetzt wird.

Interrupt-Nummer:	17 ($11)	*Interrupt*
Interruptquelle:	CPU	*Quelle*
Klasse:	fault	*Klasse*
Typ:	benign	*Typ*
ErrorCode:	Es wird der Wert »0« als ErrorCode auf den Stack gelegt.	*ErrorCode*

Rücksprung

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt die Instruktion, die die Exception auslöste.

Statuswechsel

Der Programmzustand wird nicht geändert, da die ungültige Instruktion nicht ausgeführt wird.

Bemerkungen

#AC ist nur dafür zuständig, die fehlende Ausrichtung an Word-(2-Byte-)-, DoubleWord-(4-Byte-)- und QuadWord-(8-Byte-)Grenzen zu signalisieren. Falls 128-Bit-(= 16-Byte-)Daten ausgerichtet werden müssen, hat das an 16-Byte-Grenzen (»Paragraphen-Grenzen«) zu erfolgen. Sollten solche Daten nicht ausgerichtet sein, wird eine #GP ausgelöst, keine #AC!

#ACs werden nur im Usermodus (Privilegstufe 3) ausgelöst! Daher erfolgt bei Zugriffen auf Daten in Segmenten höherer Privilegstufen (z.B. Deskriptorentabellen mit Privilegstufe 0) auch dann keine #AC, wenn diese nicht ausgerichtet sind (Quelle hat Privilegstufe 0). Umgekehrt aber führt die schreibende Veränderung solcher Daten oder das Beschreiben der GDTR, IDTR, LDTR und TR aus dem Usermodus ggf. sehr wohl zu einer #AC (Quelle hat Privilegstufe 3).

FXSAVE und FXRSTOR verwenden einen 512-Byte-Operanden, der an einer Paragraphengrenze (16 Bytes) ausgerichtet sein muss. Ein alignment check im Usermodus (CPL = 3) kann nun je nach Implementation zu einer #GP oder einer #AC führen.

MOVUPS und MOVUPD benutzen 128-Bit-(= 16-Byte-)Daten und wären daher im Falle unausgerichteter Operanden Kandidaten für eine #GP (s.o.), erzeugen diese jedoch nicht (»mov unaligned«). Das bedeutet aber nicht, dass keine Prüfung auf Ausrichtung erfolgte: Ist der alignment check aktiviert, wird eine #AC ausgelöst, wenn der Operand nicht an einer Word-, DoubleWord- oder QuadWord-Grenze ausgerichtet ist.

Machine Check Der Prozessor oder ein externer *agent* hat einen internen Fehler (machine error oder bus error) entdeckt und signalisiert mit #MC einen nicht behebbaren Grund zum Programmabbruch.

Interrupt	Interrupt-Nummer:	18 ($12)
Quelle	Interruptquelle:	CPU
Klasse	Klasse:	abort
Typ	Typ:	benign
ErrorCode	ErrorCode:	keiner; die Information wird in den machine check MSR (machine specific registers) zur Verfügung gestellt.

Rücksprung Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt, implementationsbedingt, nicht notwendigerweise auf den Befehl, der die Exception verursachte.

Statuswechsel Die Exception geht in jedem Fall mit einer Veränderung des Programmstatus einher, weshalb sie auch zur Klasse abort gehört, da eine Programmfortführung nicht möglich ist.

Bemerkungen Die #MC ist modellspezifisch! Die Implementation bei Pentium, der P6-Familie, beim Pentium 4 oder folgenden Prozessoren ist unterschiedlich und ggf. nicht kompatibel!

SIMD Floating-Point Eine #XF zeigt an, dass die CPU eine Ausnahmesituation bei einer Fließkommaberechnung im Rahmen der SIMD-Erweiterungen (SSE- und SSE2-Befehle) vorgefunden hat. Es können sechs verschiedene Arten von SIMD-Fließkomma-Exceptions auftreten:

- invalid operation (#I)
- divide by zero (#Z)
- denormalized operand (#D)
- numeric overflow (#O)
- numeric underflow (#U)
- inexact result (#P, precision)

Die Exceptions sind denen, die bei Fließkomma-Exceptions der FPU (#MF) auftreten können, sehr ähnlich. Einzelheiten zu diesen Interrupts werden im nächsten Kapitel dargestellt. Wenn die CPU eine Ausnahmesituation feststellt, gibt es zwei Möglichkeiten:

- Das korrespondierende Maskenbit im MXCSR ist gelöscht, die entsprechende numerische Exception also unmaskiert. Nur in diesem

Fall wird eine #XF ausgelöst, allerdings nur, wenn das Betriebssystem einen entsprechenden Handler zur Verfügung stellt. Signalisiert wird dies durch ein gesetztes Flag OSXMMEXCEPT im Kontrollregister CR4.

- Das Maskenbit im MXCSR ist gesetzt, die numerische Exception somit maskiert. In diesem Fall behandelt die CPU die Exception selbst, indem sie eine für diesen Fall vorgesehene Standardbehandlung durchführt. Eine #XF wird nicht ausgelöst.

Interrupt-Nummer:	19 ($13)	*Interrupt*
Interruptquelle:	CPU	*Quelle*
Klasse:	fault	*Klasse*
Typ:	benign	*Typ*
ErrorCode:	keiner. Der Grund für die Exception kann aus dem MXCSR festgestellt werden.	*ErrorCode*

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt die SSE- oder SSE2-Instruktion, die die Exception auslöste. *Rücksprung*

Eine Änderung des Programmstatus nach einer #XF erfolgt nicht, da der Befehl, der die Exception auslöste, nicht ausgeführt wird. Im Allgemeinen ist dann die Information, auf die der Exception-Handler zurückgreifen kann, ausreichend, um den Fehler zu beseitigen und den Programmablauf gefahrlos wieder aufnehmen zu lassen. *Statuswechsel*

Obwohl #XF von ihrer Ursache her eine große Ähnlichkeit zu #MF haben, gibt es dennoch große Unterschiede: *Bemerkungen*

- #MF werden von der CPU generiert, nachdem die FPU eine numerische Exception bei der Bearbeitung eines FPU-Befehls gefunden hat. Das bedeutet: Die Auslösung einer #MF ist abhängig von der Kommunikation CPU – FPU. SIMD-Befehle dagegen werden von der CPU selbst ausgeführt, weshalb #XF exceptions auch unmittelbar von der CPU ausgelöst werden.
- Aufgrund der in weiten Strecken unabhängigen Befehlsverarbeitung von CPU und FPU kann die Ursache einer #MF und ihre Auslösung sehr weit auseinander liegen. Kriterium ist, dass ein WAIT/FWAIT oder eine »synchronisierende« FPU-Instruktion ausgeführt wird oder ein task switch erfolgt. Da SIMD-Befehle im Befehlsstrom der CPU abgearbeitet werden, werden Ausnahmesituationen sofort und unmittelbar erkannt und entsprechend eine #XF ausgelöst.

- #MF können nur bei der numerischen Bearbeitung eines einzigen Datums auftreten. Das bedeutet, dass jede #MF zu genau einer FPU-Instruktion und einem bearbeiteten Datum gehört. SIMD bedeutet *single instruction on multiple data*. Das bedeutet, dass hier mit einer Instruktion mehrere voneinander unabhängige (= »gepackte«) Daten bearbeitet werden. Eine #XF ist damit so lange »eindeutig«, wie eine Ausnahmesituation einem Datum zugeordnet werden kann. So verhindert z.B. eine invalid operand exception für Teildatum1 nicht, dass eine divide by zero exception für Teildatum2 berichtet und in einer #XF zusammengefasst wird. Wenn dagegen für ein und dasselbe Teildatum mehr als ein Exceptiongrund vorliegt, wird lediglich eine Exception für dieses Teildatum berichtet. Dies erfolgt anhand einer Prioritätsliste:

Priorität	Beschreibung
1 (höchste)	#I aufgrund einer sNaN als Operand oder, bei Minimum-Maximum-Berechnungen oder verschiedenen Vergleichen, jeder NaN als Operand
2	qNaN als Operand. Obwohl eine qNaN keine Exception darstellt, hat die Behandlung von qNaNs Priorität vor verschiedenen Exception. Beispiel: eine qNaN durch 0 dividiert resultiert in einer qNaN, nicht einer #Z
3	alle weiteren #I oder eine #Z
4	#D
5	#U, #O, ggf. zusammen mit #P
6 (niedrigste)	#P

  Selbstverständlich betrifft dies nur unmaskierte Exceptions.

Reserved Die mit den Nummern 20 bis 32 belegten Interrupts gelten als reserviert und dienen Intel dazu, Erfordernisse von künftigen Prozessoren zu bedienen.

Interrupt Interrupt-Nummer: 20 ($14) bis 31 ($1F)

Interrupts Die Interrupts mit den Nummern 32 bis 255 nennt man Softwareinterrupts, da sie von der Software gezielt aufgerufen werden, um bestimmte Dienstleistungen zu erbringen.

Interrupt Interrupt-Nummer: 32 ($20) bis 255 ($FF)

Quelle Interruptquelle: Software

ErrorCode ErrorCode: keiner

Der auf den Stack gesicherte Inhalt des *instruction pointers* in CS:(E)IP (»Rücksprungadresse«) zeigt auf die Instruktion, die dem INT-Befehl folgt.

Rücksprung

2.5.6 FPU-Exceptions

Zwischen FPU-Exceptions und CPU-Exceptions gibt es, abgesehen von der Ursache der Exception und dem Kontext, in dem sie erfolgt, einen gravierenden Unterschied: Bei CPU-Exceptions entdeckt die CPU bei der Bearbeitung einer Instruktion, dass etwas »nicht stimmt«, unterbricht ihre Tätigkeit und widmet sich dann zunächst der Ursachenanalyse und schließlich der Bearbeitung der Ausnahmesituation.

Bei FPU-Exceptions ist das nicht der Fall. Hier bemerkt die FPU bei der Bearbeitung eines ihrer Befehle, dass eine Ausnahmesituation eingetreten ist. Sie unterbricht ebenfalls ihre Tätigkeit, analysiert die Ursache und stellt die Ausnahme in Form von Signalen dar. Doch kann sie die Exception-Behandlung nicht vornehmen. Dies ist im alleinigen Verantwortungsbereich der CPU! Daher muss sie der CPU »melden«, dass bei der Bearbeitung der Fließkomma-Befehle eine Exception eingetreten ist.

Nun ist ein Fehler bei der FPU für die CPU weit weniger wichtig als z.B. die Kontrolle der Peripherie. Das bedeutet, dass sie weder einen eigenen PIN besitzt, der analog der NMI#- oder INTR#-Pins das augenblickliche Reagieren der CPU veranlasst, falls eine FPU-Exception aufgetreten ist, sondern lediglich einen, über den sie den aktuellen Zustand der FPU abfragen kann. Während also ein *non-maskable interrupt* (NMI) oder ein *interrupt request* (INTR) der *programmable interrupt controller* (PICs) die CPU sofort auf den Plan ruft, die Exception zu behandeln, muss sie im Falle von FPU-Exception darum gebeten werden.

Dies erfolgt durch einen einzigen Mechanismus: Das Ausführen einer Exception-meldenden (= »synchronisierenden«) FPU-Instruktion oder eines WAIT/FWAIT-Befehls. Trifft die CPU im Befehlsstrom auf eine solche Instruktion, so prüft sie (über ihre »Standleitung«) das ES-Flag im status register der FPU. Ist dieses exception summary flag gesetzt, liegt mindestens eine (unmaskierte) FPU-Exception vor, die zu bearbeiten ist. In diesem Fall löst die CPU eine #MF aus und ruft in ihrem Verlauf den Exception-Handler auf, der sich mit FPU-Exceptions zu befassen hat. Konsequenz: Keine synchronisierende Instruktion – keine #MF!



Welche FPU-Befehle »synchronisierende« Befehle sind, können Sie der Besprechung der einzelnen Befehle in Band 2, Die Assembler-Referenz, entnehmen. Hier ist für jeden Befehl in Abschnitt Exceptions verzeichnet, ob er eine #MF auszulösen vermag oder nicht. Im Prinzip sind dies die meisten arithmetischen FPU-Befehle sowie die »wartenden« System-Befehle der FPU.



Dieses Zusammenspiel hat eine wichtige Konsequenz: Wenn die FPU auf Daten zugreifen soll, die auch von der CPU verändert werden (könnten), oder umgekehrt die CPU abhängig ist von Daten, die die FPU berechnet hat, müssen Mechanismen benutzt werden, CPU und FPU zu koordinieren und ihre Aktivitäten zu synchronisieren! Dies ist vor allem bei den Speicherbefehlen der FPU wichtig!

Synchronisierung

Eine solche Synchronisation kann sinnvollerweise nur via WAIT/FWAIT erfolgen. Zwar kann auch jede andere »synchronisierende« Instruktion verwendet werden; doch muss in diesem Fall darauf geachtet werden, dass nun nicht diese Funktion zu analogen Problemen führt.

Das bedeutet, dass immer dann ein WAIT/FWAIT in den Befehlsstrom einzufügen ist, wenn CPU und FPU mit den gleichen Daten arbeiten. So muss nach einem FPU-Befehl, der ein Ergebnis erbracht hat, das die CPU irgendwie weiterverwenden soll, unmittelbar nach dem FPU-Befehl ein WAIT eingestreut werden. Dies hat zur Folge, dass die CPU prüft, ob das Ergebnis der vorangegangenen FPU-Instruktion gültig ist und das Datum weiterverwendet werden kann.



Besondere Vorsicht ist auch bei verschiedenen FPU-System-Befehlen geboten! So gehören alle »nicht-wartenden« Versionen dieser Befehle (FNCLEX, FNINIT, FNSTCW, FNSTENV, FNSTSW, FNSAVE, FXSAVE) zu den nicht-synchronisierenden Befehlen, lösen somit keine Exception aus! FNCLEX, FNINIT, FNSTENV und FNSAVE löschen im Rahmen ihrer Aktivitäten alle Exception-Bits oder maskieren alle Exceptions. Lediglich FNSTCW und FNSTSW lassen wartende Exceptions stehen, sodass sie durch einen nachfolgenden synchronisierenden Befehl bearbeitet werden können. Bei allen anderen Befehlen sind die Ausnahmesituationen unrettbar verloren.

Besonders gravierend macht sich das bemerkbar, wenn mit den korrespondierenden Ladebefehlen die ehemalige Umgebung wieder hergestellt wird. Denn nun, vor allem bei FRSTOR/FXRSTOR, werden alle FPU-Register restauriert. Das bedeutet, dass die Situation wiederherge-

stellt wird, die bei der Exception herrschte, ohne dass jedoch die Exception bearbeitet worden wäre.



Und in einem weiteren Detail unterscheiden sich CPU- und FPU-Exceptions: Während bei CPU-Exceptions jeder Exceptiongrund seine eigene Exception und somit seinen Exception-Handler hat, werden alle FPU-Exceptions in einer CPU-Exception #MF zusammengefasst. Es müssen somit Flags eingebunden werden, die dem Handler signalisieren, welche FPU-Exception er zu behandeln hat. Bitte beachten Sie, dass diese Flags im status word der FPU »sticky« sind. Das bedeutet, sie bleiben solange gesetzt, bis sie explizit gelöscht werden. Das ist insofern von Bedeutung, als beispielsweise Bit 6 des StatusWord, SF (stack fault), angibt, ob ein *stack fault* Ursache für eine *invalid operation exception* #I war oder nicht. Wurde nun durch einen vorangegangene stack fault das Bit gesetzt, durch den Exception-Handler jedoch nicht zurückgesetzt, so ist es bei der nächsten #I immer noch gesetzt. Dies ist dann kein Problem, wenn die Ursache wiederum ein stack fault ist. Doch falls nun ein invalid arithmetic operand die exception verursacht, ergibt sich das Problem: Der Exception-Handler findet ein gesetztes Bit 6 und schließt daher messerscharf und falsch auf einen Stackfehler. Der Stack wird gesund-misshandelt und die eigentliche Ursache bleibt bestehen.

Prioritäten

Falls im Kontext eines Befehls mehrere numerische Exceptions auftreten, bedient sie die CPU anhand einer Prioritätenliste. Sie wird in Tabelle 2.5 angegeben.

Priorität	Beschreibung
1 (höchste)	#I mit der Reihenfolge: #IS (underflow vor overflow!), #IA (nicht unterstützter Operand vor sNaN)
2	qNaN; obwohl eine qNaN keine Exception darstellt, hat ihre Erzeugung jedoch Vorrang vor bestimmten Exceptions. Beispiel: Die Division einer qNaN durch Null führt zu einer qNaN, nicht zu einer #Z
3	alle verbliebenen #IA, danach #Z
4	#D
5	#U und #O, ggf. in Verbindung mit #P
6 (niedrigste)	#P

Tabelle 2.5: Prioritätenliste für die Bearbeitung von numerischen Exceptions

Hierbei sind #I (mit #IA und #IS), #Z und #D »pre-operation exceptions«, was bedeutet, dass sie vor der Ausführung des Befehls entdeckt und ausgelöst werden können. #U, #O und #P dagegen sind »post-ope-

ration« exceptions. Sie können erst auftreten, wenn die Operation bereits durchgeführt wurde. Bei pre-operation exceptions bleibt somit der Zieloperand unverändert und erweckt den Anschein, die Operation sei nicht durchgeführt worden (was ja realiter auch der Fall ist!). Bei post-operation exceptions dagegen kann der Zieloperand sehr wohl verändert worden sein.



Häufig erzeugt die Ausführung eines FPU-Befehls mehr als eine Exception. So kann z.B. die Division einer sNaN durch Null eine invalid operation exception (sNaNs als Operanden) wie auch eine divide by zero exception auslösen. In solchen Fällen kann es sein, dass die Exception mit der höheren Priorität ausgeführt und die mit der niedrigeren ignoriert wird. So würde bei der Division der sNaN durch 0 und jeweils maskierter #I und #Z eine qNaN als Ergebnis der automatisch behandelten #I erzeugt und die #Z unter den Tisch fallen.

Tabelle 2.6 zeigt die möglichen FPU-Exceptions, die im Folgenden genauer beschrieben werden.

Exception	Subtyp	Ursache	Flag	Maske
#I	#IS #IA	FPU-Stack overflow / underflow (SF gesetzt) Invalid operand (SF gelöscht)	IE	IM
#D	-	Denormalized Operand	DE	DM
#Z	-	Divide by zero	ZE	ZM
#O	-	Numeric overflow	OE	OM
#U	-	Numeric underflow	UE	UM
#P	-	Inexact result	PE	PM

Tabelle 2.6: Liste der möglichen FPU-Exceptions

Invalid Operation

Eine #I signalisiert eine Ausnahmesituation aufgrund einer unerlaubten Operation. Anhand der Ursachen einer solchen invalid operation lassen sich zwei »Unterexceptions« definieren:

- stack overflow oder underflow (#IS)
- invalid arithmetic operand (#IA)

Bit 6 des status words gibt im Falle einer #I Auskunft darüber, welcher Subtyp eingetreten ist: Ist Bit 6, SF (stack fault), gesetzt, so hat eine #IS stattgefunden. Ist es dagegen gelöscht, war eine #IA Grund für die Exception.

stack overflow/ underflow

Bei einer #IS zeigt ein gesetztes SF-Flag einen von zwei möglichen Stack-Faults an, der bei verschiedenen Operationen, vor allem Lade- und Speicheroperationen, auftreten kann:

- stack overflow bei einem Versuch, ein Register zu beschreiben, das nicht als *empty* markiert ist, oder
- 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.

Der Begriff stack overflow stammt ursprünglich daher, dass zum Laden eines Wertes auf den FPU-Stack mindestens ein freier Platz auf dem Stack verfügbar sein muss (was gleichbedeutend damit ist, dass der aktuelle TOS leer sein muss, da nur in ihn geladen werden kann!) – andernfalls »liefe der Stack über«. Analog kann von einem Stack, der bereits leer ist, kein weiterer Wert entfernt werden (was wiederum nur aus dem TOS möglich ist, somit darf der nicht leer sein!), der Versuch, das zu tun, führte somit zu einem »Stack-Unterlauf« (blödes Wort, aber ich kenne keines, das prägnanter ist. Außerdem ist es die direkte Übersetzung von *stack underflow*!).

invalid arithmetic operand

Ein gelöschtes SF-Flag zeigt eine Fülle verschiedener Ursachen für einen arithmetischen Fehler an.

- Jede arithmetische Operation mit Operanden, die ein nicht unterstütztes Format besitzen
- Jede arithmetische Operation mit einer signalling NaNs (sNaN) als Quelloperanden
- Die Einbeziehung von NaNs in Test- oder Vergleichsbefehle
- Addition von Unendlichkeiten mit verschiedenen Vorzeichen oder Subtraktion von Unendlichkeiten mit gleichem Vorzeichen
- Multiplikation von 0 mit ∞ oder ∞ mit 0
- Division von ∞ mit ∞ oder 0 mit 0
- Negative Operanden bei FSQRT (Ausnahme: FSQRT(-0) := -0!) oder FYL2X (Ausnahme: FYL2X(-0) := -∞!) oder ein negativer Operand jenseits von –1 bei FYL2XP1 (was gleichbedeutend ist mit FYL2X < 0!)
- Der Divisor bei FPREM oder FPREM1 ist 0 oder der Dividend ∞

- FCOS, FTAN, FSIN und FSINCOS mit ∞ als Operanden
- Der Quelloperand von FIST oder FISTP enthält einen Wert, der nicht den Wertebereich des Zieloperanden über- oder unterschreitet
- 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

Flag status word: IE (Bit 0)

Maske control word: IM (Bit 0)

Aktion bei Maskierung

#IS:

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.

#IA:

Zunächst wird das IE-Flag gesetzt. (ACHTUNG! Das SF-Flag wird nicht explizit gelöscht!) In Abhängigkeit von den oben genannten Quellen der Exception erfolgt dann je nach Ursache eine der folgenden Aktionen:

Ursache	Aktion
Operation mit Operanden, die nicht das unterstütze Format aufweisen	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Operation mit einer oder mehreren NaNs	Übergabe einer qNaN an den Zielparameter (siehe folgende Tabelle)
Die Einbeziehung von NaNs in Test- oder Vergleichsbefehle	Setzen von C0, C2 und C3 im Statuswort bzw. CF, PF und ZF im EFlags-Register (»nicht vergleichbar«)
Addition von Infiniten mit verschiedenen Vorzeichen oder Subtraktion von Infiniten mit gleichen Vorzeichen	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Multiplikation von 0 mit ∞ oder ∞ mit 0	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Division von ∞ mit ∞ oder 0 mit 0	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)

Ursache	Aktion
Negative Operanden bei FSQRT oder FYL2X oder ein Operand < -1 bei FYL2XP1	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Der Divisor bei FPREM oder FPREM1 ist 0 oder der Dividend ∞.	Löschen von C0 im condition code des status words und Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
FCOS, FTAN, FSIN und FSINCOS mit ∞ als Operanden	Löschen von C0 im condition code des status words und Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Der Quelloperand von FIST bzw. FISTP enthält einen Wert, der den Wertebereich des Zieloperanden über- oder unterschreitet.	Übergabe der *Integer-Indefinite* (vgl. Abbildung 5.13 auf Seite 803.)
Der Quelloperand von FBSTP ist ein leeres Register, enthält eine NaN, ∞ oder einen Wert, der nicht mit 18 Dezimalen dargestellt werden kann.	Übergabe der qNaN *BCD-Indefinite* (vgl. Abbildung 5.22 auf Seite 811.)
Ein oder zwei leere Register bei FXCH	Belegen des/der leeren Register mit der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792), danach Austausch der Register.

Bei NaNs als Operanden erhält der Zieloperand aufgrund maskierter (= automatischer) Behandlung der Exception folgende Werte:

Quell-Operanden	Ziel-Operand
sNaN und qNaN	qNaN aus dem Quelloperanden mit der qNaN
sNaN und sNaN	qNaN aus der sNaN mit der größeren Mantisse *)
qNaN und qNaN	qNaN mit der größeren Mantisse
sNaN und normale Finite	qNaN aus der sNaN *)
qNaN und normale Finite	qNaN aus dem Quelloperanden mit der qNaN
sNaN (Ein-Operanden-Befehl)	qNaN aus der sNaN *)
qNaN (Ein-Operanden-Befehl)	qNaN aus dem Quelloperanden

*) Die Konvertierung der sNaN in die korrespondierende qNaN erfolgt dadurch, dass das M-Bit (also bei ExtendedReals das Bit nach dem J-Bit und bei den anderen Real-Formaten das *most significant* bit der Mantisse) gesetzt wird. Bei sNaNs ist es gelöscht (vgl. »Codierung von Fließkommazahlen« auf Seite 788).

Aktion bei fehlender Maskierung *#IS:*

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.

#IA:

Es wird das IE-Flag gesetzt und der Exception-Handler aufgerufen. Der Zeiger auf den TOS bleibt unverändert, ebenso wie die Inhalte der Register, Ziel- und Quelloperanden. Üblicherweise wird keine #IA ausgelöst, wenn einer oder beide Operanden einer Instruktion eine qNaN und keine sNaN beteiligt ist. Ausnahme: FCOM, FCOMP, FCMPP, FCOMI und FCOMIP. Bei diesen Instruktionen wird eine #IA ausgelöst, sobald einer der Operanden eine qNaN ist..

Bemerkungen Die FPU setzt war explizit das Flag SF (stack fault) im Falle eines stack overflow oder underflow, sie löscht es aber nicht bei einem invalid operand (FPU-Flags sind »sticky«!).

Kompatibilität Falls bei der Ausführung von FSQRT, FDIV, FPREM oder der Konvertierung einer ExtendedReal in eine BCD oder Integer ein denormaler Operand (vgl. »Codierung von Fließkommazahlen« auf Seite 788) auftritt, lösen 32-Bit-FPUs keine #IA aus! In diesem Fall wird der Wert vor der Befehlsausführung normalisiert. Auf 16-Bit-NPXen dagegen wird eine #IA ausgelöst.

Denormalized Operand Eine #D signalisiert den Ausnahmezustand aufgrund der Bearbeitung eines denormalen Operanden. Hierunter versteht man einen Operanden, dessen Wert so klein ist, dass er mit den »normalen« Möglichkeiten der Darstellung einer Realzahl nicht mehr darstellbar ist (vgl. »Codierung von Fließkommazahlen« auf Seite 788). Konkret wird eine #D ausgelöst, wenn

- versucht wird, eine arithmetische Operation mit denormalisierten Operanden durchzuführen, oder
- versucht wird, eine denormalisierte Single- oder DoubleReal, *nicht* aber eine denormalisierte ExtendedReal, in ein Register einzulesen.

Flag status word: DE (Bit 1)

Maske control word: DM (Bit 1)

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 ExtendedReal-Format 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 Single- oder DoubleReal wird ebenfalls das DE-Flag gesetzt, dann die Zahl ins ExtendedReal überführt und dabei normalisiert. (Da diese ExtendedReal eine höhere Genauigkeit mit mehr Nachkommastellen besitzt, kann jede denormalisierte Single- oder Doublereal in eine normalisierte ExtendedReal überführt werden!)

Aktion bei fehlender Maskierung

Es wird das DE-Flag gesetzt und der Exception-Handler aufgerufen. Weitere Aktionen außerhalb des Handlers erfolgen nicht, insbesondere bleiben der Zeiger auf den TOS und die Inhalte der Register und Quelloperanden unverändert. ACHTUNG: Im Falle von Ladeoperationen heißt das, dass der Wert *nicht* geladen wurde!

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!

Kompatibilität

Im Falle einer maskierten Exception normalisieren 32-Bit-FPUs Denormale wann immer möglich. 16-Bit-NPXe dagegen geben ein denormalisiertes Ergebnis zurück! Falls somit Software, die von 16-Bit- auf 32-Bit-Systeme portiert wurde, den Exception-Handler für die #D nur dazu benutzt, denormalisierte Daten zu normalisieren, ist dies bei 32-Bit-FPUs redundant. Die Performance kann in diesem Fall dadurch erheblich gesteigert werden, dass die Exception maskiert wird.

Auf 16-Bit-NPXen wird die #D auch nicht bei transzendentalen Funktionen oder FXTRACT ausgelöst, bei 32-Bit-FPUs dagegen sehr wohl.

Divide by Zero

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 eine #Z aus, wenn versucht wird, einen Nicht-Null-Operanden durch 0 zu dividieren.

Flag status word: ZE (Bit2)

Maske control word: ZM (Bit 2)

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. Weitere Aktionen außerhalb des Handlers erfolgen nicht, insbesondere bleiben der Zeiger auf den TOS und die Inhalte der Register und Quelloperanden unverändert. Die Instruktion wird nicht durchgeführt.

Numeric Overflow 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 ExtendedReal einen maximalen Bereich von $-1.11..11 \cdot 2^{16383}$ bis $+1.11..11 \cdot 2^{16383}$ ($\cong \pm 1{,}18 \cdot 10^{4932}$, siehe »Codierung von Fließkommazahlen« auf Seite 788). 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.

Die Schwellwerte für die #O sind im Falle einer SingleReal $\pm 1.0 \cdot 2^{128}$, im Falle einer DoubleReal $\pm 1.0 \cdot 2^{1024}$ und im Falle der ExtendedReal $\pm 1.0 \cdot 2^{16384}$, je ausschließlich.

Dies kann immer dann erfolgen, wenn eine arithmetische Operation erfolgte oder aber eine ExtendedReal als Single- oder DoubleReal abgespeichert werden soll und deren Wertebereich übersteigt. Diese Exception tritt nicht auf, wenn eine ExtendedReal als Integer oder BCD abgespeichert werden soll, auch dann nicht, wenn der entsprechende Wertebereich überschritten wird. In diesem Fall wird eine #IA ausgelöst.

status word: OE (Bit 3) *Flag*

control word: OM (Bit 3) *Maske*

In jedem Falle wird das OE-Flag gesetzt. Das Ergebnis der Aktion bei einer maskierten #O hängt davon ab, welcher Rundungsmodus eingestellt ist. Betrachtet wird dabei das Vorzeichen des Ergebnisses der Operation, das nicht mehr korrekt dargestellt werden kann. *Aktion bei Maskierung*

RC	Rundung	negatives Vorzeichen *)	positives Vorzeichen *)
00_b	zur nächsten oder ganzen Zahl	-∞	+∞
01_b	in Richtung »minus unendlich«	-∞	+MaxFinite
10_b	in Richtung »plus unendlich«	-MaxFinite	+∞
11_b	Abschneiden der Nachkommastellen	-MaxFinite	+MaxFinite

*) MaxFinite ist die größte Zahl, die als »normale« Zahl im Zielformat darstellbar ist.

Falls der Zieloperand eine Speicherstelle ist, wird das OE-Flag gesetzt und der Exception-Handler aufgerufen. Der TOS und die Quelloperanden bleiben unverändert. Der Handler hat nun die Wahl, entweder das Ergebnis an die Belange des Zieloperanden anzupassen (Stichwort: »saturation«) und den Speichervorgang zu wiederholen oder eine Rundung vorzunehmen, die den Bedürfnissen des Zieloperanden entspricht. In jedem Falle sollte der Handler einen Wert abspeichern! *Aktion bei fehlender Maskierung*

Falls dagegen der Zieloperand ein FPU-Register ist, wird das Ergebnis gerundet und *der Exponent* durch $3 \cdot 2^{13} = 24.576$ dividiert (skaliert), was einer Division des Ergebnisses durch $2^{24.576}$ entspricht, 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. Schließlich 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.

Die Skalierung mit dem Faktor $3 \cdot 2^{13}$ im Falle eines gelöschten OM-Flags als Antwort auf die Exception soll den Exponenten so weit wie möglich »in die Mitte« des definierten Exponentenbereiches bringen. Auf diese Weise können folgende Berechnungen, so sie alle mit dem Faktor ska- *Bemerkungen*

liert werden, weiterhin durchgeführt werden – wobei das Risiko, dass es zu einem weiteren Überlauf kommt, entsprechend geringer ist.

Numeric Underflow

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 ExtendedReal 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.

Die Schwellwerte für die #U sind im Falle einer SingleReal $\pm 1.0 \cdot 2^{-126}$, im Falle einer DoubleReal $\pm 1.0 \cdot 2^{-1022}$ und im Falle der ExtendedReal $\pm 1.0 \cdot 2^{-16382}$, je ausschließlich.

Dies kann immer dann erfolgen, wenn eine arithmetische Operation erfolgte oder aber eine ExtendedReal als Single- oder DoubleReal abgespeichert werden soll.

Flag status word: UE (Bit 4)

Maske control word: UM (Bit 5)

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 dann *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), was einer Multiplikation des Ergebnisses mit $2^{24.576}$ entspricht, 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

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

Die #P, auch inexact result exception genannt, wird immer dann ausgelöst, wenn das Ergebnis einer Operation nicht korrekt im Zielformat dargestellt werden kann. So kann beispielsweise das Ergebnis der Division von 1 durch 3 binär nicht »exakt«, also unter Nutzung der vorgegebenen, beschränkten Anzahl von Mantissenbits ohne Rundung dargestellt werden.

Flag

status word: PE (Bit 5)

Maske

control word: PM (Bit 5)

Aktion bei Maskierung

Zunächst prüft die FPU, ob der Grund für die #P ein *overflow* oder *underflow* war. Ist beides 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 #P ein *overflow* oder *underflow* war. Ist dies nicht der Fall, so wird wie im Falle einer Maskierung weitergemacht: Das PE-Flag wird gesetzt, das Ergebnis gerundet und im Zieloperanden abgelegt. Die im Kontrollwort in den Bits 11 und 10 vorgegebene Rundungsart kommt dabei zum Einsatz. Abschließend wird der Exception-Handler für #P 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 aufgerufen.

Bemerkungen

Falls – maskiert oder nicht! – eine #P in Verbindung mit einer nicht maskierten #O oder #U auftritt und der Zieloperand eine Speicherstelle ist, was nur bei Speicherbefehlen der Fall sein kann, wird die #P ignoriert.

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.

Eine #P tritt aufgrund ihrer Ursachen so häufig auf (welche Realzahl lässt sich schon »exakt« im binären Format darstellen!), dass sie in der Regel maskiert und somit automatisch beantwortet wird. Die Möglichkeit, diese Exception zu demaskieren, besteht daher auch nur im Hinblick auf Programme, die aus welchen Gründen auch immer darauf angewiesen sind, zumindest informiert zu werden, wenn eine exakte Darstellung eines Ergebnisses nicht möglich ist.

2.5.7 SIMD-Realzahl-Exceptions

SIMD-Exceptions lassen sich nur bei Realzahlen feststellen, also bei allen SSE- und SSE2-Befehlen, die mit Realzahlen zu tun haben. Integer-SIMD-Befehle dagegen werden wie CPU-Befehle behandelt und erzeugen daher auch »nur« die üblichen CPU-Exceptions.

SIMD-Exceptions unterscheiden sich von CPU- und FPU-Exceptions in der Weise, dass sie eine Zwitterstellung einnehmen. Zum einen signalisieren sie wie FPU-Exceptions eine Ausnahmesituation, die bei der Bearbeitung von Realzahlen aufgetreten ist. SIMD-Realzahl- und FPU-Exceptions sind insoweit vergleichbar, als sie als identische Exceptions in einem etwas anderen Kontext aufgefasst werden können.



Die Übereinstimmung mit FPU-Exceptions geht so weit, dass auch die SIMD-Realzahl-Exceptions in einer CPU-Exception #XF zusammengefasst werden. Es müssen analog den FPU-Exceptions somit Flags einge-

bunden werden, die dem Handler signalisieren, welche Realzahl-Exception er zu behandeln hat. Dies erfolgt absolut identisch zu FPU-Exceptions. Bitte beachten Sie, dass diese Flags im MXCSR ebenfalls »sticky« sind. Das bedeutet, sie bleiben solange gesetzt, bis sie explizit (durch den Exception-Handler) gelöscht werden.

Andererseits stellt nicht die FPU oder eine FPU-ähnliche Berechnungseinheit die Exception fest, sondern die CPU selbst, da sie die SIMD-Befehle ausführt. Daher ist der Mechanismus, der zur Auslösung einer SIMD-Realzahl-Exception führt, sehr viel einfacher und direkter. Es muss kein WAIT/FWAIT eingestreut werden und es ist auch keine Synchronisierung erforderlich. Die CPU stellt während der Bearbeitung eines SIMD-Befehls eine Ausnahmesituation fest und reagiert unmittelbar durch Aufruf des korrespondierenden Exception-Handlers darauf. Weiterer Aspekt: Es gibt keinen FPU-Stack und somit keine Subtypen einer invalid operation: Es kann sich im Falle der Auslösung einer #I nur um ungültige arithmetische Operanden als Exceptionquelle handeln (#IA bei FPU-Exceptions).

Ein weiterer Unterschied ist, dass SIMD-Daten meistens gepackte Daten sind. Das bedeutet, dass eine Instruktion mehrere Daten betrifft. Somit kann jede Exception entweder eines oder mehrere Daten der gepackten Struktur betreffen.

Dies kann nicht signalisiert werden! Intern verfügt die CPU für jedes Teildatum der Struktur über einen Satz Statusflags, die dazu benutzt werden, die entsprechende Exceptionsituation für das einzelne Datum darzustellen. Zum Berichten »nach draußen« in die Statusflags des MXCSR werden diese Einzelflags aber zu den im Register definierten »Sammelflags« durch eine logische UND-Verknüpfung zusammengefasst.

Das bedeutet: Im Falle nicht maskierter Exceptions hat der Exception-Handler zu prüfen, welches Teildatum für welche Exception verantwortlich ist. So kann z.B. die »untere« DoubleReal eine #Z auslösen, während die »obere« eine #D generiert.



Bei der Generierung und Bearbeitung von SIMD-Realzahl-Exceptions spielt ein weiteres Flag eine bedeutende Rolle: das OSXMMEXCEPT Flag (Bit 10) des Kontrollregisters CR4. Ist dieses Flag gesetzt, stellt das Betriebssystem einen Exception-Handler für SIMD-Exceptions (#XF) zur Verfügung. Andernfalls können SIMD-Realzahl-Exceptions nicht

bearbeitet werden und die CPU generiert bei jedem SIMD-Realzahl-Befehl eine invalid opcode exception #UD.

Prioritäten Falls im Kontext eines Befehls mehrere numerische Exceptions auftreten, bedient sie die CPU anhand einer Prioritätenliste. Sie wird in Tabelle 2.7 angegeben.

Priorität	Beschreibung
1 (höchste)	#I (nicht unterstützter Operand vor sNaN)
2	qNaN; obwohl eine qNaN keine Exception darstellt, hat ihre Erzeugung jedoch Vorrang vor bestimmten Exceptions. Beispiel: Die Division einer qNaN durch Null führt zu einer qNaN, nicht zu einer #Z
3	alle verbliebenen #I, danach #Z
4	#D
5	#U und #O, ggf. in Verbindung mit #P
6 (niedrigste)	#P

Tabelle 2.7: Prioritätenliste für die Bearbeitung von numerischen Exceptions

Hierbei sind #I, #Z und #D »pre-operation exceptions«, was bedeutet, dass sie vor der Ausführung des Befehls entdeckt und ausgelöst werden können. #U, #O und #P dagegen sind »post-operation« exceptions. Sie können erst auftreten, wenn die Operation bereits durchgeführt wurde. Bei pre-operation exceptions bleibt somit der Zieloperand unverändert und erweckt den Anschein, die Operation sei nicht durchgeführt worden (was ja realiter auch der Fall ist!). Bei post-operation exceptions dagegen kann der Zieloperand sehr wohl verändert worden sein.



Häufig erzeugt die Ausführung eines SIMD-Befehls mehr als eine Exception. So kann z.B. die Division einer sNaN durch Null eine *invalid operation exception* (sNaNs als Operanden) wie auch eine *divide by zero exception* auslösen. In solchen Fällen kann es sein, dass die Exception mit der höheren Priorität ausgeführt und die mit der niedrigeren ignoriert wird. So würde bei der Division der sNaN durch 0 und jeweils maskierter #I und #Z eine qNaN als Ergebnis der automatisch behandelten #I erzeugt und die #Z unter den Tisch fallen.

Tabelle 2.8 zeigt die möglichen SSE/SSE2-Exceptions, die im Folgenden genauer beschrieben werden.

Exception	Subtyp	Ursache	Flag	Maske	Einfluss
#I		Invalid operand	IE	IM	-
#D	-	Denormalized operand	DE	DM	DAZ
#Z	-	Divide by zero	ZE	ZM	-
#O	-	Numeric overflow	OE	OM	-
#U	-	Numeric underflow	UE	UM	FZ
#P	-	Inexact result	PE	PM	-

Tabelle 2.8: Liste der möglichen SSE/SSE2-Exceptions

Invalid Operation

Eine #I signalisiert eine Ausnahmesituation aufgrund einer unerlaubten Operation. Dies kann eine Fülle verschiedener Ursachen haben:

- Jede arithmetische Operation mit Operanden, die ein nicht unterstütztes Format besitzen
- Jede arithmetische Operation mit einer signalling NaNs (sNaN) als einer Komponente des/der Quelloperanden
- Die Einbeziehung von NaNs in Vergleichsbefehle
- Addition von Unendlichkeiten mit verschiedenen Vorzeichen oder Subtraktion von Unendlichkeiten mit gleichem Vorzeichen
- Multiplikation von 0 mit ∞ oder ∞ mit 0
- Division von ∞ mit ∞ oder 0 mit 0
- Negative Operanden bei der Quadratwurzelbildung (Ausnahme: -0)
- Maximal- bzw. Minimalwert-Bildung mit NaNs als Operanden
- Konversion einer sNaN in ein Realzahlformat
- Konversion einer NaN oder Infiniten zu einer Integer oder Überschreitung des darstellbaren Wertebereichs bei der Konversion zu einer integer

IE (Bit 0 MXCSR) *Flag*

IM (Bit 7 MXCSR) *Maske*

Aktion bei Maskierung

Zunächst wird das IE-Flag gesetzt. In Abhängigkeit von den oben genannten Quellen der Exception erfolgt dann je nach Ursache eine der folgenden Aktionen:

Ursache	Aktion
Operation mit Operanden, die nicht das unterstützte Format aufweisen	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Operation mit einer oder mehreren NaNs	Übergabe einer qNaN an den Zielparameter (siehe folgende Tabelle)
Die Einbeziehung von NaNs in die Vergleichsbefehle CMPSS, CMPPS, CMPSD und CMPPD	Rückgabe einer Maske mit gelöschten, bei Vergleichstyp »not equal« oder »unordered« mit gesetzten Bits.
Die Einbeziehung von NaNs in die Vergleichsbefehle COMISS und COMISD	Setzen von CF, PF und ZF im EFlags-Register (»nicht vergleichbar«)
Addition von Infiniten mit verschiedenen Vorzeichen oder Subtraktion von Infiniten mit gleichen Vorzeichen	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Multiplikation von 0 mit ∞ oder ∞ mit 0	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Division von ∞ mit ∞ oder 0 mit 0	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
Negative Operanden bei SQRTSS, SQRTPS, SQRTSD und SQRTPD (Ausnahme: -0!)	Übergabe der qNaN *Real-Indefinite* (vgl. Abbildung 5.5 auf Seite 792.)
MAXSS, MAXPS, MAXSD, MAXPD, MINSS, MINPS, MINSD und MINPD mut NaNs als Operanden	Rückgabe des Wertes aus Operand2 (zweiter Quelloperand)
CVTPD2PS, CVTSD2SS, CVTPS2PD und CVTSS2SD mit sNaNs als Operanden	Rückgabe der aus der sNaN gebildeten qNaN (vgl. nächste Tabelle)
CVTPS2PI, CVTTPS2PI, CVTSS2SI, CVTTSS2SI, CVTPD2PI, CVTTPD2PI, CVTSD2SI, CVTTSD2SI, CVTPD2DQ, CVTTPD2DQ, CVTPS2DQ oder CVTTPS2DQ mit einer NaN oder ∞ oder Überschreitung des Wertebereiches	Übergabe der *Integer-Indefinite* (vgl. Abbildung 5.13 auf Seite 803.)

Bei NaNs als Operanden erhält der Zieloperand aufgrund maskierter (= automatischer) Behandlung der Exception folgende Werte:

Quell-Operanden	Ziel-Operand
sNaN und qNaN	Inhalt des ersten Quelloperanden. Ist dies eine sNaN, wird sie zur qNaN konvertiert. *)
sNaN und sNaN	Inhalt des ersten Quelloperanden, zur qNaN konvertiert. *)
qNaN und qNaN	qNaN des ersten Quelloperanden
sNaN und normale Finite	qNaN, konvertiert aus der sNaN *)
qNaN und normale Finite	qNaN aus dem Quelloperanden
sNaN (bei Ein-Operanden-Befehlen)	qNaN, konvertiert aus der sNaN *)
qNaN (bei Ein-Operanden-Befehlen)	qNaN aus dem Quelloperanden

*) Die Konvertierung der sNaN in die korrespondierende qNaN erfolgt dadurch, dass das M-Bit (also bei ExtendedReals das Bit nach dem J-Bit und bei den anderen Real-Formaten das *most significant* bit der Mantisse) gesetzt wird. Bei sNaNs ist es gelöscht (vgl. »Codierung von Fließkommazahlen« auf Seite 788).

Aktion bei fehlender Maskierung

Es wird das IE-Flag gesetzt und der Exception-Handler aufgerufen. Die Inhalte der Register, Ziel- und Quelloperanden bleiben unverändert. Üblicherweise wird die Exception nicht ausgelöst, sobald einer oder mehrere der Quelloperanden qNaNs sind und keiner eine sNaN. Ausnahme: Eine qNaN bei COMISS bzw. COMISD führt in jedem Fall zu einer Exception.

Denormal Operand

Eine #D signalisiert den Ausnahmezustand aufgrund der Bearbeitung eines denormalen Operanden. Hierunter versteht man einen Operanden, dessen Wert so klein ist, dass er mit den »normalen« Möglichkeiten der Darstellung einer Realzahl nicht mehr darstellbar ist (vgl. »Codierung von Fließkommazahlen« auf Seite 788).

DE (Bit 1 MXCSR) *Flag*

DM (Bit 8 MXCSR) *Maske*

DAZ (Bit 6 MXCSR) *Steuerung*

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 ExtendedReal-

Format 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.

Aktion bei fehlender Maskierung Es wird das DE-Flag gesetzt und der Exception-Handler aufgerufen. Weitere Aktionen außerhalb des Handlers erfolgen nicht, insbesondere bleiben der Zeiger auf den TOS und die Inhalte der Register und Quelloperanden unverändert. ACHTUNG: Im Falle von Ladeoperationen heißt das, dass der Wert *nicht* geladen wurde!

DAZ-Flag Mit Hilfe von Bit 6 des MXCSR (DAZ; *denormals are zero*) kann die Auslösung einer #D verhindert werden. Ist DAZ gesetzt, so werden Denormale vor der Durchführung des Befehls in den Wert Null konvertiert, sodass die Grundlage zur Exceptionauslösung auch im Falle einer unmaskierten Exception entfällt. Das Vorzeichen der Null ist das des eigentlichen, denormalisierten Ergebnisses. ACHTUNG: Dieses Verhalten ist nicht IEEE-Standard-754-konform! Das bedeutet, dass unterschiedliche Ergebnisse entstehen können, wenn man die gleichen Operationen im Rahmen von SIMD-Befehlen oder FPU-Befehlen durchführt.

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, oder durch Setzen des DAZ-Flags.

Divide by Zero Alle Befehle, die eine Division durchführen (DIVSS, DIVPS, DIVSD und DIVPD), lösen eine #Z aus, wenn versucht wird, einen Nicht-Null-Operanden durch 0 zu dividieren.

Flag ZE (Bit 2 MXCSR)

Maske ZM (Bit 9 MXCSR)

Aktion bei Maskierung Die CPU 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!).

Aktion bei fehlender Maskierung

Es wird das ZE-Flag gesetzt und der Exception-Handler aufgerufen. Weitere Aktionen außerhalb des Handlers erfolgen nicht, insbesondere bleiben die Inhalte der Register und Quelloperanden unverändert. Die Instruktion wird nicht durchgeführt.

Numeric Overflow

Eine #O-Exception wird immer dann ausgelöst, wenn das gerundete Ergebnis einer Operation den maximal darstellbaren Bereich des Zieloperanden überschreitet. Die Schwellwerte für die #O sind im Falle einer SingleReal $\pm 1.0 \cdot 2^{128}$, im Falle einer DoubleReal $\pm 1.0 \cdot 2^{1024}$, je ausschließlich.

Dies kann immer dann erfolgen, wenn eine arithmetische Operation erfolgte, also nach ADDSS, ADDPS, ADDSD, ADDPD, SUBSS, SUBPS, SUBSD, SUBPD, MULSS, MULPS, MULSD, MULPD, DIVSS, DIVPS, DIVSD, DIVPD, CVTPD2PS, und CVTSD2SS. (CVTPS2PD und CVTSS2SD können diese Exception nicht auslösen, da der Wertebereich einer SingleReal im Wertebereich einer DoubleReal liegt.)

Flag

OE (Bit 3 MXCSR)

Maske

OM (Bit 10 MXCSR)

Aktion bei Maskierung

In jedem Falle wird das OE-Flag gesetzt. Das Ergebnis der Aktion bei einer maskierten #O hängt davon ab, welcher Rundungsmodus (RC, rounding control: Bits 13 und 14 des MXCSR) eingestellt ist. Betrachtet wird dabei das Vorzeichen des Ergebnisses der Operation, das nicht mehr korrekt dargestellt werden kann.

RC	Rundung	negatives Vorzeichen *)	positives Vorzeichen *)
00_b	zur nächsten oder ganzen Zahl	-∞	+∞
01_b	in Richtung »minus unendlich«	-∞	+MaxFinite
10_b	in Richtung »plus unendlich«	-MaxFinite	+∞
11_b	Abschneiden der Nachkommastellen	-MaxFinite	+MaxFinite

*) MaxFinite ist die größte Zahl, die als »normale« Zahl im Zielformat darstellbar ist.

Aktion bei fehlender Maskierung

Das OE-Flag wird gesetzt und der Exception-Handler aufgerufen. Der Inhalt der Quelloperanden sowie des Zieloperanden bleiben unverändert. Der Handler hat nun die Wahl, entweder das Ergebnis an die Belange des Zieloperanden anzupassen (Stichwort: »saturation«) oder eine Rundung vorzunehmen, die den Bedürfnissen des Zieloperanden entspricht.

Numeric Underflow 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! Die Schwellwerte für die #U sind im Falle einer SingleReal $\pm 1.0 \cdot 2^{-126}$, im Falle einer DoubleReal $\pm 1.0 \cdot 2^{-1022}$, je ausschließlich.

Dies kann immer dann erfolgen, wenn eine arithmetische Operation erfolgte, also nach ADDSS, ADDPS, ADDSD, ADDPD, SUBSS, SUBPS, SUBSD, SUBPD, MULSS, MULPS, MULSD, MULPD, DIVSS, DIVPS, DIVSD, DIVPD, CVTPD2PS, und CVTSD2SS. (CVTPS2PD und CVTSS2SD können diese Exception nicht auslösen, da der Wertebereich einer SingleReal im Wertebereich einer DoubleReal liegt.)

Flag UE (Bit 4 MXCSR)

Maske UM (Bit 11 MXCSR)

Steuerung FZ (Bit 15 MXCSR)

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 dann *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 Die CPU setzt das UE-Flag und ruft den Exception-Handler ohne weitere Manipulationen auf. Insbesondere werden dabei die Inhalte der Quell- und Zieloperanden nicht verändert. Im Gegensatz zu der korrespondierenden FPU-Exception erfolgt somit keine Skalierung des Ergebnisses.

FZ-Flag Mit Hilfe von Bit 15 des MXCSR (FZ; *flush to zero*) kann die Auslösung einer #U verhindert werden. Ist FZ gesetzt *und #U maskiert*, so werden Werte, die den kleinsten darstellbaren Wert unterschreiten, in den Wert Null konvertiert. Das Vorzeichen dieser Null ist das des eigentlichen, nicht darstellbaren Ergebnisses. Ferner wird UE und PE gesetzt. Im Falle einer unmaskierten #U hat FZ keine Bedeutung. ACHTUNG: Dieses Verhalten ist nicht IEEE-Standard-754-konform! Das bedeutet, dass unterschiedliche Ergebnisse entstehen können, wenn man die gleichen Operationen im Rahmen von SIMD-Befehlen oder FPU-Befehlen durchführt.

Precision

Die #P, auch inexact result exception genannt, wird immer dann ausgelöst, wenn das Ergebnis einer Operation nicht korrekt im Zielformat dargestellt werden kann. So kann beispielsweise das Ergebnis der Division von 1 durch 3 binär nicht »exakt«, also unter Nutzung der vorgegebenen, beschränkten Anzahl von Mantissenbits ohne Rundung dargestellt werden.

Flag

PE (Bit 5 MXCSR)

Maske

PM (Bit 12 MXCSR)

Aktion bei Maskierung

Zunächst prüft die FPU, ob der Grund für die #P ein *overflow* oder *underflow* war. Ist beides 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.

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 #P ein *overflow* oder *underflow* war. Ist dies nicht der Fall, so wird wie im Falle einer Maskierung weitergemacht: Das PE-Flag wird gesetzt, das Ergebnis gerundet und im Zieloperanden abgelegt. Die im Kontrollwort in den Bits 11 und 10 vorgegebene Rundungsart kommt dabei zum Einsatz. Abschließend wird der Exception-Handler für #P 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 aufgerufen.

Bemerkungen

Diese Exception hat häufig anzutreffende Ursachen; die nicht vollständig darstellbaren Ergebnisse von Brüchen sind nur ein Beispiel. Eine #P tritt aufgrund ihrer Ursachen so häufig auf (welche Realzahl lässt sich schon »exakt« im binären Format darstellen!), dass sie in der Regel maskiert und somit automatisch beantwortet wird. Die Möglichkeit, diese Exception zu demaskieren, besteht daher auch nur im Hinblick auf Programme, die aus welchen Gründen auch immer darauf angewiesen sind, zumindest informiert zu werden, wenn eine exakte Darstellung eines Ergebnisses nicht möglich ist.

2.5.8 Interrupts und Exceptions im Real und Virtual 8086 Mode

Grundsätzlich sind die Interrupt- und Exceptiongründe im real mode und im virtual 8086 mode sehr ähnlich denen, die im protected mode auftreten. Tabelle 2.9 zeigt die bereits in Tabelle 2.2 für den protected mode genannte Liste möglicher Exceptions und Interrupts und gibt an, ob sie im real mode und im v86 mode verfügbar sind.

#		Beschreibung	virtual 8086 mode *)	real mode *)	8086 CPU
0	#DE	Divide Error	ja	ja	ja
1	#DB	Debug	ja	ja	nein
2	-	Non-maskable Interrupt	ja	ja	ja
3	#BP	Break Point	ja	ja	ja
4	#OF	Overflow	ja	ja	ja
5	#BR	Bound Range Exceeded	ja	ja	reserviert
6	#UD	Invalid Opcode	ja	ja	reserviert
7	#NM	Device Not Available	ja	ja	reserviert
8	#DF	Double Fault	ja	ja	reserviert
9	-	FPU Segment Overflow	reserviert	reserviert	reserviert
10	#TS	Invalid TSS	ja	reserviert	reserviert
11	#NP	Segment Not Present	ja	reserviert	reserviert
12	#SS	Stack Segment Fault	ja	ja	reserviert
13	#GP	General Protection	ja	ja	reserviert
14	#PF	Page Fault	ja	reserviert	reserviert
15	-	reserviert	reserviert	reserviert	reserviert
16	#MF	Math Fault (FPU-Exception)	ja	ja	reserviert
17	#AC	Alignment Check	ja	reserviert	reserviert
18	#MC	Machine Check	ja	ja	reserviert
19	#XF	SIMD-Exception (floating point)	ja	ja	reserviert
20-31	-	reserviert	reserviert	reserviert	reserviert
32-255	-	»frei verfügbare« Interrupts	ja	ja	ja

Die grau unterlegten Interrupts/Exceptions gelten als reserviert und sollten nicht benutzt werden. *) Diese Angaben beziehen sich auf den v86 bzw. real mode des Pentium 4. Bitte beachten Sie im Hinblick auf andere Prozessoren die im Kapitel »Historie« auf Seite 874 genannten Zeitpunkte der Implementierung.

Tabelle 2.9: Liste der möglichen Exceptions und Interrupts im virtual 8086 und real mode

Der Vergleich der beiden Tabellen zeigt, dass alle protected mode exceptions und interrupts auch im v86 mode definiert sind. Dem real mode dagegen fehlen die Exceptions, die typisch für den protected mode sind und mit ihm für ihn eingeführt wurden: Task-Management (multi-tasking: #TS), Schutzkonzepte und Speichersegmentierung (#NP) sowie Paging (#PF). Auch ein alignment check macht im real mode wenig Sinn.

Bei etwas genauerer Betrachtung allerdings stellen wir einige Unterschiede fest:

Real Mode Exceptions

Die Exceptions, die im real mode definiert sind, sind Exceptions, die auch im protected mode keinen ErrorCode kennen. Das bedeutet, Real-Mode-Exceptions und ihre Handler kennen keine ErrorCodes.

V86-Mode Exceptiond

Der virtual 8086 mode kategorisiert die Exception und Interrupts in drei Klassen:

- Klasse 1: Alle CPU- und Hardware-generierten Interrupts, inklusive NMIs und INTRs. Sie werden durch die Exception- und Interrupthandler behandelt, die der den V86-Modus einbettende protected mode zur Verfügung stellt. (Bitte beachten Sie, dass der virtual 8086 mode ein Modus ist, der im protected mode abläuft!
- Klasse 2: »Maskierbare« Hardware-Interrupts, die durch die virtual mode extensions verfügbar werden.
- Klasse 3: Alle Software-Interrupts, die durch den INT-Befehl (und seine Sonderformen BOUND, INTO und INT3) ausgelöst werden.

Bei der Frage, wie nun im v86 mode ein Interrupt bzw. eine Exception behandelt wird, spielen auch noch folgende Felder und Flags eine Rolle:

- IOPL (EFlags-Register). Es steuert unter anderem die Nutzung der Flags VIF und VIP, die ihrerseits einen Einfluss auf die Bearbeitung von Klasse-2-Exceptions haben.
- VME (CR4). Es schaltet die virtual mode extensions frei.
- Software interrupt redirection bit map (TSS). Es definiert, ob die Softwareinterrupts durch das Real-Mode-Programm oder die Handler des protected mode behandelt werden.
- VIF und VIP (EFlags). Stellt eine virtuelle Unterstützung für Interrupts zur Verfügung.

In die Interrupt-Verarbeitung greifen auch drei verschiedene Interrupthandler ein:

- die protected mode interrupt and exception handlers. Dies sind die Handler, die der protected mode für die in den vorangehenden Kapiteln geschilderten Interrupts und Exceptions implementiert.
- die virtual 8086 monitor interrupt and exception handlers. Sie gehören zum virtual 8086 monitor, einem Teil des protected mode kernel, der für die Kontrolle des v86 mode zuständig ist. Diese Interrupts werden in der Regel im Rahmen des protected mode handlers für die #GP (general protection exception, Interrupt-Nummer 13) behandelt.
- 8086 program interrupt and exception handler. Das sind die Interrupthandler, die Teil des Real-Mode-8086-Programms sind und ihre eigene interrupt vector table an Adresse $0_0000 des 8086-Programms haben.

Teil 2: Erzeugung und Verwendung von Assemblermodulen

3 Der Stand-Alone-Assembler

Der Inline-Assembler ist sicherlich eine tolle Sache, vor allem, nachdem Delphi 6.0 und der zu erwartende CBuilder 6.0 eine gewaltige Erweiterung des unterstützten Befehlssatzes erfahren haben. Auch Visual C++ hat durch den Patch von MASM von Version 6.13 auf 6.15 eine Menge dazugelernt, wie man hört. Bis zu diesem Zeitpunkt konnten »inline« keine SIMD-Befehle genutzt werden und der Befehlssatz war auf die Möglichkeiten eines 80386 beschränkt.

Das hat sich geändert, und so erhebt sich die Frage: Welchen Sinn machen Stand-alone-Assembler wie MASM und TASM noch? Die Frage scheint berechtigt, und so haben sowohl Microsoft als auch Borland den Vertrieb der eigenständigen Assembler eingestellt: mangels Nachfrage.

Vermutlich haben viele Leute eine falsche Vorstellung darüber, wie »kompliziert« ein Assembler eigentlich ist. Sieht man einmal davon ab, dass auch heute noch die Assembler unter DOS arbeiten und mittels Kommandozeilen-Parameter gesteuert werden, stehen sie in Benutzerfreundlichkeit Hochsprachencompilern in nichts nach – vor allem wenn man diese in der Kommandozeilen-Version nutzt.

Daher akzeptieren viele Hochsprachen-Entwicklungsumgebungen Assembler nur in der Form des Inline-Assemblers in Hochsprachen, wenn überhaupt. Dennoch macht es Sinn, den eigenständigen Assembler nicht zu begraben. Denn es gibt noch eine Reihe von Gründen, weshalb er immer noch seinen Platz hat.

- Sie können assembleroptimierte Module entwickeln, die Sie über eine definierte Aufrufkonvention in unterschiedliche Hochsprachen einbinden können: C++, Delphi, Basic.
- Sie können vor allem mit Hilfe von Makros, die es in den Inline-Versionen nicht gibt, Dinge realisieren, die ohne Assembler nicht oder

nur sehr schwer realisierbar sind. Wir werden noch Beispiele kennen lernen.

- Sie haben weitgehende Kontrolle über das, was Sie programmieren wollen.

Und da der zusätzliche Aufwand sehr gering ist, einmal verstanden zu haben, wie man »stand-alone« assembelt und weiterhin die modernen Assembler von heute so komfortabel wie Hochsprachen sind, gibt es nicht wirklich einen Grund, nicht »hardcore« zu programmieren.

Bleibt also zu hoffen, dass Microsoft und Borland in irgendeiner Weise auch in Zukunft MASM und TASM zur Verfügung stellen und, so es sinnvoll ist, auch Patches veröffentlichen, die Erweiterungen der Möglichkeiten der CPU berücksichtigen.

3.1 Vorbemerkungen

Zur Nutzung des Assemblers finden Sie auf der beiliegenden CD-ROM einige Dateien, sodass an dieser Stelle auf entsprechende Beispiele verzichtet werden kann.

3.1.1 Datenbezeichnungen

In diesem Buch wurden bislang die Datenbezeichnungen verwendet, die auf Seite 30 definiert wurden. Dies wird auch so bleiben! Jedoch müssen Sie sich damit abfinden, dass der Assembler, genau wie die Hochsprachen auch, eigene Bezeichnungen für Datentypen hat. Von ihnen wird in diesem Kapitel auch zu reden sein! In Tabelle 5.13 auf Seite 816 finden Sie eine Gegenüberstellung der verschiedenen Datenbezeichnungen.

3.1.2 Symbole

Der Assembler arbeitet symbolorientiert. Unter einem Symbol versteht er eine beliebige Reihenfolge von Zeichen, die jedoch je nach Kontext gewissen Regeln unterworfen ist und bestimmte Funktionen hat.

So kann der Programmierer Adressen von Einsprungpunkten in Codesequenzen mit Symbolen, Namen, etikettieren, sprich »labeln«. Diese *Label* sind für den Assembler nichts anderes als Platzhalter für Adressen im Speicher. Dementsprechend »übersetzt« er auch jeweils ein Label im Quelltext in die dazugehörige Adresse im OBJ-File.

Auch Variable kann der Assembler, wie in Hochsprachen, benennen. Hier steht ebenfalls hinter dem Symbol, dem Namen der Variablen, die Adresse, unter der sie im Datensegment zu finden ist. Wie in Hochsprachen auch, muss sich der Programmierer keine Gedanken um diese Adressen machen! Der Assembler führt entsprechende Zuordnungslisten.

Eine dritte Art von Symbolen sind Ersetzungen. Hier ersetzt der Assembler das Symbol durch einen »Wert«. Solche Werte können Zahlen, aber auch Strings sein. Dies entspricht den »untypisierten Konstanten« in Hochsprachen.

Zu dieser dritten Art von Symbolen gehören auch die *vordefinierten Symbole*, die die Assembler bereitstellen. Sie werden durch den Assembler deklariert und können wie selbst deklarierte Symbole benutzt werden.

3.1.3 Expression

In den folgenden Kapiteln wird häufig der Begriff *expression* zu lesen sein. Er wird immer dann auftreten, wenn z.B. bei Deklarationen einem Datum ein Wert zugeordnet werden soll, wobei der Begriff »Datum« hier sehr weit gefasst sein soll. Daher hier die Definition:

Unter *expression* versteht man jede gültige Kombination von mathematischen oder logischen Variablen, Konstanten, Strings und Operatoren, die einen einzigen Wert ergeben. Expressions müssen zum Zeitpunkt der Assemblierung durch den Assembler eindeutig berechenbar sein. Daher können Variablen nur dann Teil einer Expression sein, wenn sie initialisiert und vor der Stelle deklariert wurden, an der die Expression steht.

Beispiele von gültigen und ungültigen Expressions:

```
4711                  gültig: ergibt numerischen Wert 4711
'A'                   gültig: repräsentiert den Wert 65
'String'              ungültig: ergibt keinen numerischen Wert
5 * 3                 gültig: kann evaluiert werden
4 * [EAX]             ungültig: Inhalt von EAX unbekannt
-2                    gültig: unärer Operator »-«
5 * (3 + 5) / -4      gültig: ergibt -10
4 * SIZE ByteVar      gültig, wenn ByteVar bereits initialisiert
ByteVar AND OFFh      gültig, wenn ByteVar bereits initialisiert
4711d OR 08000h       gültig: ergibt 09267h
```

3.1.4 Qualifizierte Typen

In den folgenden Abschnitten werden Sie häufig den Begriff »Qualifizierter Typ« lesen. Was ist das?

Es gibt zwei Arten der Definition des Begriffs *Qualifizierter Typ*

- Jeder Typ, der in Assembler deklariert ist, wie die einfachen Typen BYTE, WORD, DWORD, QWORD, OWORD und TBYTE, aber auch structures, records oder intrinsische Typen. Ferner ist jeder Typ qualifiziert, wenn er mittels TYPEDEF anstandslos deklariert werden konnte.
- Jeder Verweis auf einen Qualifizierten Typen der Form

  ```
  [distance] PTR [qualified type]
  ```

  wobei *distance* und *qualified type* optional sind und *distance* folgende »Werte« annehmen kann: FAR, FAR16, FAR32, NEAR, NEAR16, NEAR32, SHORT

Bei der Verwendung von *Qualifizierten Typen* können Komponenten nur dann in Form einer Vorwärts-Referenz verwendet werden, wenn es sich um structures oder records handelt. Andere Typen müssen bereits deklariert sein.

Wenn bei Verweisen auf qualifizierte Typen *distance* nicht angegeben wird, wird der Typ des Pointers durch den rechten Operanden und das aktuelle Speichermodell bestimmt. Wenn nicht über die Direktive .MODEL ein Speichermodell angegeben wird, wird NEAR als *distance* angenommen.

3.1.5 Beispiele

TASM verfügt über zwei Betriebsmodi, den MASM-Modus, der Kompatibilität zum MASM garantiert, und den IDEAL-Modus, der TASM-spezifisch ist. Da im Rahmen dieses Buches auf Kompatibilität geachtet wird, sind die Beispiele in den folgenden Kapiteln am MASM angelehnt und sollten im MASM-Modus von TASM anstandslos verstanden werden. Auf Beispiele in TASMs IDEAL-Modus wird verzichtet, wenn die Unterschiede rein syntaktischer Art sind und somit leicht von Ihnen selbst umgeschrieben werden können.

3.2 Direktiven

In den folgenden Abschnitten werden Assembler-Direktiven vorgestellt, die quasi »Standard« sind und von den meisten Assemblern, in jedem Fall aber von Microsofts MASM und Borlands TASM (im MASM-Modus) »verstanden« werden. In zwei weiteren Kapiteln werden dann Besonderheiten besprochen, die MASM und TASM über diesen Standardsatz hinaus haben. Diese spezifischen Ergänzungen sind nicht kompatibel und können nur im jeweiligen Assembler eingesetzt werden.

3.2.1 Direktiven zur Datendeklaration

Analog zu Hochsprachen muss der Assembler wissen, um was für ein Datum es sich handelt, das da unter einem Namen (oder im Fachjargon: Symbol) angesprochen wird. Hierzu dienen die Direktiven zur Datendeklaration.

Der Assembler erlaubt hierbei zwei Deklarationsarten:

- uninitialisierte Deklaration
- initialisierte Deklaration

Allerdings folgen beide Arten dem gleichen Formalismus, sie unterscheiden sich lediglich im Argument, das für die Initialisierung verwendet wird.



Bitte beachten Sie, dass Datendeklarationen beim Assembler etwas anders erfolgen als in Hochsprachen. In Hochsprachen nennt man uninitialisierte Daten Variablen und der Compiler steckt sie in ein bestimmtes Datensegment für uninitialisierte Daten. Initialisierte Daten dagegen sind häufig Konstanten, die im Programmablauf nicht verändert werden können (dürfen) und somit je nach Programmiersprache besonders behandelt und in ein anderes Datensegment gepackt werden (können). Dazwischen gibt es je nach Compiler auch initialisierte Variable, die wiederum in einem anderen Datensegment stecken (können). Das alles steuert der Compiler, der hier bestimmten Vereinbarungen der Kompilierung folgt, die die Compilerbauer vorgegeben haben.

Der Assembler tut dies nicht! *Sie* bestimmen, in welches Datensegment welches Datum kommt, indem Sie den Ort der Deklaration bestimmen. So werden Daten, die im Codesegment deklariert werden, durch den

Assembler auch dort angesiedelt, vollkommen egal, ob sie initialisiert sind oder nicht, ob sie als Konstanten dienen oder als Variable. Denn der Assembler kennt den Unterschied nicht!

Daten-Allozierung

Formal erfolgt eine Datenallozierung nach

[Name] Datentyp Initialisierer [, Initialisierer [, Initialisierer ...] ...]]
[Name] Datentyp Const DUP (Initialisierer [, Initialisierer [, Initialisierer ...] ...])

wobei *Datentyp* einer der im Folgenden genannten Typen sein kann und die eckigen Klammern bedeuten, dass die Angabe optional ist. Der Ausdruck `Const DUP ( )` bewirkt, dass die Initialisierungen, die in den runden Klammern stehen, *const*-mal wiederholt werden. *Name* ist ein Symbolname, den Sie frei wählen können.



Ganz so frei wählen können Sie ihn nicht! So darf er nicht identisch mit einem vordefinierten Symbol sein, also einem Namen, den der Assembler bereits kennt und für andere Zwecke verwendet, z.B. »DUP« (reserviert für den Operatoren DUP) oder »EAX« (reserviert für das Register EAX). Und er darf nicht mit einer Ziffer beginnen, da Zeichenfolgen, die mit einer Ziffer beginnen, für den Assembler Werte sind, die in einem Ausdruck verwendet werden.



Sie können den Namen jedoch auch weglassen. Sinn macht das aber nur, wenn Sie auf die so reservierten Speicherbereiche irgendwie anders zugreifen können. Im Stringbeispiel, das weiter unten folgt, werden Sie eine Anwendung davon sehen: So dürfen im Assembler Deklarationen nicht über mehrere Zeilen erfolgen, sie müssen innerhalb einer Zeile abgeschlossen sein. Einer der Strings im Beispiel weiter unten lässt sich aber nicht in einer Zeile deklarieren. Daher wird die erste Zeile mit einem Namen versehen, um auf den String durch ein Symbol (den Namen) zugreifen zu können; da sich die anderen, namenlosen, Bytes direkt an den namentragenden anschließen, gibt es keine Probleme. Eine weitere Anwendung liegt im Rahmen der Definition von Strukturen, wie wir noch sehen werden.

Initialisierer kann ein Ausdruck der folgenden Art sein:

? Das Fragezeichen dient zur Deklaration uninitialisierter Daten. In diesem Fall wird zwar Speicher für das Datum alloziert, nicht aber mit einem vorgegebenen Wert belegt. Man spricht hier von *unbestimmten Vorgaben*.

Beispiel:

```
ByteVar BYTE ?
```

Wert Unter *Wert* versteht der Assembler einen Ausdruck (*expression*, vgl. Seite 558), den er in einen einzigen numerischen Wert übersetzen kann. Dies kann entweder ein numerischer Wert selbst sein, z.B. »5«, oder ein Zeichen, z.B. 'C' (das er in den numerischen Wert $43 = 67, den ASCII-Wert für C, übersetzt), aber auch ein komplexer Ausdruck, der ggf. Operatoren verwendet, aber einen numerischen Wert liefert. Beispiele:

```
ByteVar  BYTE OFFh
CharVar  BYTE 'H'
ByteVar2 BYTE (Length CharVar) * 8
```

Wie gesagt: Es ist egal, was für Sinn oder Unsinn Sie hier angeben – der Assembler akzeptiert ihn, solange *Wert* den syntaktischen Regeln genügt und dessen Auswertung einen numerischen Wert ergibt, mit dem das Datum initialisiert werden kann.

Wertfeld Ein Wertfeld verwendet eine Deklaration, die den Operator DUP (»duplicate«) beinhaltet. DUP wird in der Form *count DUP expression* verwendet und wiederholt den in *expression* genannten Ausdruck *count*-mal, wie das Beispiel zeigt:

```
ArrayVar BYTE 20 DUP 0
```

deklariert einen Speicherbereich namens *ArrayVar* mit 20 Bytes Umfang und initialisiert ihn mit dem Wert »0«. Das kann beliebig kompliziert werden:

```
Array3D WORD 3 DUP (2 DUP (1,2,3), 3 DUP (4,5))
```

deklariert einen Speicherbereich namens *Array3D*, der aus 36 Words ($3 \cdot ((2 \cdot 3) + (3 \cdot 2))$) besteht und folgenden Inhalt hat:

1,2,3,1,2,3,4,5,4,5,4,5, 1,2,3,1,2,3,4,5,4,5,4,5, 1,2,3,1,2,3,4,5,4,5,4,5

String Auch ganze Strings können als Initialisierer einem Datum zugeordnet werden. Hierbei werden die einzelnen Zeichen in einfachen (') oder doppelten (") Anführungszeichen eingeschlossen:

```
StrVar BYTE 'Dies ist ein ASCII-String'
```

StrVar wird deklariert als Feld von Bytes und mit den Werten 68d, 105d, 101d, 115d, 32d, 105d, 115d, 116d, 32d, 101d, 105d, 110d, 32d, 65d, 83d, 67d, 73d, 73d, 173d, 83d, 116d, 114d, 105d, 110d, 103d vorbelegt.

Gemäß der Intel-Notation wird hierbei das am weitesten links stehende Byte (Zeichen) an der niedrigsten Speicherstelle abgelegt, sodass wie gewohnt das in Abbildung 3.1 gezeigte Speicherabbild entsteht, wobei links unten bei niedrigen Speicheradressen das most significant byte (MSB) ist, rechts oben bei höheren Speicheradressen das least significant byte (LSB).

Damit ist das Ergebnis das gleiche, als hätte man *StrVar* so deklariert:

```
StrVar BYTE 'D','i','e','s',' ','i','s','t',' '
       BYTE 'e','i','n',' ','A','S','C','I','I'
       BYTE '-','S','t','r','i','n','g'
```



Abbildung 3.1: Speicherabbild des Strings "Dies ist ein ASCII-String"



Bitte beachten Sie, dass Sie Strings nur dann wie oben deklarieren können, wenn der verwendete Datentyp BYTE ist. Daten vom Typ WORD, DWORD und QWORD dürfen zwar auch »Strings« zugeordnet werden, allerdings nur in ihrer Größe, also mit 2, 4 oder 8 Bytes Umfang:

```
WordStr  WORD  'Hi'
WordStr2 WORD  'Fan'     ; falsch, da drei Zeichen
DWordStr DWORD 'Fan!'
QWordStr QWORD 'Hi, Fan!'
```

Im Unterschied zur String-Deklaration oben handelt es sich hier um eine »Zeichendeklaration«. Und bei ihr wird der am weitesten links stehende Buchstabe an höchster Speicherstelle abgelegt, der am weitesten rechts stehende an niedrigster – so wie ja auch bei Integers im Word-, DoubleWord- oder QuadWord-Format das höchstwertige Bit links und das niedrigstwertige rechts steht. Das Speicherabbild von *QWordStr* ist in Abbildung 3.2 dargestellt. Der Inhalt der Speicherstelle ist somit gegenläufig dem nach BYTE-Deklaration!



Abbildung 3.2: Speicherabbild des QuadWord-Strings "Hi, Fan!"

Diese Darstellung betraf nur »Zeichen«. Was aber, wenn man ganze »Strings« auf diese Weise deklariert, also mehrere Daten eines Type hintereinander deklariert? Zur Beantwortung dieser Frage nochmals die Deklaration von einem String, einmal mit Bytes und einmal mit DWords und Words:

```
StrVar1 BYTE  'Das ist  ein  Test'
StrVar2 DWORD 'Das ','ist ','ein ','Test'
StrVar3 WORD  'Da','s ','is','t ','ei','n ','Te','st'
```

liefert die in Abbildung 3.3 dargestellten Speicherabbilder für *StrVar1*, *StrVar2* und *StrVar3*.



Abbildung 3.3: Speicherabbild des Strings "Das ist ein Test" mit verschiedenen Datentypen

Zunächst ist aus der Abbildung ersichtlich, dass die zuerst deklarierte Variable *StrVar1* an der niedrigsten Speicheradresse abgelegt wurde (untere Zeile). Sie hat, wie für Byte-deklarierte Strings üblich, die bereits in Abbildung 3.1 dargestellte Ausrichtung des Strings. Die darauf folgende Deklaration der Variablen *StrVar2* legt die deklarierten »Zeichen« des Strings auch nach der Reihe ihrer Deklaration von rechts nach links, d.h. von niedrigen zu hohen Adressen ab (mittlere Reihe), folgt also konsequent der Intel-Notation. Allerdings sind sie mit DoubleWords codiert, sodass sie analog Abbildung 3.2 innerhalb der DoubleWords von links nach rechts ausgerichtet sind! Die zum Schluss deklarierte Variable *StrVar3* setzt dem Ganzen noch die Krone auf, da hier die Zeichen in Words codiert sind und somit ein scheinbar heilloses Durcheinander in den Laufrichtungen herrscht. Erkennen Sie in *StrVar3* noch den ursprünglichen Text?



Was lernen wir daraus? Lesen Sie Daten immer in dem Format aus dem Speicher aus, mit dem Sie sie hineingeschrieben haben. Das Byte-weise Auslesen eines Word-weise im Speicher abgelegten Strings und umgekehrt wird Ihnen keine große Freude bereiten! Und, wenn nicht absolut gewollt und bewusst diese Feinheiten der Assembler-Programmierung benutzt werden sollen, deklarieren Sie Strings immer mit Byte-Variablen! Eine bewusste und gewollte Anwendung könnte jedoch in einer einfachen und leicht durchschaubaren Art bestehen, Strings im Compilat »unsichtbar« zu machen. So sticht zumindest bei »`iDsei tsm ie naPssowdr`« nicht sofort ins Auge, welche Stringkonstante an Adresse $0496_58A3 steht: »Dies ist mein Password«.



Noch ein Wort zu numerischen Werten! Die Art, wie numerische Eingaben interpretiert werden, hängt von der aktuellen Zahlenbasis ab, die mit der Direktive .RADIX (vgl. Seite 686) eingestellt werden kann. Alternativ kann jedoch die gewünschte Zahlenbasis durch ein entsprechendes Suffix vorgegeben werden: 11b (binär) , 17o (oktal), 12d (dezimal), 0FFh (hexadezimal). Bitte beachten Sie, dass beim Assembler Zahlen immer mit einer Ziffer beginnen, weshalb z.B. $FF als 0FFh dargestellt werden muss.

Zugriff auf Daten

Der Variablenname ist ein Symbol, das der Assembler mit der Adresse des Datums im Datensegment gleichsetzt, die im Rahmen der Allozierung von Daten generiert werden. Auf diese Weise ist die Verbindung geschaffen zwischen der Arbeitsweise des Programmierers, der mit Symbolen (Variablennamen) besser zurechtkommt und arbeitet, und dem Prozessor, der nur Adressen in einem Adressraum kennt. Wenn man also im Rahmen von Instruktionen ein Datum über seinen symbolischen Namen benutzt und schreibt

```
MOV     EAX, DWordVar
```

so heißt das für den Prozessor:

```
MOV     EAX, DWORD PTR 000001234h
```

und der Assembler stellt diese Beziehung her, da er weiß, dass er durch die Deklaration der Variablen *DWordVar* irgendwo im Quelltext gemäß

```
DWordVar DWORD ?
```

vier Bytes an Adresse $0000_1234, der nächsten freien Adresse im Datensegment, für ein Datum reserviert hatte, das der Programmierer als DWORD interpretiert und *DWordVar* nennt.



Dies ist auch der Grund dafür, warum ein Debugger gerne den Quelltext des Assemblermoduls hätte und ggf. anmahnt oder zumindest sein Fehlen meldet (»program has no symbol table«). Falls er nämlich die Symbole nicht kennt, die mit einzelnen vom Assembler in die Befehlssequenzen eingebauten Adressen verbunden sind – was in der Regel der Fall sein dürfte –, so muss er zur Darstellung die Adresse selbst heranziehen. Dies sind dann die Momente, die ich so liebe und an denen so nette Konstrukte wie

```
MOV     EAX, DWORD PTR 00000786h
```

im Disassemblat erscheinen, die bärig aufschlussreich sind und einem das Lesen und Verstehen so leicht machen!

BYTE
WORD
DWORD
QWORD

Diese Direktiven sind dafür zuständig, Speicherbereiche in Byte-Größe (BYTE), Word-Größe (WORD), DoubleWord-Größe (DWORD) oder QuadWord-Größe (QWORD) zu reservieren und dem Assembler unter dem gewählten symbolischen Namen bekannt zu machen. Bei diesen Direktiven handelt es sich um die grundlegenden Daten, die der Assembler »versteht«. Sie können immer dann verwendet werden, wenn ein Befehl einen Operanden der entsprechenden Größe benötigt. So kann ein DWORD unabhängig vom Vorzeichen (das Assembler und CPU überhaupt nicht interessiert!) eine SmallInt oder ein Word aufnehmen, DWORDS können Integers, aber auch Pointer oder SingleReals beinhalten und QuadWords ebenfalls Integers oder DoubleReals. Grund: Alle diese Daten haben jeweils die gleiche Größe. Und den Assembler interessiert nur die Größe, aber nicht die Art des betreffenden Datums.

OWORD



Die SIMD-Erweiterungen unter SSE haben Register eingeführt, die 128 Bit groß sind und mit entsprechenden Daten umgehen können. Mit den QWORDS standen bis zu dieser Erweiterung jedoch »nur« 64-Bit-Daten zur Verfügung. Folglich wurde die Einführung eines neuen Datentyps erforderlich, der ein 128-Bit- oder 16-Byte-Datum definieren und handeln kann: das OctelWord OWORD, das auch als DoubleQuadWord bekannt ist.



Diese Syntax-Erweiterung des MASM-Sprachschatzes erfolgte mit MASM 6.14 im Rahmen der Erweiterung der Syntax um die SSE-Befehle!



Es ist zu erwarten, dass der mit C++-Builder 6.0 von Borland ausgelieferte TASM diese Syntax-Erweiterung ebenfalls erfährt, da C++ sicherlich nicht hinter Delphi zurückbleiben soll, Delphi in Version 6.0 jedoch bereits die SSE-Befehle und damit OWORDs kennt. Leider kann ich das zum Zeitpunkt der Manuskripterstellung nicht verifizieren, da Borland noch keine Beta-Version des C++-Builders 6.0 zur Verfügung stellt.

SBYTE
SWORD
SDWORD

Wenn nun mit den »Signed«-Varianten von BYTE, WORD und DWORD dennoch Direktiven existieren, die ein Vorzeichen kennen und somit den in diesem Buch verwendeten ShortInts, SmallInts und LongInts entsprechen, so liegt das ausschließlich an einem bisschen Komfort, den man dem archaischen Assembler im Laufe seiner Evolution verpassen wollte. Der Assembler verwendet diese Deklarationen intern bei Vergleichen und bedingten Assemblierungen sowie bei der Realisierung von Hochsprachen-Interfaces (INVOKE).

REAL4
REAL8
REAL10

Was BYTE, WORD, DWORD und QWORD für die CPU, sind REAL4, REAL8 und REAL10 für die FPU. Sie entsprechen somit den in diesem Buch verwendeten SingleReals, DoubleReals und ExtendedReals.

TBYTE

Auch wenn sie nicht sehr geliebt und (in meinen Augen!) überflüssig sind: Auch BCDs kennt die FPU. Daher gibt es extra für sie einen Datentyp: TBYTE. Wie der Name schon suggeriert, umfasst er zehn Bytes und ist somit formal identisch zum Typ REAL10.

DB
DW
DD
DQ
DT

Für die eben besprochenen Datentypen gibt es noch »eingeschränkt funktionsfähige« Abkürzungen. So steht DB für »define byte«, DW für »define word«, DD für »define DWord«, DQ für »define QWord« und DT für »define ten bytes«. Wie die Bezeichnungen schon ausdrücken, werden sie *nur* zur »Definition«, also zur Datendeklaration verwendet. Dies ist auch die Einschränkung.

Das bedeutet: DB, DW, DD und DQ können absolut gleichwertig eingesetzt werden wie BYTE, WORD, DWORD, QWORD, REAL4 (= DW), REAL8 (= DD) oder REAL10 (=DT), um Daten zu deklarieren (»define«):

```
ByteVar   DB ?
StrVar    DB "Dies ist ein anderer String"
WordVar   DW OFFFFh
SignedVar DW -30000
SingleVar DD 1.1234E-56
```

```
DWordVar  DD 123456789
DoubleVar DQ –1.234E56
BCDVar    DT 4711
ExtendVar DT 9.87654321E123
```

Sie können jedoch nicht als Direktive z.B. im Rahmen des »type casting« eingesetzt werden:

```
MOV   EAX, DWORD PTR Variable  ist korrekt,
MOV   EAX, DD PTR Variable     ist verboten!
```



Ich persönlich verwende gerne die Kurzformen zur Datendeklaration. Erstens sind sie so schön kompakt, zweitens signalisieren sie durch ihre pure Anwesenheit, was erfolgt: Datendeklaration, und drittens lassen sich so gut lesbare Listings erstellen. Ich erkaufe mir das, indem ich mit z.B. DQ sowohl CPU-Integers als auch FPU-DoubleReals definiere. So what!

FWORD
DF
PWORD
DP

FWORD ist ein Datentyp, der früher unter 16-Bit-Betriebssystemen nicht eingesetzt werden musste (konnte, brauchte!) und heute in den modernen 32-Bit-Betriebssystemen wieder nicht eingesetzt werden muss (kann, braucht) – oder zumindest fast! Denn FWORD bzw. DF steht für FarWord bzw. »define FWORD«. Und unter FarWord versteht man eine 48-Bit-Struktur, die sich aus einem 16- und einem 32-Bit-Zeiger zusammensetzt, insgesamt also 6 Bytes umfasst.

FWORD beherbergt also Zeiger. In 16-Bit-Umgebungen ist dieser Datentyp überflüssig, da dort die Adressierung über einen 16-Bit-Selektor und einen 16-Bit-Offset erfolgt, insgesamt Zeiger also maximal 32-Bit umfassen und somit in DWORDS gehalten werden können. In modernen 32-Bit-Umgebungen haben wir das flat model vorliegen, das die gesamten 4 GByte des maximal ansprechbaren Adressraums linear über einen 32-Bit-Offset verfügbar macht. Somit benötigt man für die Zeiger wiederum nur 32-Bit und damit DWORDS. FWORDS machen also nur Sinn, wenn neben dem 32-Bit-Offset auch ein 16-Bit-Selektor abgespeichert werden soll. Dann – und nur dann – schlägt die Stunde der FWORDS.

Und die Stunde schlägt nur dann, wenn z.B. mit Hilfe von Intersegment- oder Interprivileg-CALLs oder JMPs ein gleichzeitiges Verändern des CS- und EIP-Registers erforderlich und den Befehlen somit eine Adresse übergeben werden muss, die auf eine FWORD-Struktur

zeigt. Aber das kommt ja für uns Nicht-Betriebssystem-Programmierer nicht in Frage ...



PWORD, pointer word, und DP sind Synonyme für FWORD und DF.

3.2.2 Direktiven zur Typ-Deklaration

Wie in Hochsprachen auch, können in Assembler eigene Typen deklariert werden. Dies betrifft sowohl »einfache« Datentypen wie auch kompliziertere Datenstrukturen.



An dieser Stelle stellt sich ein kleines sprachliches Problem. Zum einen gibt es die Struktur im engeren Sinne, die durch eine Typ-Deklaration mit der Direktive STRUC(T) erzeugt wird. Auf der anderen Seite können solche Strukturen auch im Rahmen komplexer »Daten-Strukturen« eingesetzt werden. Somit gibt es ein Wort für zwei unterschiedliche Sachverhalte.

Ich werde in den folgenden Kapiteln das Problem dadurch lösen, dass ich den englischen *terminus technicus* »structure« immer dann verwende, wenn ich eine Struktur meine, die mittels STRUC(T) generiert wird. Der deutsche Begriff »Struktur« bleibt entweder allgemeinen Datenstrukturen oder den komplexen, zusammengesetzten Strukturen vorbehalten.

STRUC(T) ENDS

Einer dieser über die »einfachen« Datentypen hinausgehenden und somit den komplizierteren Datenstrukturen zuzuordnenden Typen ist die »*structure*«. Ihre Deklaration wird eingeleitet durch die Direktive STRUCT und beendet durch ENDS, »end of segment«.



STRUCT in Assembler ist das, was es in C++ auch ist: Eine Zusammenfassung von Daten zu einer Datenstruktur, die über ein gemeinsames Symbol angesprochen werden kann. Eine *structure* ist somit so groß wie die Summe aller ihrer Mitglieder (*members*). In Delphi und Pascal wird die STRUCT als RECORD bezeichnet, was etwas unglücklich ist, da es unter Assembler auch den Datentyp RECORD gibt, der nichts mit Delphi-Records zu tun hat!



Die formale Deklaration von structures erfolgt in MASM und TASM etwas unterschiedlich. So besitzt TASM den IDEAL-Modus und hat Erweiterungen vorgenommen, die einen Einsatz von structures im Rah-

men von Objektorientierter Programmierung (OOP) ermöglichen. Daher wird im Folgenden die allgemeine Deklaration dargestellt, die so von MASM und TASM (im MASM-Modus) unterstützt wird. Die MASM- oder TASM-spezifischen Abweichungen und Unterschiede schließen sich daran an.

Deklaration

Die allgemeine formale Deklaration von structures lautet:

```
[Name] STRUCT
    StructMembers
[Name] ENDS
```

Name ist der Symbolname, der der structure gegeben wird. Er kann bei der Einleitung der Deklaration vor dem Schlüsselwort STRUCT sowie beim Abschluss der Deklaration vor dem Schlüsselwort ENDS stehen!



Name muss angegeben werden, wenn eine Typ-Deklaration erfolgt, mit Hilfe derer Variablen alloziert werden sollen. Weggelassen werden darf *Name* nur dann, wenn die Direktive STRUCT im Rahmen verschachtelter Strukturen eingesetzt wird. Wird *Name* vor der Direktive ENDS weggelassen, wird das aktuell geöffnete »Segment« geschlossen. Dies ist die structure, falls nicht innerhalb von ihr Deklarationen von anderen »Segmenten« erfolgten, die noch offen sind! In diesem Falle würde mit dem vermeintlich STRUCT abschließenden ENDS das noch geöffnete Segment geschlossen und vom Assembler eine noch geöffnete structure moniert. Um dies auszuschließen, kann durch *Name* das von ENDS zu schließende »Segment« genannt werden.

StructMembers nun ist eine Liste der einzelnen Komponenten, die die structure ausmachen. An dieser Stelle können die Deklarationen einfacher Daten wie BYTEs, WORDs & Co. stehen, aber auch komplexe Deklarationen wie structures, unions oder records. Und selbst eigene deklarierte Datentypen können Sie verwenden. Ein Beispiel finden Sie weiter unten.



Gegenüber der allgemeinen Form der Deklaration besitzt MASM noch zwei MASM-spezifische Argumente, die nach den Befehl STRUCT angegeben werden können. Unter MASM sieht somit die formale Deklaration von structures wie folgt aus:

```
[Name] STRUCT  [align] [, NONUNIQUE]
    StructMembers
[Name] ENDS
```



So ist es manchmal wichtig, Daten auszurichten. Hierunter versteht man, dass Daten an Adressen beginnen sollen, die durch einen bestimmten Divisor restlos teilbar sind. Man spricht von WORD-Ausrichtung, wenn die Adresse ohne Restbildung durch 2 teilbar ist, bei DWORD-Ausrichtung muss sie durch 4 teilbar sein. Alle CPUs ab dem 80386 (mit 32-Bit-Registern) arbeiten in 32-Bit-Umgebungen besonders effizient bei Speicherzugriffen, wenn die Daten DWORD-ausgerichtet sind.

Je nach Deklaration der structure und des Typs der Komponenten kann es dazu kommen, dass Daten nicht optimal ausgerichtet sind. So liegt z.B. in

```
MisAligned STRUCT
    FirstByte   DB ?
    FirstDWord  DD ?
    SecondByte  DB ?
    SecondDWord DD ?
ENDS
```

FirstByte an einer »geraden« Adresse (soweit das mittels dieser structure definierte Datum an einer geraden Adresse beginnt!), *FirstDWord* aber beginnt an der nächsten, ungeraden Adresse (+1) und ist damit unausgerichtet. Auch *SecondByte* liegt nicht optimal (+5). Somit liegt *SecondDWord* exakt in der Mitte zwischen zwei ausgerichteten Adressen (+6)!

MASM gestattet daher, innerhalb der STRUCT-Deklaration einen Wert für das alignment anzugeben:

```
Aligned STRUCT 4
    FirstByte   DB ?
    FirstDWord  DD ?
    SecondByte  DB ?
    SecondDWord DD ?
ENDS
```

Dies bewirkt, dass alle Komponenten an DWORD-Grenzen ausgerichtet werden (wie man Adressen, die durch 4 restlos teilbar sind, nennt) und *FirstDWord* nun an die nächste DWORD-Grenze gesetzt wird, in unserem Falle bei +4. Damit hat sich auch der Beginn von `SecondByte` verschoben (+8), weshalb auch dieses Byte und *SecondDWord* (+12) nun ausgerichtet sind.



Es ist wie so vieles im Leben eine Güterabwägung, ob man *align* benutzt oder nicht! Den Zuwachs an Performance durch effizienteren Speicherzugriff mit ausgerichteten Daten erkauft man sich durch Speicherverschwendung! Da die »Füllbytes«, die zwischen einem Datum und dem nächsten ausgerichteten Datum durch *align* eingeführt werden, nicht adressiert werden sollen (sonst brauchte man die Daten ja nicht auszurichten!), sind sie nutzlos. Je mehr alignment erforderlich ist, desto mehr Speicher wird verschwendet: Performance auf Kosten von Datengröße.



Anders als z.B. in RECORDs, können die Namen der Komponenten in structures mehrfach im Quellcode verwendet werden. Ihre Gültigkeit ist somit nicht global, sondern lokal auf die Struktur begrenzt. Gibt es dagegen im gesamten Quellcode kein Symbol mit gleichem Namen, so sind solche Komponentennamen »unique« und quasi global sichtbar. Das bedeutet, man kann sie z.B. zur Adressierung benutzen, ohne eine »qualifizierte« Angabe machen zu müssen (was das ist, klären wir weiter unten!). Dies kann manchmal unerwünscht sein, und so erlaubt MASM bei einer STRUCT-Dekaration das Symbol NONUNIQUE als Argument anzugeben. Es definiert alle Komponentennamen als nicht einzigartig (»non-unique«), auch wenn sie es sind, und erzwingt somit die Verwendung qualifizierter Zugriffe auf die Komponenten.



Auch TASM hat zwei Besonderheiten: eine kleine, aber wichtige syntaktische Abwandlung im MASM-Modus und die Deklaration von structures im TASM-Modus:

MASM-Modus

```
[StructName] STRUC
    StructMembers
[StructName] ENDS
```

TASM gestattet hier neben dem MASM-STRUCT auch das TASM-eigene STRUC ohne »T« im Schlüsselwort. Verwenden Sie jedoch STRUCT, wenn Sie Quellcode schreiben, der von MASM und TASM akzeptiert werden soll. STRUC dient zur Herstellung gewollter und bewusster Inkompatibilitäten, wenn die Ergänzungen für Objektorientierte Programmierung (OOP) benutzt werden sollen.

IDEAL-Modus

```
STRUC(T)[StructName]
    StructMembers
ENDS    [StructName]
```

Wie in IDEAL-Modus üblich, erfolgt die Deklaration eines Typs mit der Nennung des Schlüsselwortes, dem dann der Name folgt. So auch hier: Die Deklaration beginnt mit dem Schlüsselwort STRUC(T), gefolgt durch den Namen, den Sie der structure geben wollen. Auch im IDEAL-Modus folgt dann die Liste der Komponenten-Deklarationen, die durch ENDS abgeschlossen wird. Im IDEAL-Modus ist die Angabe des Namen der structure mit der ENDS-Direktive ebenfalls optional.



Auch TASM erlaubt, die Komponenten der structure an Grenzen auszurichten. Allerdings erfolgt dies hier nicht über das zusätzliche, optionale Argument *align*, sondern über die Direktive ALIGN, die für eine Datenausrichtung sorgt und in der Deklaration der structure angegeben wird:

```
Aligned STRUCT
    ALIGN       4
    FirstByte   DB ?
    FirstDWord  DD ?
    SecondByte  DB ?
    SecondDWord DD ?
ENDS
```

Dies bewirkt, dass *FirstDWord* an die nächste Adresse gesetzt wird, die durch vier restlos teilbar ist, in unserem Falle bei +4. Damit hat sich auch der Beginn von `SecondByte` verschoben (+8), weshalb auch dieses Byte und *SecondDWord* (+12) nun an DWORD-Grenzen ausgerichtet sind.



TASM unterstützt auch OOP. Daher wurde die Syntax erweitert, um auch Objekte definieren zu können. ACHTUNG: Hier *müssen* Sie das Schlüsselwort STRUC verwenden, da diese Erweiterung in MASM nicht implementiert ist und daher absichtlich Inkompatibilitäten erzeugt werden sollen! Die Deklaration von Objekten ist sowohl im MASM- als auch im IDEAL-Modus möglich:

MASM-Modus
```
[ObjName] STRUC [Modifiers] [Parent] [METHOD MethodList]
    StructMembers
ENDS
```

IDEAL-Modus
```
STRUC [ObjName] [Modifiers] [Parent] [METHOD MethodList]
    StructMembers
ENDS
```

Zu den *StructMembers* brauche ich nichts weiter zu sagen, die Deklaration erfolgt wie bei »einfachen« Strukturen.

Als *Modifier* kommen folgende Schlüsselworte in Frage: GLOBAL, was bewirkt, dass die Adresse der *virtual method table* (VMT) global veröffentlicht wird, NEAR, was bedeutet, dass die VMT durch einen NEAR-Pointer (16-Bit, wenn das aktuelle Programmiermodell USE16 ist, und 32-Bit, wenn es USE32 ist!) angesprochen wird und FAR, wenn sie durch einen FAR-Pointer (16-Bit-Selektor + 16-Bit-Offset bzw. 16-Bit-Selektor und 32-Bit-Offset) angesprochen werden soll.

Parent ist der Name des zugrunde liegenden Objektes. Das neu deklarierte Objekt erbt alle *StructMembers* und Methods von ihm. TASM unterstützt nur die »einfache Erbfolge«, soll heißen, es kann immer nur von einem Basisobjekt erben, weshalb als *Parent* auch nur ein Objekt angegeben werden kann.

MethodList hat die Struktur

```
TableMember [, TableMember [, TableMember ...]]]
```

Die einzelnen *TableMembers* folgen der Syntax, die bei der Deklaration von Tables (vgl. Seite 597) eingesetzt wird.

Info: Bitte haben Sie Verständnis dafür, dass ich im Rahmen dieses Buches nicht genauer auf OOP und die OOP-Fähigkeiten von TASM eingehen kann! Falls Sie hierzu Informationen benötigen, verweise ich zum einen auf die Dokumentation von TASM 5.x und zum anderen auf weiterführende Literatur.



Daten-Allozierung

Formal erfolgt die Allozierung eines Datums vom Typ STRUCT (oder STRUC) nach:

[VarName] StructName < [Expression [, Expression [, Expression ...]]] >
[VarName] StructName { [Initialisierer [, Initialisierer [, Initialisierer ...]]] }
[VarName] StructName Const DUP ({ [[Initialisierer [, Initialisierer ...]]] })
[VarName] StructName ?

und damit wie die Allozierung von Daten einfacher Typen auch. Auf die Darstellung der Allozierung von Daten, die im Rahmen von OOP bei TASM eingesetzt werden sollen, wird an dieser Stelle verzichtet!

Const DUP ( ) führt auch in diesem Fall wieder dazu, dass *const*-mal Speicher für die in der structure deklarierten Komponenten alloziert und die in den runden Klammern stehenden Initialisierungen wiederholt werden. *Expression* ist ein Ausdruck, wie er auch bei der Deklara-

tion von »einfachen« Daten verwendet wird (vgl. Seite 561). *Initialisierer* ist wie bei den einfachen Typen auch ein Ausdruck der Form

```
MemberName = expression,
```

der einen für die entsprechende Komponente gültigen Wert ergeben muss (vgl. Seite 558).



Bei der Allozierung von Variablen (Instanzen von Typen) generell dürfen Namen, mit denen sie benannt werden, im Quellcode nur einmal zur Deklaration verwendet werden (Symbole müssen einzigartig oder »unique« sein), da Variable grundsätzlich global im Assembler-Modul sichtbar sind. Innerhalb von structures ist dies anders! Da eine Instanz einer structure selbst einen einzigartigen Namen hat und ihre Komponenten nur über diese structure angesprochen werden können, können die Komponenten Namen aufweisen, die beliebig häufig bereits vergeben worden sein können. Denn über ihren »qualifizierten« Namen, bei dem der Variablenname ein Teil ist (s.u.), ist jede Komponente eindeutig identifizierbar, auch wenn *MemberName* mehrfach auftreten sollte.

Fühlen Sie sich daher frei, für die *MemberName*s jeden beliebigen Namen zu verwenden, selbst wenn es sich um reservierte Symbole des Assemblers handelt. Einzige Einschränkung: *Innerhalb* einer structure muss der entsprechende Name *unique* sein! Aber das dürfte klar sein.

Soll die structure uninitialisiert bleiben, so geben Sie wie bei den einfachen Typen das Symbol »?« an:

```
VarName StructName ?
```



Beachten Sie bitte, dass »`VarName StructName ?`« nicht das Gleiche ist wie »`VarName StructName <?>`« oder »`VarName StructName {?}`«! Im ersten Fall wird das gesamte Datum *VarName* uninitialisiert gelassen, im zweiten Fall nur die erste Komponente auf »uninitialisiert« gesetzt und im dritten Fall eine Fehlermeldung ausgelöst. Warum, werden Sie weiter unten sehen!

Soll jedoch die Komponente mit den Initialwerten initialisiert werden, die während der Deklaration der structure angegeben wurden, so erfolgt dies in folgender Form:

```
[VarName] StructName {}
[VarName] StructName <>
```

Die leeren, mit geschweiften »{ }« oder spitzen »< >« Klammern umfassten Bereiche veranlassen den Assembler, nur die Komponenten zu initialisieren, die bei der Deklaration der structure einen Initialwert mitbekommen haben. Alle anderen Komponenten bleiben uninitialisiert. Hierbei ist es unerheblich, ob leere geschweifte oder spitze Klammern verwendet werden!

Sollen schließlich einzelne oder alle Komponenten der structure initialisiert werden, kann man die geschweiften oder spitzen Klammern verwenden, um die zu benutzenden Initialwerte zu umklammern. Der Unterschied besteht darin, dass bei geschweiften Klammern jeweils Komponente und Initialwert in der Form

```
{Member1=Expr1 [, Member2=Expr2 [, Member3=Expr3 ...]]]}
```

angegeben werden müssen, während bei den spitzen Klammern die Initialwerte ohne Nennung der Komponentennamen in der Reihe der Deklaration der zu ihnen gehörenden Komponenten angegeben werden können, was sicherlich kompakter ist, die Lesbarkeit aber nicht positiv beeinflusst:

```
<Expr1 [, Expr2 [, Expr3 ...]]]>
```



Die Klammern bei der Deklaration eines Datums vom Typ STRUCT sind obligatorisch! Sie müssen immer angegeben werden, selbst wenn keine Initialisierungen erfolgen sollen und somit der umklammerte Bereich leer bleibt. Dabei können Sie wählen, ob Sie die geschweiften Klammern »{ }« oder die spitzen »<>« verwenden.



Es ist sehr wahrscheinlich, dass vor allem bei größeren structures die Initialisierer nicht alle in eine Zeile passen. Sie müssen somit auf mehrere Zeilen verteilt werden. Dies ist möglich nach jedem die einzelnen Komponenten-Initialisierungen trennenden Komma, also beispielsweise

```
VarName StructName < expression1, expression2,
                     expression3, expression4 >
```

Hierbei muss die öffnende Klammer in der Zeile der Deklaration der structure stehen und die schließende in der Zeile der letzten Initialisierung, wie im Beispiel gezeigt. Doch manchmal ist das nicht möglich (oder nicht erwünscht). Hier hilft das »Fortführungszeichen« (*continuation character*) »\«, das dann jeweils letztes Zeichen der Zeile sein muss

(außer Kommentaren!) und auch innerhalb komplexerer Expressions eingesetzt werden kann:

```
VarName StructName < \
                     expression1, expression2,
                     expres \
                     sion3,
                     expression4 \
                   >
```

Verschachtelte Strukturen

Structures können auch ineinander verschachtelt werden. Das bedeutet, dass eine structure in eine andere eingebettet (»nested«) ist und somit komplexere Strukturen ausbilden kann:

```
NestedStruct STRUCT
   AByte      DB ?
   STRUCT
      AWord   DW ?
      ADWord  DD ?
      ENDS
   BByte      DB ?
NestedStruct ENDS
```

In diesem Beispiel ist eine unbenannte structure bestehend aus einem WORD und einem DWORD zwischen zwei Bytes der »übergeordneten« structure eingebettet. Zugegeben: Hier ist das Quatsch, denn im Datensegment ist die Reihenfolge der Daten *AByte – AWord – ADWord – BByte*, also ganz so, als hätte man die structure wie folgt definiert:

```
SimpleStruct STRUCT
   AByte   DB ?
   AWord   DW ?
   ADWord  DD ?
   BByte   DB ?
SimpleStruct ENDS
```

Sinn aber machen verschachtelte structures, wenn andere Elemente einbezogen werden, wie z.B. Variante, und somit komplexe Strukturen bilden:

```
UnionStruct STRUCT
   AByte         DB ?
   UNION
      STRUCT
         AWord   DW ?
         ADWord  DD ?
      ENDS
```

```
      STRUCT
         BDWord  DD ?
         BWord   DW ?
      ENDS
   ENS
   BByte         DB ?
UnionStruct ENDS
```

Hier ist die Reihenfolge entweder *AByte – AWord – ADWord – BByte* oder *AByte – BDWord – BWord – BByte*.

Sie haben mehrere Möglichkeiten, structures und/oder unions zu verschachteln:




```
DemoStruct STRUCT
            SimpleStruct ?     ; Methode 1, namenlos

   Simple   SimpleStruct ?     ; Methode 2, benamt

   STRUCT                      ; Methode 3, namenlos
      UByte  DB           ?
      UWord  DB           ?
   ENDS

   Named STRUCT                ; Methode 4, benamt
      NByte  DB           ?
      NWord  DW           ?
   ENDS
DemoStruct ENDS
```

In Methode 1 wird einfach eine »außerhalb« der aktuellen deklarierte structure (hier: *SimpleStruct* von weiter oben!) eingebunden, aber keine Variable dieses Typs benannt. In diesem Fall gehen die Komponenten der »außen stehenden« structure in die neu deklarierte ein, und zwar genauso, als würden sie hier deklariert. Das bedeutet, *DemoStruct* verfügt durch die Einbindung von *SimpleStruct* nach Methode 1 über die Komponenten *AByte*, *AWord*, *ADWord* und *BByte*.

Bei Methode 2 erfolgt auch die Einbindung von *DemoStruct*, jedoch wird hier eine Variable dieser structure als Komponente alloziert.

In Methode 3 wird eine unbenannte structure innerhalb der aktuellen structure deklariert (»verschachtelt«). Der einzige Unterschied zu einer »normalen« Deklaration ist, dass kein Name für die structure vergeben wird. Dies ist auch nicht erforderlich, da ja alle Komponenten der verschachtelten structure wie Komponenten der übergeordneten structure aufgefasst werden können (vgl. *NestedStruct/SimpleStruct*)!

Soll jedoch auch die verschachtelte structure einen Namen bekommen, so kommt Methode 4 zum Einsatz.

Zugriff auf die Komponenten einer structure erhält man über ihren qualifizierten Namen, der sich aus dem Namen der betreffenden Variablen, dem DOT-Operator ».« und dem Komponentennamen ergibt:

```
VarName.MemberName
```

Dies nennt man »qualifizierten Zugriff« auf eine Komponente einer structure.

Was verbirgt sich dahinter? Analog den »einfachen« Daten wird mit *VarName* eine Adresse im Speicher gleichgesetzt, die die gesamte structure aufnimmt, also so groß wie die Gesamtheit der in ihr allozierten Daten ist. Die Komponentennamen sind nun nicht, wie die Variablennamen, Symbole für eine Adresse im Datensegment. Vielmehr sind sie nichts anderes als Offsets zu der Adresse der Datenstruktur. So hat das erste allozierte Datum der structure immer den Offset 0, das zweite den Offset (0 + Größe des ersten Datums), das dritte den Offset (0 + Größe des ersten Datums + Größe des zweiten Datums) usw.

Das bedeutet, dass für den Assembler der Ausdruck *Varname.MemberName* gleichbedeutend ist mit Adresse(*VarName*) + Offset(*MemberName*). Somit übersetzt er bei der Assemblierung dieses Konstrukt, dessen Werte ja bereits zu Zeiten der Deklaration und Allozierung bekannt sind, in eine stinknormale Adresse, die den Offset des ersten Bytes der Komponente der structure im Adressraum des Datensegments angibt.



Im Falle verschachtelter structures kommt es darauf an, wie sie deklariert wurden. Gesetzt den Fall, es gäbe eine Variable *Demo* vom Typ *DemoStruct*, so kann auf die Komponenten der namenlos eingebundenen structure direkt zugegriffen werden:

```
OR      Demo.BByte, 080h
```

Die Komponenten der Komponente *Simple* dagegen müssen »qualifiziert« angesprochen werden, was dazu führt, dass zwei DOT-Operatoren erforderlich sind:

```
MOV     AX, Demo.Simple.AWord
```

In diesem Fall liefert Demo die Basisadresse der »übergeordneten« structure und Simple den Offset des ersten Bytes der »verschachtelten« structure namens Simple. AWord wiederum liefert den Offset des ers-

ten Bytes des mittels AWord allozierten Words. Die Adresse dieses Bytes im Datensegment wird somit gebildet aus Adresse(*Demo*) + Offset(*Simple*) + Offset(*AWord*).

Bei unbenannt deklarierten verschachtelten structures erfolgt der Zugriff »normal«, also so, als handelte es sich um nicht verschachtelte Komponenten:

```
MOV     Demo.UWord, BX
```

Komponenten deklarierter, verschachtelter structures können ebenfalls nur »qualifiziert« angesprochen werden:

```
AND     Demo.Named.NWord, 08000h
```

Es folgt ein komplexes Beispiel, das die Deklaration von structures, unions und die Allozierung von davon abgeleiteten Daten zeigt: *Beispiel*

```
NameRec  STRUCT
   Last  DB 20 DUP (0)          ; 20 Null-Zeichen
   First DB 10 DUP (0)          ; 10 Null-Zeichen
ENDS


BirthRec RECORD Year:7 {        ; siehe Seite 588
   Month:4, Day:5 }


DateList TYPEDEF DWORD PTR      ; siehe Seite 594


Company  UNION                  ; siehe Seite 584
   Unit  DB 5  DUP (0)
   Name  DB 20 DUP (0)
ENDS

Person   STRUCT
   ID       DW       ?         ; »einfacher« Typ
   Name     NameRec  {}        ; STRUC
   Birth    BirthRec {}        ; RECORD
   Employed DB    ?
   Comp     Company  {}        ; UNION
   Dates    DateList ?         ; selbst definierter Typ
ENDS
```

Zunächst wird eine structure namens NameRec deklariert, die zwei Komponenten (members) hat: *Last* und *First*. *Last* ist ein Byte-Feld bestehend aus 20 Bytes, die mit dem Null-Zeichen vorbelegt werden. Dieses Feld wird den Nachnamen aufnehmen. *First* ist das Vornamen-Pendant bestehend aus einem 10-Byte-Feld. Damit ist die structure deklariert.

Zur Demonstration komplexerer Strukturen werden nun ein RECORD, ein eigener Typ sowie eine Variante deklariert, was an entsprechender Stelle ausführlicher besprochen wird. Diese Typen werden nun in einer zweiten structure eingesetzt.

Diese structure, *Person*, alloziert nun zunächst ein Datum eines »einfachen Typs«, eines DWORDS, das uninitialisiert bleibt. Danach wird die weiter oben deklarierte structure *NameRec* benutzt, die Variable *Name* zu allozieren. Da *NameRec* eine structure aus zwei Feldern mit jeweils 20 und 10 Byte ist, umfasst *Name* somit einen Speicherbereich von 30 Bytes. Auch *NameRec* bleibt uninitialisiert. Es folgt die Allozierung einer Variablen von Typ RECORD mit vier Bytes (DWORD) und einer vom Typ UNION mit 20 Byte. Sie bleiben ebenso wie die Variable vom selbst definierten Typ *DateList*, der einen DWORD-Pointer beschreibt und somit vier Bytes Umfang hat, uninitialisiert.

Nun kann ganz einfach eine Variable deklariert und initialisiert werden, die eine Person nach den deklarierten Kriterien beschreibt:

```
APerson PERSON ?
```

Das Symbol »?« sagt dem Assembler, dass unabhängig von eventuell in der Deklaration vorgegebenen Initialwerten alle Komponenten der Struktur uninitialisiert bleiben sollen. Das bedeutet, dass die Initialwerte der Deklarationen unbeachtet bleiben.

Die Symbolnamen der structure und ihrer member können Sie wie in Tabelle 3.1 dargestellt in Ausdrücken benutzen:

Name	**Operator**			
	ohne	**TYPE***	**LENGTH(OF)**	**SIZE(OF)**
StructureName	Adresse der structure	Größe der structure	Anzahl allozierter Daten des Typs	Größe der structure
MemberName	Offset des ersten Member-Bytes	Größe des Datentyps	Anzahl allozierter Daten des Typs	Größe des Members*

*: nur bei MASM und im MASM-Modus von TASM

Tabelle 3.1: Ergebnis der Verwendung der Komponenten einer structure mit und ohne Operatoren

Zwischen LENGTHOF und LENGTH bzw. SIZEOF und SIZE ergibt sich hierbei ein Unterschied (vgl. Seite 696/697 bzw. 701/701).

TYPE kann bei TASM nur im MASM-Modus eingesetzt werden, da nur dort Typen durch einen numerischen Wert beschrieben werden, der der Größe eines Datums von diesem Typen in Bytes entspricht. Die Berechnung der Größe eines members erfolgt nach der Formel

SIZE(MemberName) = LENGTH(MemberName) · TYPE(MemberName)
SIZEOF(MemberName) = LENGTHOF(MemberName) · TYPE(MemberName).

Da TYPE nur im MASM-Modus von TASM einen verwertbaren numerischen Wert zurückgibt, ist die Berechnung von SIZE auch nur in diesem Modus möglich!

Gegeben seien die Deklarationen *Beispiel*

```
AStruct STRUCT
   AString      DB 'Dies ist ein String!'   ; 20 Bytes
   BString      DB 20 DUP ('Hallo')         ; 20 · 5 Bytes
ENDS


AUnion UNION
   AByte        DB ?
   AString      DB 'Dies ist auch!'         ; 14 Bytes
   ADWord       DD 20 DUP (?)               ; 20 · 4 Bytes
ENDS


BStruct STRUCT
   OneByte      DB       ?
   TwoWords     DW       2 DUP (?)
   ThreeDoubles DD       1, 2, 3
   Array3D      DB       3 DUP (1, 2, 3, 4, 5)
   AMessage     AStruct  5 DUP ({})
   BMessage     AStruct  {}, {}, {}, {}, {}
   ADemo        AUnion   3 DUP (<>)
   BDemo        AUnion   <>, {}, ?
ENDS


Example         BStruct  2 DUP (?)
```

Dann ergibt sich für die Verwendung der *MemberNames* und *Examples* in Ausdrücken mit den aufgelisteten Operatoren:

Member	Typ	TYPE	LENGTH	SIZE	LENGTHOF	SIZEOF
OneByte	BYTE	1	1	1	1	1
TwoWords	WORD	2	2	4	2	4
ThreeDoubles	DWORD	4	1	4	3	12
Array3D	BYTE	1	3	1	15	15
AMessage	AStruct	120	5	600	5	600
BMessage	AStruct	120	1	120	5	600
ADemo	AStruct	80	3	240	3	240
BDemo	AStruct	80	1	80	3	240
Example	BStruct	1712	1	1712	2	3424

Bitte beachten Sie, dass die in der Zeile »*Example*« stehenden Werte nicht die Summen der in den jeweiligen Spalten darüber stehenden Werte sind (sein können!), da die Größe einer Allokation der Variablen *Example* die Summe aller Komponenten der Struktur ist. Sie ist daher die Summe aller in der Spalte »*SIZEOF*« für die Members stehenden Werte!

UNION ENDS

Der Assembler kennt auch den Typ »*union*«, zu deutsch: »Variante«. Seine Deklaration wird eingeleitet durch die Direktive UNION und beendet durch ENDS, »end of segment«.



Analog zu STRUCT ist UNION in Assembler das, was es in C++ auch ist: eine Variante, also ein Speicherbereich, den sich verschiedene, unterschiedliche Daten teilen. Eine *union* ist somit so groß wie das größte ihrer Mitglieder (members), das Beschreiben eines Mitglieds überschreibt somit den bisherigen Inhalt der Varianten. In Delphi und Pascal gibt es *unions* nur im Rahmen des varianten Teils eines RECORDs! Eine C++- oder Assembler-analoge Variante in Delphi besteht also nur aus dem »case ... end«-Konstrukt eines Records ohne statische Felder.

Deklaration

Die allgemeine formale Deklaration von unions lautet:

```
[Name] UNION
   UnionMembers
[Name] ENDS
```

Name ist der Symbolname, der der union gegeben wird. Er kann bei der Einleitung der Deklaration vor dem Schlüsselwort UNION sowie beim Abschluss der Deklaration vor dem Schlüsselwort ENDS stehen!



Name muss angegeben werden, wenn eine Typ-Deklaration erfolgt, mit Hilfe derer Variable alloziert werden sollen. Weggelassen werden darf *Name* nur dann, wenn die Direktive UNION im Rahmen verschachtelter Strukturen eingesetzt wird. Wird *Name* vor der Direktive ENDS weggelassen, wird das aktuell geöffnete »Segment« geschlossen. Dies ist die UNION, falls nicht innerhalb von ihr Deklarationen von anderen »Segmenten« erfolgten, die noch offen sind! In diesem Falle würde mit dem vermeintlich UNION abschließenden ENDS das noch geöffnete Segment geschlossen und vom Assembler eine noch geöffnete union moniert. Um dies auszuschließen, kann durch *Name* das von ENDS zu schließende »Segment« genannt werden.

UnionMembers nun ist eine Liste der einzelnen Komponenten, die die union ausmachen. An dieser Stelle können die Deklarationen einfacher Daten wie BYTEs, WORDs & Co. stehen, aber auch komplexe Deklarationen wie structures, unions oder records. Und selbst eigene deklarierte Datentypen können Sie verwenden.



Im Unterschied zu structures belegen alle member der union den gleichen Speicherplatz! Das bedeutet, eine union ist genauso groß wie das größte Datum, das als member alloziert wurde. Alle anderen members teilen diesen Bereich, sodass alle MemberNames für den gleichen Offset zur Union-Adresse stehen: für den Offset 0! Auf diesen reservierten Speicherbereich kann nun mit Hilfe der members unterschiedlich zugegriffen werden.



Auf diese Weise ist es möglich, Datenstrukturen zu definieren, auf deren unterschiedliche Teile mit unterschiedlichen Namen zugegriffen werden kann. Ein Beispiel: In 32-Bit-Umgebungen besteht ein FAR-Pointer aus einem 16-Bit-Selektor und einem 32-Bit-Offset. Der Befehl `LDS Reg, Mem48` erwartet als Quelloperanden eine 48-Bit-Struktur, die den Selektor und den Offset enthält. In diesen Speicherbereich muss aber einmal dieser 16-Bit-Selektor und ein 32-Bit-Offset eingetragen worden sein.

Einfach wird das Ganze, wenn man eine union deklariert:

```
FarPointer UNION
   FP     DF ?
   STRUC
      NP  DD ?
      SG  DW ?
   ENDS
ENDS
```

Die union hat zwei members: Eine Variable namens FP vom Typ FWORD und 6 Byte Umfang (48 Bit!) sowie eine structure mit den Variablen NP vom Typ DWORD (4 Byte = 32 Bit) und SG vom Typ WORD (2 Byte = 16 Bit). Somit besitzen beide member und damit die union die gleiche Größe: 6 Byte. Über FP kann nun der reservierte Speicherbereich mit den vollen 6 Byte angesprochen werden. NP spricht die »unteren« 4 Byte an, SG die »oberen« 2 Byte. (Cave: die Offset der Member beginnen bei 0 und steigen mit der Position ihrer Deklaration, sodass NP im Speicher »vor«, das heißt bei niedrigeren Offsets, SG angeordnet wird.)

Nehmen wir einmal an, dass eine Variable »`Pointer FarPointer ?`« deklariert wurde, so ist das Laden eines neuen Datensegments ein Klacks:

```
MOV     Pointer.SG, DS                ; Selektor-Teil
MOV     Pointer.NP, OFFSET StrVar     ; Offset-Teil
:
:
LES     ESI, Pointer.FP               ; gesamter Far-Pointer
```



Gegenüber der allgemeinen Form der Deklaration besitzt MASM noch zwei MASM-spezifische Argumente, die nach den Befehl UNION angegeben werden können. Unter MASM sieht somit die formale Deklaration von unions wie folgt aus:

```
[Name] UNION  [align] [, NONUNIQUE]
   UnionMembers
[Name] ENDS
```

Die optionalen, zusätzlichen Argumente sind die gleichen wie bei der Deklaration von structures (vgl. Seite 571). Sie betreffen die Ausrichtung des durch die union belegten Speicherbereiches (*align*) und die Verwendung qualifizierter Namen (*NONUNIQUE*).



TASM hat den IDEAL-Modus und somit eine weitere Form der formalen Deklaration einer union in diesem Modus. Diese unterscheidet sich jedoch wie üblich lediglich in der Reihenfolge der Angaben:

IDEAL-Modus

```
UNION [UnionName]
   UnionMembers
ENDS [UnionName]
```

Wie in IDEAL-Modus üblich, erfolgt die Deklaration eines Typs mit der Nennung des Schlüsselwortes, dem dann der Name folgt. So auch hier: Die Deklaration beginnt mit dem Schlüsselwort UNION, gefolgt durch den Namen, den Sie der union geben wollen. Auch im IDEAL-Modus folgt dann die Liste der Komponenten-Deklarationen, die durch ENDS abgeschlossen wird. Im IDEAL-Modus ist die Angabe des Namen der union mit der ENDS-Direktive ebenfalls optional.

Auch TASM erlaubt, die Komponenten der union, genauer: den Speicher, den die Komponenten gemeinsam belegen, an Grenzen auszurichten. Dies erfolgt analog zu den structures (vgl. Seite 574) mit der Direktive ALIGN.

Daten-Allozierung

Formal erfolgt die Allozierung eines Datums vom Typ UNION analog der eines Datums vom Typ STRUCT. Daher wird an dieser Stelle auf die entsprechenden Passagen verwiesen (siehe Seite 575). Dort finden sich auch Angaben zu verschachtelten Strukturen, die sinngemäß auch für unions gelten, sowie ein Beispiel, wie unions im Rahmen komplexer Strukturen verwendet werden können.



Einen Unterschied, der die Initialisierung von unions betrifft, gibt es aber doch! Er resultiert aus der Tatsache, dass die member einer union anders als die member einer structure alle am gleichen Offset beginnen. (Anders formuliert: Während structures soundsoviele members mit je einem Gesicht haben, haben unions einen member mit soundsovielen Gesichtern!).

Initialisiert werden kann der von unions belegte Speicherbereich somit nur einmal, da es ihn nur einmal gibt! Gesetzt den Fall, es existiere die Deklaration

```
DemoUnion UNION
   B  DB ?
   W  DW ?
   D  DD 4711
ENDS
```

so können Sie nur *entweder* B *oder* W *oder* D initialisieren, nicht aber B und W, W und D oder B und D oder gar alle drei! Denn welcher Wert ist denn nun der Initialwert, der in das DWORD, das die union repräsentiert, eingetragen werden soll? Initialisieren Sie daher in unions immer nur einen member und markieren Sie die anderen als uninitialisiert, andernfalls erhalten Sie Fehlermeldungen des Assemblers.

Das gilt natürlich auch für Initialisierungen, die im Rahmen der Allozierung von Daten erfolgen:

```
Demo DemoUnion {}                          ; ok
Demo DemoUnion <>                          ; ok
Demo DemoUnion {W=0815}                    ; ok
Demo DemoUnion {B=0FFH,D=012345678h}       ; Fehler!
Demo DemoUnion <0FFh>                      ; ok
Demo DemoUnion <,1,2>                      ; Fehler!
Demo DemoUnion <,,3>                       ; ok
Demo DemoUnion ?                           ; ok
```



Um vor unliebsamen Überraschungen gefeit zu sein, initialisieren Sie möglichst immer das größte Datum der union. Dann bleibt Ihnen erspart, unschöne Resultate zu erhalten, wenn Sie bei der DemoUnion beispielsweise B initialisiert haben (womit nur das niedrigstwertige Byte der Union einen definierten Wert hat), aber D auslesen (und somit 24 führende, uninitialisierte Bits im DWORD haben).

Verschachtelte Strukturen

Unions können wie structures ineinander verschachtelt werden. Das bedeutet, dass eine union in eine andere eingebettet (»nested«) ist und somit komplexere Strukturen ausbilden kann. Siehe hierzu auch die analogen Angaben zu structures auf Seite 578.



Die Symbolnamen der union und ihrer member können Sie wie in Tabelle 3.2 dargestellt in Ausdrücken benutzen:

Name	Operator			
	ohne	**TYPE***	**LENGTH(OF)**	**SIZE(OF)**
UnionName	Adresse der union	Größe der union	Anzahl allozierter Daten des Typs	Größe der union
MemberName	Offset = 0	Größe des Typs des gr. Members	Anzahl allozierter Daten des Typs	Größe des größten Members*

*: nur bei MASM und im MASM-Modus von TASM

Tabelle 3.2: Ergebnis der Verwendung der Komponenten einer union mit und ohne Operatoren

Zur Verwendung der Symbole UnionName und MemberName in Ausdrücken mit und ohne Operatoren vgl. Seite 583. Bitte beachten Sie hierbei, dass – egal welchen MemberName einer Union Sie angeben – immer nur die Daten des größten Members der union verwendet werden: Es gibt ja nur einen Member mit verschiedenen Gesichtern!

RECORD

Anders als structures und unions haben »*records*« in C++ oder Delphi keine Übereinstimmungen. Records werden verwendet, um einfache Datentypen wie Bytes, Words oder DoubleWords zu strukturieren und ihre *Bits* einzeln oder als Bitfelder ansprechen oder manipulieren zu können.



Der von Delphi her bekannte Datentyp RECORD hat nichts mit Assembler-Records zu tun! Delphis *records* entsprechen unter Assembler dem Typ STRUCT.

Deklaration

Die Direktive RECORD wird in folgender formalen Deklaration verwendet:

```
RecordName RECORD Field [, Field [, Field ...]]
```

RecordName ist hierbei wieder der Symbolname, unter dem das Record angesprochen werden soll. *Field* hat die Form

```
FieldName : Width [= Expression],
```

wobei *FieldName* ein Symbolname ist, unter dem das Bit-Feld der Breite *Width* angesprochen werden soll. Optional initialisiert wird dieses Feld durch *Expression*. Wird dieser Ausdruck nicht verwendet, sind die Bit-Felder uninitialisiert! Bei Records heißt das: mit Null vorbesetzt.

Records müssen in ihrer Gesamtgröße einem der einfachen Daten (BYTE, WORD oder DWORD) entsprechen, da sie letztlich nichts anderes als solche einfache Daten sind, bei denen lediglich einzelne Bits formal zu Feldern zusammengeschlossen wurden. Sie müssen nicht alle Bits dieser Daten für die Definition von Feldern nutzen; aber dennoch können im Speicher nur die »vollen« Bitfelder von 8, 16 oder 32 Bits, die hinter den Daten stehen, abgespeichert werden. Alle nicht besetzten Bits bleiben uninitialisiert.

Beispiel

Wenn Sie eine Zeitangabe machen wollen, so benötigen Sie für die Stunden (im 24-Stunden-Format) 5 Bits ($2^5 = 32$) und für Minuten 6 Bits ($2^6 = 64$). Das sind zusammen 11 Bits. Wenn Sie nun noch ein Datum codieren wollen, so fallen nochmals 5 Bits für die Tage und 4 Bits für die Mo-

nate an. Bleiben für das Jahr 12 Bits übrig (wenn auf 32 Bits DWORD-Größe aufgefüllt wird), was 2^{12} = 4096 entspricht – eine Jahreszahl, die wir wohl knapp verpassen und gerade nicht mehr erleben werden, weshalb das ausreichend ist. Mit diesen Vorgaben kann nun ein record deklariert werden, das die Abspeicherung von Zeit- und Datumsangaben von Christi Geburt an sehr komfortabel gestaltet:

```
TDRec RECORD Year:12=2002, Month:4, Day:5, Hour:5, Min:6
```

Damit haben Sie ein record definiert, dessen sechs Bits 5 bis 0 für die Minute stehen (denken Sie daran: Nach Intel wird immer hinten oder rechts angefangen!), die fünf Bits 10 bis 6 für die Minuten, 15 bis 11 für den Tag, 19 bis 16 für den Monat und 31 bis 20 für das Jahr. Dem Feld *Year* wird der Initialisierungswert »2002« zugeordnet, der immer dann automatisch eingesetzt wird, wenn ein Datum dieses Typs deklariert wird.

Daten-Allozierung

Nun können Sie *TDRec* als »ganz normale« Direktive zur Datenallozierung benutzen und wie die einfachen Daten vom Typ BYTE & Co. auch gleich initialisieren. Das erfolgt analog zu den »einfachen« Typen formal durch

[VarName] RecordName < [Initialisierer [, Initialisierer [, Initialisierer ...]]] >
[VarName] RecordName { [Initialisierer [, Initialisierer [, Initialisierer ...]]] }
[VarName] RecordName Const DUP ({ [[Initialisierer [, Initialisierer ...]]] })
[VarName] RecordName ?

Initialisierer ist wie bei den einfachen Typen auch ein Ausdruck der Form

```
FieldName = expression,
```

der einen für das entsprechende Feld gültigen Wert ergeben muss (vgl. Seite 558). Soll der record uninitialisiert bleiben, so geben Sie wie bei den einfachen Typen das Symbol »?« an:

```
ThisDateTime TDREC ?
```



Beachten Sie bitte, dass »`ThisDateTime TDREC ?`« nicht das Gleiche ist wie »`ThisDateTime TDREC <?>`« oder »`ThisDateTime TDREC {?}`«! Im ersten Fall wird das gesamte Datum *ThisDateTime* uninitialisiert gelassen, im zweiten Fall nur das Feld *Year* auf »uninitialisiert« gesetzt und im dritten Fall eine Fehlermeldung ausgelöst. Warum? Wir werden es sofort verstehen!

Soll dagegen die Variable mit den Initialwerten initialisiert werden, die während der Deklaration angegeben wurden, so erfolgt dies in folgender Form:

```
ThisDateTime TDREC {}
Publication  TDREC <>
```

Die leeren, mit geschweiften »{ }« oder spitzen »< >« Klammern umfassten Bereiche veranlassen den Assembler, nur die Felder zu initialisieren, die bei der Deklaration des records einen Initialwert mitbekommen haben. Alle anderen Felder bleiben uninitialisiert (was im Falle von Record-Feldern »0« heißt!). Hierbei ist es unerheblich, ob leere geschweifte oder spitze Klammern verwendet werden!

Wann dieses Buch erscheint, weiß ich noch nicht! Daher bleiben die Klammern hinter *Publication* leer. Dennoch wird das Feld *Year* in *Publication* aufgrund der Typendeklaration mit 2002, dem vermutlichen Erscheinungsjahr vorbesetzt.

Ich weiß aber, wann ich diese Zeilen schreibe. Daher kann ich die Variable *ThisDateTime* mit dem heutigen Datum und der aktuellen Uhrzeit (Oh Gott, oh Gott! ...) initialisieren Dazu kann ich die geschweiften oder spitzen Klammern verwenden, um die Initialwerte zu umklammern. Der Unterschied besteht darin, dass bei geschweiften Klammern jeweils Feld und Initialwert in der Form

```
{Field1=Init1 [, Field2=Init2 [, Filed3=Init3 ...]]]}
```

angegeben werden müssen, während bei den spitzen Klammern die Initialwerte ohne Nennung des Feldes in der Reihe der Deklaration der zu ihnen gehörenden Felder angegeben werden können, was sicherlich kompakter ist, die Lesbarkeit aber nicht positiv beeinflusst:

```
<Init1 [, Init2 [, Init3 ...]]]>
```

Ich entscheide mich dennoch für Variante 2:

```
ThisDateTime TDREC <2001, 9, 23, 1, 50>
```

Da das Jahr bereits während der Deklaration des Typs mit dem Wert 2002 vorbesetzt wurde, wird es nun im Rahmen der Variablen-Initialisierung mit 2001 überschrieben.



Die Klammern bei der Deklaration eines Datums vom einem Datentyp, der mit RECORD deklariert wurde, sind obligatorisch! Sie müssen immer angegeben werden, selbst wenn keine Initialisierungen erfolgen (siehe *Publication*) und somit der umklammerte Bereich leer bleibt. Aus-

nahme: Das Symbol »?« wird verwendet, um eine uninitialisierte Variable des Typs zu erzeugen. Dabei können Sie wählen, ob Sie die geschweiften Klammern »{ }« oder die spitzen »<>« verwenden. Im Rahmen von records in Kontrolldirektiven (wie .IF oder .WHILE) können jedoch nur die geschweiften verwendet werden!

Das Datum *Publication* hätte wie ThisTimeDate allerdings auch initialisiert werden können. Sollen nur einzelne Felder initialisiert werden, bestehen analog der vollständigen Initialisierung folgende Möglichkeiten:

```
Publication TDRec {Month = 1, Hour = 9}
```

bei der gezielt die Felder *Month* und *Hour* geändert werden, oder, gleichbedeutend

```
Publication TDRec <,1,,9>
```

Hier wird das am weitesten links stehende Feld, *Year*, nicht verändert, was der fehlende Wert vor dem ersten Komma anzeigt. Es besitzt somit weiterhin den Defaultwert 2002 aus der Record-Deklaration! Das folgende Feld *Month* wird mit 1 besetzt, das darauf folgende Feld *Day* wird ebenfalls ausgelassen, bleibt also uninitialisiert. Nachdem *Hour* auf 9 gesetzt wurde, wird die Wertzuweisung abgebrochen, *Min* bleibt uninitialisiert.

Operator Sie können *TDRec* aber auch als Operator in Instruktionen einsetzen, z.B. zum »type casting«, oder die im Record definierten Felder als Argument für den Operator MASK (vgl. Seite 698) verwenden, um Bit-Masken herzustellen, mit denen Sie Daten verändern können:

```
MOV     EAX, TDREC {2002,1,1,0,0}      ; type casting
MOV     EBX, MASK Year                 ; Maske: FFF0_0000h
AND     EAX, EBX                       ; Extraktion des Jahres
SHR     EAX, Year                      ; »Division« durch 2^20
```



Um nicht missverstanden zu werden: Records erweitern nicht den Befehlssatz der CPU oder FPU um Möglichkeiten, einzelne Bits anzusprechen oder zu Bitfeldern zusammenzufassen! Records helfen Ihnen nur, die sehr archaischen Befehlssequenzen und (kompakten) Daten ein wenig les- und handhabbarer zu gestalten. So übersetzt der Assembler das Codefragment oben und das folgende in absolut den gleichen Code:

```
MOV     EAX, 07D210800h                ; 01.01.2002, 00:00
MOV     EBX, 0FFF00000h                ; Maske für Jahr
```

```
AND     EAX, EBX                        ; Extraktion des Jahres
SHR     EAX, 20                         ; »Division« durch 2^20
```

Hätten Sie gedacht, dass $7D210800 den Jahreswechsel 2001/2002 codiert? Was, bitte, ist anschaulicher: dieses DoubleWord-Ungetüm oder das elegante *TDREC (2002, 1, 1, 0, 0)*? Und `MASK Year` und `SHR EAX, Year` erleichtern das Lesen von Assembler-Listings doch erheblich, oder?



An der Benutzung des Feldnamen *Year* im Beispiel oben im Zusammenhang mit dem Operator MASK und solo können Sie sehen, dass die Feldnamen in records im Gegensatz zu denen in structures oder unions nicht lokal (auf die Struktur bezogen), sondern global (innerhalb des Assemblermoduls) verfügbar sind und sein müssen. Das bedeutet, es darf keine gleich lautenden Variablen-, Label- oder sonstige Namen geben. Auch können Feldnamen nicht redefiniert werden.



Die Symbolnamen des records und seiner Felder können Sie wie in Tabelle 3.3 dargestellt in Ausdrücken benutzen:

Name	**Operator**			
	ohne	**SIZE(OF)**	**WIDTH**	**MASK**
RecordName	Maske im record definierter Bits	Größe des records in Bytes	Gesamtzahl Bits im record	Maske im record definierter Bits
FieldName	Nummer erstes Feldbit im record	-	Zahl der Bits des Feldes	Maske im Feld definierter Bits

Tabelle 3.3: Ergebnis der Verwendung der Komponenten eines Records mit und ohne Operatoren

Für das Beispiel von oben ergäbe sich mit TDRec somit Folgendes:

```
MOV     EAX, TDRec              ; $FFFF_FFFF: alle Bits definiert
MOV     EAX, Year               ; EAX = 20: Bit 20 1. Bit von Year
MOV     EAX, SIZE TDRec         ; EAX =  4: DWord = 4 Bytes
MOV     EAX, WIDTH TDRec        ; EAX = 32: alle Bits definiert
MOV     EAX, WIDTH Month        ; EAX =  4: 4 Bits für Feld Monat
MOV     EAX, MASK TDRec         ; $FFFF_FFFF: alle Bits definiert
MOV     EAX, MASK Day           ; $0000_F800: Day hat Bits 11 bis 15
```



Es kann vorkommen, dass eine Assemblerzeile nicht ausreicht, um ein record zu deklarieren. In diesem Fall sind Sie darauf angewiesen, mehrere Zeilen zu benutzen. Das können Sie tun, können innerhalb einer Record-Deklaration aber nicht den sonst üblichen back slash »\« als

Zeichen dafür, dass die Deklaration in der nächsten Zeile fortgesetzt wird, verwenden! Vielmehr müssen Sie den sich über mehrere Zeilen erstreckenden Block in geschweifte Klammern »{ }« einschließen, wobei das letzte Zeichen in der ersten Zeile die geöffnete geschweifte Klammer sein *muss* (Ausnahme: es folgen Kommentare!). Folgende Deklarationen sind somit korrekt:

```
Time   RECORD {          ; Zeit-Record
               Hour : 5
               Min  : 6
               Sec5 : 4   ; 5-Sekunden-Schritte
               }

Attrib RECORD Flash : 1 = 1, BackG : 3 {
               Intens: 1 = 1, ForeG : 3 }
```

TYPEDEF TYPEDEF definiert einen neuen oder erzeugt ein Synonym für einen bestehenden Typen. Die allgemeine Deklaration lautet:

```
TypeName TYPEDEF QualifiedType
```

QualifiedType kann hierbei nicht nur ein beliebiger Typ wie Datentyp, STRUCT, UNION oder RECORD sein, sondern auch ein Zeiger auf einen »qualifizierten Typen« darstellen. *QualifiedType* hat dann die Form:

```
[Distance] PTR [QualifiedType]
```

Distance ist optional und kann die Symbole NEAR (innerhalb des aktuellen Segments) oder FAR (außerhalb des aktuellen Segments) annehmen. NEAR und FAR benutzen die aktuelle (über *MODEL* eingestellte) Segmentgröße.

Beispiele

```
Char   TYPEDEF BYTE
PChar  TYPEDEF NEAR PTR BYTE
Array  TYPEDEF BYTE
PArray TYPEDEF PTR Array
FPTR   TYPEDEF FAR PTR
STRCT  TYPEDEF NestedStruct; vgl. Seite 578
PSTRCT TYPEDEF NEAR PTR STRCT
```



MASM gestattet hinter TYPEDEF ein Schlüsselwort, mit dem man einen Typen des Prototypen einer Prozedur deklarieren kann:

```
TypeName TYPEDEF PROTO Definition
```

wobei *Definition* wie unter PROTO beschrieben deklariert ist (vgl. Seite 614):

```
[Distanz] [Sprache] [Parameter[:Tag] ,...]
```



TASM verfügt wiederum über zwei Besonderheiten: Zum einen verwendet es für die Deklaration von Prozedur-Prototypen die Direktive PROCTYPE (siehe folgender Abschnitt), zum anderen besitzt es ja über den MASM-Modus hinaus noch den IDEAL-Modus. So wundert es wohl niemanden, wenn es für diesen Modus auch eine eigene Syntax gibt:

```
TYPEDEF TypeName QualifiedType
```

PROCTYPE



PROCTYPE ist die TASM-Version zur Deklaration von prozeduralen Typen. MASM verfügt hierzu über die Syntax-Erweiterung PROTO der Direktive TYPEDEF (siehe vorangehenden Abschnitt).

Natürlich verfügt TASM wiederum je nach aktuellem Modus über zwei unterschiedliche Formen der Syntax:

```
TypeName PROCTYPE Definition
```

MASM-Modus

Für den IDEAL-Modus besteht der Unterschied wie immer in einem Vertauschen der Reihenfolge des Typnamen und des Schlüsselwortes PROCTYPE:

```
PROCTYPE TypeName Definition
```

IDEAL-Modus

wobei *Definition* in beiden Fällen wie folgt beschrieben ist:

```
[[Modifizierer] Sprache] [Distanz] [Parameterliste]
```

Zu *Modifizierer*, *Sprache* und *Distanz* siehe PROTO auf Seite 614, zu *Parameterliste* siehe ARG auf Seite 617.

ENUM



ENUM erlaubt die Deklaration eines Aufzählungstypen. ENUM ist eine Syntaxerweiterung des TASM und hat unter MASM keine Entsprechung.

Mit ENUM ist es möglich, einzelnen Bits eines Bytes, Words oder DoubleWords ein Symbol zuzuordnen, unter dem es ansprechbar ist. Insofern ähnelt es der Deklaration eines RECORD-Typen, bei dem ja ganze Bitfelder eines Datums einem Symbol zugeordnet werden.

Natürlich verfügt TASM wiederum je nach aktuellem Modus über zwei unterschiedliche Formen der Syntax:

MASM-Modus
```
EnumName ENUM [Symbol1 [, Symbol2 [, ... ]]]
```

IDEAL-Modus
```
ENUM EnumName [Symbol1 [, Symbol2 [, ... ]]]
```

wobei *SymbolN* in beiden Fällen wie folgt beschrieben ist:

```
SymbolName [= Wert]
```

ENUM inkrementiert in der Reihenfolge der Nennungen der Symbole 1 bis N die Werte, die diesen Symbolen zugeordnet werden, automatisch um 1, wenn nicht über die direkte Wertzuweisung gearbeitet wird.

Beispiel
```
DayOfWeek ENUM Sonntag=0, Montag, Dienstag, Mittwoch, {
               Donnerstag, Freitag, Samstag
               IrgendeinWochentag=98, unbekannt }
```

Die Deklaration nutzt ein BYTE zur Speicherung von Daten des Typs *DayOfWeek*, da mit der expliziten Zuordnung von 98 an den Symbolnamen *IrgendeinWochentag* und der sich anschließenden Nennung von *unbekannt* (= 99) der Maximalwert eines Bytes (= 255) nicht überschritten wird. Durch die Deklaration wird dem Symbol *Sonntag* der Wert »0« zugeordnet, *Montag* der Wert »1« usw. bis *Samstag* = »6«. Die Werte 7 bis 98 bleiben unbenutzt, da *IrgendeinWochentag* die explizite Wertzuweisung »= 98« erhält. *Unbekannt* erhält dann den Wert »99«, die Werte »100« bis »255« bleiben unbenutzt.

Daten-Allozierung

Nun können Sie DayOfWeek als »ganz normale« Direktive zur Datenallozierung benutzen und wie die einfachen Daten vom Typ BYTE & Co. auch gleich initialisieren. Das erfolgt analog zu den »einfachen« Typen formal durch

[VarName] EnumName SymbolN
[VarName] EnumName Const DUP (SymbolN)
[VarName] EnumName ?

in diesem Beispiel also

```
Today DayOfWeek Sonntag
```



ENUM ist somit nur eine Vereinfachung, die der Zuordnung von Werten an Symbole dient, die im Rahmen von Wertzuweisungen an Variablen benutzt werden sollen. Analoges ist daher auch mit den Direktiven *EQU* und = machbar, weshalb MASM auch auf die Möglichkeiten von ENUM verzichtet. So ist die Deklaration von *DayOfWeek* identisch mit der Summe der Deklarationen.

```
Sonntag             EQU  0
Montag              EQU  1
Dienstag            EQU  2
:
:
Samstag             EQU  6
IrgendeinWochentag EQU 98
Unbekannt           EQU 99
```

und die Datenallokation könnte erfolgen mit Hilfe von

```
VarName BYTE Symbol
```

im Beispiel also

```
Today BYTE Sonntag
```



Die mit ENUM deklarierten Symbole sind global sichtbar, was bedeutet, dass sie im gesamten Modul zur Verfügung stehen. Dies bedeutet, nachdem sie redefinierbar sind, dass Fehler auftreten können, wenn in einem Assemblermodul z.B. im Rahmen oder aufgrund der Deklaration von Aufzählungstypen zweimal das gleiche Symbol deklariert und benutzt wird!



Die Symbolnamen des Aufzählungstypen und seiner Symbole können Sie wie in Tabelle 3.4 dargestellt in Ausdrücken benutzen:

Name	**Operator**			
	ohne	**SIZE(OF)**	**WIDTH**	**MASK**
EnumName	Maske in ENUM benutzter Bits	Größe des Datums in Bytes	Anzahl benötigter Bits	Maske in ENUM benutzter Bits
SymbolName	Wert	-	-	-

Tabelle 3.4: Ergebnis der Verwendung der Komponenten eines Aufzählungstypen mit und ohne Operatoren

TABLE
TBLINIT
TBLINST
TBLPTR

Die Erweiterung der Assembler-Syntax zur Realisierung von Objektorientierter Programmierung (OOP) erforderte auch die Möglichkeit zur Deklaration und Verwaltung von (Methoden-)Tabellen.



Wenn Sie also mit TASM ein Objekt deklarieren, erzeugt TASM nicht nur eine STRUC (vgl. Seite 570), sondern auch eine TABLE, in der die Deklarationen der zum Objekt gehörenden Methoden stehen. Diese TABLE entspricht der VMT (virtual method table) von Hochsprachen-Objekten.

TBLINST ist eine Direktive, die eine Instanz der VMT erzeugt und damit Voraussetzung für die Verwendung von Objekten in Assembler ist. TBLINIT dagegen initialisiert diese VMT.



Bitte haben Sie Verständnis, wenn ich im Rahmen dieses Buches nicht auf die OOP unter Assembler und daher auf Einzelheiten der Direktiven TABLE, TBLINIT, TBLINST und TBLPTR eingehen kann. Falls Sie hierzu Informationen benötigen, verweise ich Sie auf die Handbücher und Online-Informationen, die mit TASM ausgeliefert werden.

3.2.3 Direktiven zur Symboldeklaration

EQU EQU ist eine Direktive, mit der einem Symbol(-Namen) ein Wert zugeordnet werden kann. Dieser Wert kann entweder ein numerischer Wert sein oder ein Text. Handelt es sich um eine numerische Wertzuweisung, so ist jeder gültige Ausdruck der Form expression erlaubt:

Deklaration
```
Name EQU expression
```

Kann der Ausdruck dagegen nicht eindeutig berechnet werden (ungültiger Ausdruck), so wird er dem Symbol als Text zugeordnet. Soll dagegen dem Symbol ein Text zugeordnet werden, wird er in spitze Klammern eingeschlossen:

```
Name EQU <expression>
```

Hier wird dem Symbol *Name* nicht das Ergebnis des Ausdrucks *expression*, sondern die Zeichenfolge *»expression«* zugeordnet.

```
ValidB EQU 5                    ; =  5
ValidN EQU 3 * 4 + (5 / ValidByte); = 13
InvalN EQU 3 * MyDate           ; = "3 * MyDate"
ValidT EQU <3 * ValidByte>      ; = "3 * ValidByte"
ValidS EQU <"String">           ; = ""String""
```



Mit EQU deklarierte Symbole können im Gegensatz zur Deklaration mittels »=« nicht redefiniert werden!



In TASM und MASM bis Version 5.1 können mit EQU auch Alias deklariert werden. Dies ist in MASM-Versionen > 5.1 nicht mehr möglich!

=

Im Unterschied zu mit EQU deklarierten Symbolen können solche, die mit »=« deklariert wurden, beliebig häufig umdeklariert werden. Allerdings können hiermit nur numerische Symbole deklariert werden:

Deklaration

```
Name = expression
```

```
ValidB = 5                          ; =  5
ValidN = 3 * 4 + (5 / ValidByte)    ; = 13
InvalN = 3 * MyDate                 ; FEHLER!
ValidN = 20                         ; ok, redefiniert
InvalT = String                     ; FEHLER
InvalS = <String>                   ; FEHLER
```

TEXTEQU

TEXTEQU ist eine Variante von EQU, die sich ausschließlich mit Strings beschäftigt:

Deklaration

```
Name TEXTEQU textexpression
```

Textexpression kann hier ein String sein, der entweder als String (ohne spitze Klammern!) angegeben wird, von einem Textmakro zurückgegeben wird oder mittels des Operators % aus einem numerischen Ausdruck erzeugt wird. TEXTEQU führt im Gegensatz zu EQU bei der Assemblierung eine Substitution von Makros durch.

```
String1 TEXTEQU String          ; = "String"
String2 TEXTEQU @code           ; = "_TEXT"
String3 TEXTEQU %(3 * 4 + 5)    ; = "17"
```

Die Größe von mittels TEXTEQU deklarierten Konstanten darf nicht größer als 255 Zeichen sein.



CATSTR
INSTR
SIZESTR
SUBSTR

Bei diesen Direktiven handelt es sich um »Sonderformen« der Direktiven TEXTEQU oder EQU. Sie gestatten, Strings zu verbinden und einem Symbol analog TEXTEQU zuzuordnen (CATSTR), die Position eines Strings in einem zweiten zu suchen und das Ergebnis einem Symbol via EQU zuzuordnen (INSTR), die Größe eines Strings zu ermitteln und einem Symbol via EQU zuzuordnen (SIZESTR) oder aus einem String an einer Position eine bestimmte Anzahl von Zeichen zu kopieren und via TEXTEQU einem Symbol zuzuordnen (SUBSTR).

Dir Direktiven werden wie folgt verwendet:

Verwendung

```
Name CATSTR  String1, String2
Name INSTR   [Start,] String, Maske
Name SIZESTR String
Name SUBSTR  String, Start [, Länge]
```

Hierbei ist *Name* ein im Modul einzigartiger Symbolname und *String*, *String1* und *String2* ein String, der in spitzen Klammern angegeben werden muss. *Maske* ist ebenfalls ein String und bezeichnet die Zeichenfolge, die in *String* gefunden werden soll. Mit *Start* kann die Stelle im String angegeben werden, an der die Suche bzw. das Kopieren beginnen soll. Findet INSTR *Maske* in *String* nicht, wird »0« zurückgegeben. *Länge* ist die Anzahl von Zeichen, die kopiert werden sollen. Wird *Länge* nicht angegeben, werden alle verbleibenden Zeichen des Strings ab Position *Start* kopiert.



Es gibt unter MASM auch die vordefinierten Symbole @CatStr, @InStr, @SizeStr und @SubStr, die im Rahmen von Ausdrücken eingesetzt werden und Ähnliches bewerkstelligen. Siehe hierzu Seite 713.

LABEL LABEL ordnet einem Symbol Inhalt und Typ des Datums am aktuellen *location counter*, also an der Adresse des aktuellen Statements im entsprechenden Segment zu. Bei Codesegmenten handelt es sich um die Adresse der aktuellen Instruktion, bei Datensegmenten um die Adresse des aktuellen Datums.

Deklaration Formal erfolgt die Deklaration eines Labels nach

```
Name LABEL Qualifizierter Typ
```

Name ist ein Symbolname, der noch nicht im Assemblermodul verwendet worden sein darf (»unique«).

Qualifizierter Typ ist hier ein Typ nach der Definition von Seite 560.



TASM erlaubt auch bei Labels aufgrund der beiden Modi zwei Möglichkeiten der Deklaration. So ist neben der oben genannten Form, die für den MASM-Modus gültig ist, auch die für den IDEAL-Modus reservierte Form

```
LABEL Name Qualifizierter Typ
```

realisiert.



Die Sichtbarkeit von Labels ist auf das Assemblermodul beschränkt, das heißt, es ist nur innerhalb des aktuellen Moduls verfügbar. Soll auch »von außen«, also aus anderen Modulen heraus, ein Zugriff auf das Label möglich sein, so muss es mittels der Direktive PUBLIC (vgl. Seite 621) nach außen hin sichtbar gemacht werden.

Die Direktive »:« deklariert ein Label, das nur innerhalb von Codesegmenten verwendet werden kann (»near code label«). Sie wird in folgender formalen Deklaration verwendet: **:**

```
Name:
```
Deklaration

Name ist ein Symbolname, der noch nicht im Assemblermodul verwendet worden sein darf (»unique«).

Ein mittels »:« erzeugtes Label ist das Gleiche, als hätte man es mit LABEL NEAR erzeugt:



```
A:
```

und

```
A LABEL NEAR
```

sind identisch

Sie können *code labels* im Quelltext unmittelbar vor die Instruktion platzieren, auf die das Label zeigen soll, oder aber innerhalb einer eigenständigen Zeile, die der Instruktion vorangeht auf die sich das Label bezieht:



```
LBL1:   MOV     EAX, 012345678h       ; in gleicher Zeile
        JMP     LBL2
        :
        :
LBL2:                                 ; vor der gelabelten Instrukt.
        XOR     EAX, EBX
```

Wie bei allen Labels ist die Sichtbarkeit von mittels »:« erzeugten Labels auf das Assemblermodul beschränkt, das heißt, es ist nur innerhalb des aktuellen Moduls verfügbar. Solchermaßen deklarierte Labels können unter MASM auch nicht mittels der Direktive PUBLIC (vgl. Seite 621) nach außen hin sichtbar gemacht werden, unter TASM dagegen sehr wohl.



Unter der defaultmäßig eingestellten Option NOSCOPED (vgl. OPTION auf Seite 679) ist die Sichtbarkeit global im Assemblermodul. Wird dagegen die Option SCOPED gesetzt, so ist ein mittels »:« erzeugtes Label innerhalb von Prozeduren (PROC / ENDP, vgl. Seite 679) lokal in der Prozedur sichtbar, außerhalb von Prozeduren global im Modul.



:: Der doppelte Doppelpunkt deklariert Labels analog dem einfachen mit der Ausnahme, dass solchermaßen erzeugte Labels unabhängig von der Stellung der Option SCOPED grundsätzlich global im Modul sichtbar sind und mittels PUBLIC nach außen hin sichtbar gemacht werden können.



Da mit »:« deklarierte Labels unter TASM auch mit PUBLIC nach außen sichtbar gemacht werden können, gibt es keinen wirklichen Grund dafür, dass TASM auch die Direktive »::« unterstützt. Daher können Sie unter TASM »::« nur im MASM-Modus verwenden und nur dann, wenn Sie mittels VERSION (vgl. Seite 686) Kompatibilität zu MASM 5.1 hergestellt haben!

Reichweite der Labels

Üblicherweise sind Labels, die mit der Direktive »**:**« deklariert wurden, im gesamten Modul sichtbar (*code labels*). Das bedeutet, dass z.B. ein solchermaßen in einer Prozedur deklariertes Label auch außerhalb der Prozedur sichtbar ist, weshalb der Name dieses Labels einzigartig sein muss.



Zwar gelten »**:**«-Labels in MASM als nur innerhalb des deklarierenden Blocks (z.B. Prozedur) sichtbar, wenn eine Sprachangabe mit der **.**MODEL-Direktive (vgl. Seite 639) erfolgte oder die OPTION NOSCOPED gewählt wurde. Jedoch ist dies nicht ganz korrekt. Richtig ist, dass in diesen Fällen die Labelnamen nicht einzigartig sein müssen und daher in jedem Block ein Label *Name***:** deklariert werden kann. Bezüge auf diese Labels (z.B. Sprungbefehle) gelten dann jeweils nur für das im gleichen Block deklarierte Label *Name***:**. Allerdings dürfen außerhalb der Blöcke keine Symbole deklariert werden, die nicht code labels sind und *Name* heißen. Insofern ist die Sichtbarkeit nicht wirklich blockorientiert!

Es ist nun aber eventuell interessant, tatsächlich Labels zu erstellen, die nur blockorientiert sichtbar sind und deren »Reichweite« somit auf den Block beschränkt sind. MASM und TASM gehen hier verschiedene Wege.



TASM ermöglicht das, indem dem Namen *Name* des Labels zwei AT-Zeichen (»@«) vorangestellt werden:

```
@@Name:
```

Ein solchermaßen deklariertes Label ist tatsächlich nur lokal sichtbar. Es kann wie alle anderen Labels verwendet werden.



MASM benutzt auch die beiden AT-Zeichen, gestattet aber keinen Namen. Vielmehr generiert MASM einen einzigartigen Namen, indem es den AT-Zeichen eine automatisch vergebene hexadezimale Zahl folgen lässt. Da diese für den Programmierer nicht sichtbar ist, müssen zwei vordefinierte Symbole benutzt werden, die sich auf das jeweils vorangehende (@B) bzw. folgende (@F) lokale Label beziehen:

```
        :
        JMP     @B  ; Fehler, kein vorangehendes lokales Label
        JMP     @F  ; Sprung vorwärts zum nächsten lokalen Label
@@:     :           ; erstes lokales Label
        :
        JMP     @B  ; Sprung rückwärts zum vorangehenden Label
        JMP     @F  ; Sprung vorwärts zum nächsten lokalen Label
@@:     :           ; zweites lokales Label
        :
        JMP     @B  ; Sprung rückwärts zum vorangehenden Label
        JMP     @F  ; Fehler: kein folgendes lokales Label
```



Das bedeutet, die lokale Sichtbarkeit bei MASM wird dadurch erreicht, dass man dem Programmierer nur zwei vordefinierte Symbole zur Verfügung stellt, mit denen er in Blöcken navigieren kann, ansonsten aber global sichtbare Labels deklariert, die nur deshalb nicht global sichtbar sind, weil der Programmierer sie nicht kennt! Sequenzen, die in TASM durchaus realisierbar sind, wie etwa

```
        :
        JZ      @@L2
@@L1:   :
        JNZ     @@L3
@@L2:   :
        JMP     @@L1
@@L3:   :
```

sind daher unter MASM nicht möglich!

ALIAS

TASM erlaubt die Deklaration von Alias-Namen für Symbole. Dies erfolgt formal nach:



```
AliasName ALIAS OrgName
```

3.2.4 Direktiven zur Daten- und Codeausrichtung

ALIGN ALIGN dient dazu, den »location counter« auf einen Wert zu setzen, der eine ganzzahlige Potenz von 2 ist. Das bedeutet, dass das nächste Datum an einer Adresse beginnt, die durch die betreffende ganzzahlige Potenz von 2 ohne Restbildung teilbar ist. Man sagt dazu, die Adresse sei (»an einer 2^x-Byte-Grenze«) »ausgerichtet«.

Dies betrifft sowohl Daten im Codesegment, also auch Befehlssequenzen, als auch Daten in Datensegmenten, in denen die ALIGN-Direktive wohl am verbreitetsten ist. Zum Ausrichten von Befehlssequenzen werden die füllenden Bytes in Segmenten, die Befehlssequenzen enthalten (»Codesegmente«) mit NOPs ausgefüllt, bei der Ausrichtung von »echten« Daten in Segmenten ohne Instruktionen werden Null-Bytes verwendet.

Verwendung ALIGN wird formal wie folgt verwendet:

```
ALIGN expression
```

wobei *expression* ein Ausdruck ist, der ein ganzzahliges Vielfaches von 2 ergeben muss, also 1 (2^0), 2 (2^1), 4 (2^2), 8 (2^3), 16 (2^4), etc. Wird ALIGN ohne *expression* angegeben, wird als Ausrichtungswert der Wert verwendet, der bei der Deklaration des Segments angegeben wurde.

ALIGN steht im Quelltext unmittelbar vor der Stelle, die als Nächstes ausgerichtet werden soll:

```
CODE SEGMENT PARA PUBLIC      ; Codesegment an PARA-Grenzen
       :
       :
       JMP    @ABC
       ALIGN 4                ; Neue Ausrichtung : DWORD
DateC DD     ?                ; Datum im Codesegment
ABC:   ALIGN 16               ; Ausrichtung wieder PARA
       :
CODE ENDS

DATA SEGMENT DWORD PUBLIC     ; Datensegment an DWORD-Grenzen
       :
       :
       ALIGN 0                ; Neue Ausrichtung: BYTE
D1     DB    ?                ; Ohne Align würde nun jedes
D2     DB    ?                ;    Byte and DWORD-Grenzen aus-
D3     DB    ?                ;    gerichtet, so an BYTE-Grenzen;
D4     DB    ?                ;    wozu es auch immer gut sein mag
```

```
        ALIGN 2         ; Alte Ausrichtung (DWORD)
        :
        :
```



ALIGN hat nur dann eine Wirkung, wenn der aktuelle location counter, also die Adresse, an der das nächste Befehls- oder Daten-Byte eingesetzt werden soll, nicht bereits eine Adresse ist, die die Ausrichtungsbedingungen erfüllt und somit »ausgerichtet« ist.



Falls Sie ganze Segmente z.B. mit Hilfe der Argumente ausrichten, die Sie bei der Deklaration eines Segments angeben können (z.B. `CODE SEGMENT PARA PUBLIC`, wobei das Argument *PARA* für eine Ausrichtung an »Paragraphen-Grenzen =16-Byte-Grenzen steht), können Sie mit ALIGN diese Ausrichtung nicht »lockern«, indem Sie sie z.B. auf 32-Byte-Grenzen setzen (`ALIGN 32`)! Gültige ALIGN-Argumente sind daher nur Werte, die kleiner oder gleich der gewählten Segmentausrichtung sind.



Was Sie aber tun können, ist eine gewählte Segmentausrichtung mit ALIGN zunächst zu »verschärfen«, um sie später wieder zu »lockern« (oder auch nicht!):

```
CODE SEGMENT PARA PUBLIC
        :
        :
        ALIGN 4
        :
        :
        ALIGN 8
        :
        :
        ALIGN 16
        :
        :
```

EVEN

EVEN ist ein Synonym für ALIGN 2 und richtet an Word-Grenzen (»gerade« Adressen) aus.



TASM kennt neben EVEN auch die Direktive EVENDATA (vgl. Seite 606). EVENDATA sollte nur verwendet werden, wenn keine Notwen-

digkeit zur Kompatibilität mit MASM besteht, da MASM keine vergleichbare Direktive besitzt.

EVENDATA



EVENDATA richtet wie EVEN (vgl. vorangehenden Abschnitt) Daten an WORD-Grenzen aus, ist somit praktisch ein Synonym für EVEN. Der Unterschied besteht jedoch darin, dass EVENDATA anders als EVEN die entstehenden »Lücken« nicht mit NOPs oder Null-Bytes auffüllt, sondern sie uninitialisiert lässt. Daher kann EVENDATA anstelle von EVEN in uninitialisierten Datensegmenten eingesetzt werden.

ORG ORG veranlasst den Assembler, den »location counter«, also den Zeiger auf den nächsten freien Platz im Segment, an dem eine Instruktion oder ein Datum eingesetzt wird, auf einen bestimmten Wert zu setzen. Üblicherweise wird der location counter automatisch vom Assembler verwaltet. ORG nun lässt den Programmierer »manuell« Einfluss ihn nehmen.

Verwendung ORG wird formal wie folgt verwendet:

```
ORG [$+]expression
```

wobei *expression* ein Ausdruck sein muss, der eine Konstante (also ein absoluter Wert als Adresse) oder einen zu einem Label relativen Offset als numerischen Wert ergibt. Soll die mit ORG eingestellte Adresse relativ zum aktuellen Inhalt des location counter angegeben werden, so ist das Argument »*$+*« zu verwenden. In diesem Fall addiert der Assembler *expression* Bytes zum aktuellen location counter.



Ein Beispiel für die Verwendung von ORG: In alten 16-Bit-COM-Dateien lag an den Offsets 0 bis 255 des Segments der »program segment prefix«, der wesentliche Informationen beinhaltete, die das Betriebssystem DOS benötigte: die einem Programm übergebenen Kommandozeile beispielsweise, die Adresse der environment table etc. Befehle und Daten konnten somit frühestens ab Offset 256 ($100) stehen. ORG nun ermöglichte dies, indem der location counter unmittelbar nach der Deklaration des Segments auf den Wert $100 gesetzt wurde, wie in folgendem Beispiel:

```
Demo    SEGMENT WORD PUBLIC 'CODE'     ; ausgerichtet an WORD-Grenzen
        ASSUME CS:Demo, DS:Demo        ; vgl. Seite 635
        ORG 00100h                     ; beginne an Offset $100
                                       ; dazwischen uninitialisiert
Hier:   JMP     Los                    ; Dies ist Offset $1000
                                       ; Sprung über den Datenteil!
```

```
Data1   DB      ?                       ; Daten im Codesegment
Data2   DW      ?
Los:    :                               ; eigentlicher Programmstart
        :
        MOV     AX, 04C00h              ; Ende: Exit-Funktion $4C des
        INT     021h                    ; DOS-Int $21; Exit-Code: 00h
Demo    ENDS                            ; Segment zu Ende
END     Hier                            ; Einsprungpunkt "Hier"
```

Ohne die ORG-Direktive hätte das Label »Hier« den Segment-Offset 0000h, mit ORG hat es nun den Offset 0100h!

Innerhalb einer STRUCT-Direktive bezieht ORG den neuen location counter auf den Beginn der Struktur, nicht auf den Beginn des Segments wie außerhalb von Strukturen!



Beispiel

Ein etwas »realitätsnäheres« Beispiel im Zeitalter von 32-Bit-Multitasking-Multiprozessor-Betriebssystemen im protected mode entnehme ich der Dokumentation von TASM, da es ein schönes Beispiel für die Deklaration und Nutzung von verketteten Listen ist, und passe es meinen didaktischen Qualitäten gemäß etwas an ;-) Hier wird ORG im Rahmen eines Makros (vgl. Seite 656) eingesetzt, das bei der Einrichtung und Verwaltung einer doppelt verketteten Liste verwendet wird:

Zunächst wird die structure deklariert, die vom Mako verwendet wird:

```
AStruct STRUCT
   Prev DD   ?
   Next DD   ?
   Data DB   100 DUP (0)
AStruct ENDS

LHead   DD   ?
LTail   DD   ?
```

Die structure besteht aus zwei Zeigern, die die doppelt verkettete Liste realisieren, Prev und Next, die später die Adressen des vorangehenden und folgenden Elements der Liste aufnehmen. Dann werden zwei Zeiger alloziert, die später die Adresse von Anfang und Ende der verketteten Liste aufnehmen.

Als Nächstes wird das Makro deklariert. Es besitzt zwei Argumente: *name*, den Namen des Labels, unter dem das mit dem Makro erzeugte Listenelement später angesprochen werden kann, und *args*, die Argumente, mit denen der structure member *Data* von *name* belegt wird. Die

structure members *Prev* und *Next* werden vom Makro verändert werden.

Erforderlich wird auch ein Symbol, *LastN*, das den Namen des jeweils letzten allozierten Listenelements aufnimmt, damit das Makro die Verkettung (*Prev* und *Next*) herstellen kann. Es wird zunächst mittels EQU (vgl. Seite 598) auf »leer« gesetzt:

```
LastN   EQU  <>

Element MACRO name:REQ, args
IFIDNI LastN, <>
   name AStruct <0, 0, args>
        ORG  LHead
        DD   name
        ORG  LTail
        DD   name
        ORG  name + SIZE AStruct
LastN   EQU  name
ELSE
   name AStruct <LastN, 0, args>
        ORG  LastN.Next
        DD   name
        ORG  LTail
        DD   name
        ORG  name + SIZE AStruct
LastN   EQU  name
ENDIF
Element ENDM
```

Das Makro verwendet *Bedingte Assemblierung*: Wenn *LastN* leer ist, was am Anfang durch die Deklaration von *LastN* der Fall ist, so wird der zwischen IFIDNI (*if identical, ignore letter case*) und ELSE stehende Teil verwendet, andernfalls der zwischen ELSE und ENDIF stehende.

Falls also *LastN* leer ist, alloziert das Makro mit dem als Argument übergebenen *name* als Label einen Speicherbereich vom Typ *AStruct* und initialisiert es mit den Zeigern 0 für *Next* und *Prev* und den in *args* stehenden Daten für *Data*. Das Datum ist hiermit alloziert, es geht nun nur noch darum, die Einträge für den Anfang und das Ende der Liste in die Variablen *LHead* und *LTail* einzutragen und den Namen des Listenelements zu speichern, um die Verkettungen herstellen zu können, falls weitere Listenelemente folgen.

Für die Veränderung der Inhalte von *LHead* und *LTail* wird nun der location counter mit Hilfe der Direktive ORG zunächst auf *LHead*, später auf *LTail* gesetzt. An die durch den location counter bezeichnete Stelle

wird jeweils der Offset des Labels *name* eingetragen, also die Adresse des ersten allozierten Speicherbereichs, der unter dem Namen *name* fassbar ist und dem Makro übergeben wurde. Dies erfolgt durch `DD name`. Danach wird der location counter an die ursprüngliche Stelle zurückgesetzt. Diese war ja nach der Allozierung der structure die Adresse hinter der aktuell bearbeiteten structure, also (*name* + Größe der structure). Abschließend erhält *LastN* den Namen name der aktuellen structure.

Nach der Allozierung des ersten Listenelementes besitzen somit *LHead* und *LTail* den gleichen Wert: Die Adresse von *name*. *Name* ist alloziert und hat die Inhalte 0 für *Prev* und *Next*, was ja korrekt ist, da die Liste erst ein Element enthält, und die vorgegebenen Werte für *Data*. *LastN* hat den »Wert« *name*.

Wird nun das Makro ein weiteres Mal aufgerufen, so ist *LastN* nicht mehr leer. Daher kommt nun der zweite Teil des Makros zum Tragen. Als Erstes wird wiederum der Speicherbereich alloziert und mit dem Label *name* versehen. Das erfolgt ganz analog zum ersten Listenelement mit einer Ausnahme: Der structure member *Prev* erhält nun die Adresse des zuvor allozierten Listenelements, dessen Name ja in *LastN* steht. Die Rück-Verkettung ist damit hergestellt. *Next* ist 0, da noch kein nächstes Listenelement alloziert wurde.

Zur Verkettung (nach vorne!) muss die im structure member *Next* der vorangegangenen Struktur stehende Adresse verändert und mit der Adresse der aktuellen structure belegt werden. Hierzu wird wiederum mit ORG der location counter gesetzt. Die erforderliche, korrekte Adresse erhält die Direktive ORG über das Argument `LastN.Next`. Es enthält die Adresse des vorangehenden, in *LastN* stehenden Listenelements. Der DOT-Operator in Verbindung mit dem Namen des structure members *Next* addiert zu dieser Adresse den Offset, den *Next* in der structure hat. Ergebnis: die Adresse des DWORDS von *Next* in der vorangehenden Struktur. Dort wird nun die Adresse des aktuellen Listenelements mittels `DD name` eingetragen. Bleibt noch, *LTail* zu verändern, da dies ja nun auf das Listenende, also die Adresse des aktuellen Elements zeigen muss, und den location counter anschließend auf den aktuellen Wert zu setzen. *LastN* mit dem neuen *name* aktualisiert, und ... fertig!

Dieses Makro kann nun sehr gut und effizient verwendet werden, ohne dass Sie sich weiter Gedanken darum machen müssten, die Elemente der Liste zu aktualisieren und neu zu verketten oder bestimmte Reihenfolgen von Datenallokationen einzuhalten:

```
DWOne   DD   ?
Element One,<>     ; uninitialisierter Data member
DWTwo   DD   ?
WOne    DW   ?
Element Two,<'Hi, Fan!'>
Element Three,<1, 2, 3, 4, 5, 6>
WTwo    DW   ?
Element Four, <'Call me: ',0190, 0815, 4711>
```

Versuchen Sie einmal Analoges (inklusive der notwendigen Vorwärts-Referenzierung für Next) ohne ORG zu erreichen!

3.2.5 Direktiven zur Deklaration und Nutzung von Prozeduren

In diesem Kapitel werden Direktiven angesprochen, die eine einfache Erstellung und Nutzung von Prozeduren ermöglichen, so wie man es von Hochsprachen gewöhnt ist. So kann mit PROC eine gesamte Prozedur wie in einer Hochsprache deklariert werden, ohne das sich der Programmierer über das Ansprechen von Parametern oder lokalen Variablen Gedanken machen muss. Mittels INVOKE / CALL kann dann eine solche Prozedur einfach aufgerufen werden, ohne sich um die Reihenfolge und Art der zu übergebenden Parameter Gedanken machen zu müssen.



Natürlich ist es jedermann selbst überlassen, hardcore zu programmieren und an alles selbst zu denken. So kann jeder selbst die erforderlichen Parameter auf den Stack PUSHen, dann einen CALL der Routine durchführen und den Prolog und Epilog der Prozedur selbst schreiben. Wer jedoch von Hochsprachen kommt, wird die hier besprochenen Vereinfachungen zu schätzen wissen.



Übrigens: Unter Assembler gibt es nur Prozeduren! Die Unterscheidung zwischen Prozeduren und Funktionen, die Hochsprachenprogramme vornehmen, sowie den Oberbegriff zu beiden Unterprogrammformen, »Routine«, kennt der Assembler nicht! Was auf den ersten Blick verwunderlich ist, ist eigentlich gar nicht so falsch: Von Assemblerebene aus betrachtet gibt es keinen Unterschied zwischen Prozeduren und Funktionen! Beide werden mit CALL gerufen, können Parameter und/oder lokale Variable haben und werden mit RET abgeschlossen. Dem Assembler ist dabei vollkommen egal, ob irgendwann im Verlauf der »Prozedur« irgendein Register oder eine Speicherstelle mit irgendeinem Wert belegt worden ist, der dann in der rufenden

Routine (als »Funktionsergebnis«) verwendet wird. Die Unterscheidung zwischen *Prozedur* (Routine ohne übergebenes Funktionsergebnis) und *Funktion* (Routine mit übergebenem Funktionsergebnis) ist somit Interpretationssache, die auf einer höhere Ebene erfolgt als auf Assemblerebene (eben in der Hochsprache). Oder im Kopf des Assemblerprogrammierers! Wie gesagt, unter Assembler ist alles eine Frage der Interpretation!

Wir werden uns dem anpassen (da wir ja flexibel sind!) und im Rahmen der Besprechung des Stand-alone-Assembler jede Form von Unterprogramm, wie der Assembler, als Prozedur bezeichnen, wohl wissend, dass man damit Prozeduren und Funktionen realisieren kann. (Ich fände es einfach nicht passend, im Kontext der Direktive PROC, die die Deklaration einer *PROCedure* gestattet, von *Routinen* zu sprechen.)

PROC
ENDP

PROC ist eine Direktive, mit der man Assembler-Prozeduren so gestalten kann, dass sie den Konventionen einer Hochsprache genügen und aus einer Hochsprache problemlos aufgerufen werden können. Diese Konventionen umfassen einen Prozedur-Pro- und Epilog, der Prozedur übergebene Parameter, von der Prozedur zurückgegebene Funktionswerte und/oder lokale Variablen.

Deklaration

Die formale Deklaration einer Prozedur erfolgt mit

```
ProcName PROC [Distanz] [Sprache] [Sichtbarkeit]
   [<Prologargumente>]
   [USES  RegisterListe] [, Parameter[:Tag]] ...
   [LOCAL Variablenliste]
   Instruktionen
ProcName ENDP
```

PROC richtet je nach Art und Menge der angegebenen Argumente einen Stack-Rahmen ein und verwaltet ihn. Dies erfolgt im Rahmen eines sog. (Prozedur-) *Prologs* bzw. *Epilogs*. Der Prolog ist die Summe an allen Direktiven und Instruktionen, die erforderlich sind, um einen Stack-Rahmen einzurichten und mittels Symbolen einen einfachen Zugriff auf die ggf. übergebenen Parameter zu ermöglichen. Er wird durch den Assembler unmittelbar vor die erste Instruktion eingefügt. Der Epilog hingegen entfernt den Stack-Rahmen wieder anhand der durch den Prolog durchgeführten Aktionen und der verwendeten *Sprache* und wird unmittelbar vor dem die Prozedur abschließenden RET eingefügt.

ProcName ist der Name, unter dem die Prozedur deklariert wird und damit aufrufbar ist. Die Benennung folgt hierbei den Konventionen, die die als optionales Argument übergebene Sprache vorgibt.

Distanz gibt an, welche Zeigertypen der CALL-Befehl, mit dem Prozeduren aufgerufen werden, verwenden und mit welchem RET-Befehl, RET oder RETF die Prozedur abgeschlossen werden soll: 16-Bit-NEAR-CALLs (also nur ein 16-Bit-Offset), 16-Bit-FAR-CALLs (16-Bit-Selektor und 16-Bit Offset), 32-Bit-NEAR-CALLs (nur 32-Bit-Offset) oder 32-Bit-FAR-CALLs (16-Bit-Selektor und 32-Bit-Offset). Dementsprechend werden die Schlüsselworte NEAR16, FAR16, NEAR32 oder FAR32 als Distanz angegeben. NEAR spezifiziert einen NEAR-CALL gemäß der aktuellen Segmentgröße: Beträgt sie 16-Bit, entspricht dies NEAR16, ansonsten NEAR32. Analog spezifiziert FAR einen FAR-CALL entsprechend FAR16 bei 16-Bit- und FAR32- bei 32-Bit-Segmenten.

Sprache steuert die Konventionen, denen die Prozedur folgen soll. Erlaubte Argumente sind C, SYSCALL, STDCALL, BASIC, FORTRAN und PASCAL, bei TASM auch CPP (= C++) und NOLANGUAGE (= Assembler). Sie steuern die in Tabelle 3.5 dargestellten Konventionen.

	(keine)	C / C++	SYSCALL	STDCALL	BASIC	FORTRAN	PASCAL
führender Unterstrich vor Namen (»_«)		X		X			
alle Namen in Großbuchstaben					X	X	X
Parameterreihenfolge auf dem Stack	L→R	R→L	R→L	R→L	L→R	L→R	L→R
Stackbereinigung: **P**rozedur / **R**ufer	P	R	P	R / P	P	P	P
Sicherung des (E)BP-Registers					X	X	X
variable Parameterliste möglich		X	X	X			

R/P bedeutet, dass STDCALL den Rufer den Stack bereinigen lässt, wenn eine variable Parameterliste angegeben wird. Andernfalls überlässt STDCALL die Stackbereinigung der Prozedur. R→L bedeutet, dass die Parameter von rechts nach links gelesen auf den Stack gelegt werden, bei L→R werden sie in der üblichen Leserichtung auf den Stack gebracht.

Tabelle 3.5: Konventionen, nach denen Hochsprachen-Prozeduren eingerichtet werden

Sichtbarkeit steuert die »Sichtbarkeit« der Prozedur »nach draußen«, also zu anderen (Hochsprachen- oder Assembler-) Modulen. Es gibt drei erlaubte Argumente: PRIVATE, PUBLIC und EXPORT. Wird PRIVATE angegeben, ist die Prozedur nur im aktuellen Assemblermodul

sichtbar. Bei PUBLIC, der Standardeinstellung, ist das Modul nur im aktuellen Segment sichtbar, das jedoch aus mehreren Modulen bestehen kann. In allen Modulen dieses Segmentes kann die Prozedur gesehen werden. EXPORT schließlich ist PUBLIC in Verbindung mit FAR und sorgt dafür, dass der Linker die Adresse der Prozedur in die *export entry table* einträgt. Sie ist damit für jedes Modul im gesamten verfügbaren Adressraum sichtbar.

Prolog und Epilog sind Makros, die standardmäßig definiert sind. Sie sind für die Einrichtung und das Entfernen eines Stack-Rahmens und die Zuordnung der Adressen auf dem Stack übergebenen Parameter und ihre Zuordnung zu den in der Parameterliste genannten Symbolnamen zuständig. Diesen Makros kann man Argumente (*Prologargumente*) übergeben, die, eingeklammert in spitzen Klammern und durch Kommata getrennt, das Verhalten des Makros beeinflussen. Mögliche Argumente sind FORCEFRAME, was das Einrichten und Entfernen eines Stacks erzwingt, selbst wenn keiner erforderlich ist, und LOADDS, das den Inhalt des DS-Registers sichert und am Ende der Prozedur wieder restauriert.

Hinter *USES* steht eine Liste von Registern, die der Prozedur-Prolog auf den Stack retten soll und die vom Prozedur-Epilog wieder restauriert werden. Die einzelnen Register müssen mit Leerzeichen getrennt werden.

Die der Prozedur übergebenen *Parameter* werden durch Kommata getrennt. Wird das Schlüsselwort USES verwendet, muss die Parameterliste nach dem letzten Register folgen, durch ein Komma getrennt. Achtung! Parameter können nur dann angegeben werden, wenn bei der Definition des Segments oder über das Argument *Sprache* eine Sprache ausgewählt wurde!

Tag bezeichnet eine von zwei Möglichkeiten der Typangabe des dazugehörigen Parameters: Entweder folgt das Schlüsselwort VARARG, das dem Assembler sagt, dass unter dem Label Parameter eine variable Argumentenliste anzusprechen ist. VARARG muss immer das tag des letzten Parameters der Liste sein, mehrfache Verwendung ist ausgeschlossen! Oder es handelt sich um einen qualifizierten Typen (vgl. Seite 560).

LOCAL richtet die in der Variablenliste angegebenen Variablen lokal ein, was bedeutet, dass sie nicht global, sondern nur innerhalb der aktuellen Prozedur verfügbar sind und mit Beenden der Prozedur ihr Leben aushauchen! Siehe Seite 616.



Neben der MASM-Syntax gestattet TASM auch noch zwei weitere Möglichkeiten der Deklaration von Prozeduren, je eine für den MASM- und IDEAL-Modus.

MASM-Modus
```
ProcName PROC [[Modifizierer] Sprache] [Distanz]
   [ARG   Parameterliste] [RETRUNS Ergebnisliste]
   [LOCAL Variablenliste]
   [USES  Registerliste]
   Instruktionen
ProcName ENDP
```

IDEAL-Modus
```
PROC ProcName [[Modifizierer] Sprache] [Distanz]
   [ARG   Argumentenliste] [RETRUNS Ergebnisliste]
   [LOCAL Variablenliste]
   [USES  Registeriste]
   Instruktionen
ENDP ProcName
```

TASM führt, verglichen mit MASM, lediglich die Argumente *Modifizierer* und *RETURNS* ein. *Sprache*, *Distanz*, *USES* und *LOCAL* haben die gleiche Funktion wie in der MASM-Syntax, der Parameterliste wird lediglich das Schlüsselwort *ARG* (siehe Seite 617) vorangestellt. Somit muss diese Liste auch nicht zwangsläufig der USES-Anweisung folgen.

Modifizierer erlaubt die Angabe eines Schlüsselworts, das die Aktivitäten des Prolog- und Epilog-Makros noch erweitert und an bestimmte Situationen anpasst. Mögliche Schlüsselworte sind NORMAL, WINDOWS, ODDNEAR oder ODDFAR. Bei der Standardeinstellung NORMAL wird ein »normaler« Prolog und Epilog erzeugt. WINDOWS verändert beide so, dass die Prozedur auch aus Windows heraus aufgerufen werden kann – Windows benutzt einen speziellen stack frame. In diesem Fall muss jedoch die Distanz FAR sein, da Windows grundsätzlich nur als FAR deklarierte Prozeduren aufrufen kann. ODDNEAR und ODDFAR stellen die Kompatibilität mit dem *VROOM manager* (Borland Overlay Manager) her, der entweder im Betriebsmodus *oddnear* oder *oddfar* läuft.

RETURNS leitet eine Liste ein, die als Rückgabewerte auf dem stack verbleiben und nicht bereinigt werden.

PROTO Mit PROTO lassen sich analog C / C++ Prototypen von Prozeduren deklarieren, die eine Typprüfung der als Argumente übergebenen Parameter zu jedem Zeitpunkt ermöglichen. PROTO folgt der formalen Deklaration

Deklaration
```
Name PROTO [Distanz] [Sprache] [Parameter[:Tag] ,...]
```

Für Einzelheiten zu den Argumenten siehe PROC im vorangehenden Abschnitt.



Achtung! PROTO muss in jedem Fall der Prozedur-Deklaration vorangehen, also ggf. als Include-File eingeschlossen werden, falls die Prototypen-Deklaration in einem anderen Modul erfolgt. Grund: PROTO veröffentlicht den Namen der Prozedur! Das bedeutet, dass *Name* als EXTERNAL deklariert wird, wenn die eigentliche Prozedur-Deklaration nicht in dem Modul erfolgt, in dem die PROTO-Deklaration steht. In dem Modul, in dem die Prozedur deklariert wird, auf die mittels des externen Prototypen verwiesen wird, wird sie dann als PUBLIC veröffentlicht.



Ein Tipp für Delphi-Programmierer, die Prototypen von Funktionen nicht kennen (obwohl der INTERFACE-Teil einer Unit in Verbindung mit USES im Prinzip auch nichts anderes ist: USES binden den INTERFACE-Teil ein, in dem ja die Prozedur-Deklarationen, eben die Prototypen, stehen!): Prototypen machen Sinn, wenn in unterschiedlichen Modulen Gebrauch von der Deklaration einer Prozedur gemacht wird. So können z.B. analog den Header-Files in C / C++ Prototypen deklariert werden, die dann in einem Modul (z.B. einer Unit) mit der eigentlichen Prozedur-Deklaration verknüpft werden (via INCLUDE), in einem anderen per CALL-Befehl aufgerufen. Wird hier der Prototyp ebenfalls via INCLUDE eingebunden, ist sichergestellt, dass Rufer und gerufene Prozedur die gleichen Übergabekonventionen benutzen.

PROCDESC



Neben der Deklaration von Prototypen via PROTO gestattet TASM noch zwei weitere Möglichkeiten, je eine für den MASM- und IDEAL-Modus.

```
Name PROCDESC [Beschreibung]                    MASM-Modus
PROCDESC Name [Beschreibung]                    IDEAL-Modus
```

wobei *Beschreibung* wie folgt deklariert ist:

```
[[Modifizierer] Sprache] [Distanz] [Parameterliste]
```

Auch hier sind genauere Angaben zu den Argumenten im vor-vorangehenden Abschnitt beschrieben.



TASMs PROCDESC ist ein Synonym für PROTO. TASM kennt auch die Deklaration eines Prozedur-Typen mittels PROCTYPE (vgl. Seite 595).

LOCAL LOCAL veranlasst den Assembler, lokale Variablen zu erzeugen, die nur innerhalb der aktuellen Prozedur sichtbar und existent sind. Es sind Variablen, die auf dem Stack liegen und damit mit Entfernen des aktuellen Stack-Rahmens ebenfalls entfernt werden. Die Reservierung von Speicher für solche lokalen Variablen folgt der formalen Deklaration

Deklaration
```
LOCAL Argument [, Argument] ...
```

wobei Argument folgende Deklaration besitzt:

```
Name [[expression]] [:Qualifizierter Typ]
```

Name ist der Name, unter dem die lokale Variable angesprochen wird. Ihr Typ ist *Qualifizierter Typ*. Wird dieser Typ nicht angegeben, so verwendet der Assembler den Typ WORD in 16-Bit-Segmenten und DWORD in 32-Bit-Segmenten. Der Assembler ordnet ihr einen Speicherbereich der Größe *Qualifizierter Typ* auf dem Stack zu und übergibt ihr die Adresse des ersten Bytes, sodass der Programmierer sich um die Adressierung nicht weiter kümmern muss.

[expression], das in eckigen Klammern angegeben werden muss, wenn es verwendet wird, ist ein Ausdruck, der angibt, wie oft das Datum unter den Namen alloziert werden soll.

Beispiel
```
LOCAL OneByte:BYTE, [2] TwoDWords, AStruct:MyStructure
```

reserviert Platz für das Byte *OneByte* auf dem Stack, für zwei DWORDS (WORDS) unter dem Namen *TwoDWords*, wenn das Segment 32 (16) Bit groß ist – wobei TwoDWords auf das erste DWORD (WORD) zeigt! –, sowie für *AStruct* des selbst deklarierten Typs MyStructure.



Innerhalb von Makros hat LOCAL eine etwas andere Bedeutung! Hier generiert der Assembler interne Namen für Variable oder Labels, die mit LOCAL deklariert wurden, und ordnet diese dem Label oder der Variablen zu. Dies erfolgt für den Programmierer vollständig unsichtbar, er verwendet immer den unter LOCAL deklarierten Namen. Die automatisch generierten Namen unterscheiden sich voneinander, um sicherzustellen, dass bei wiederholtem Aufruf des Makros keine Konflikte aufgrund der bereits beim vorhergehenden Aufruf deklarierten Namen entstehen!



Wie soll es auch anders sein: Neben der MASM-Syntax gestattet TASM auch noch zwei Ergänzungen:

```
LOCAL Argument [, Argument] ... [= Symbol]

Name [[expression1]] [:Qualifizierter Typ[:expression2]]
```

Symbol addiert die Größe aller Argumente in der LOCAL-Liste und stellt somit die Gesamtgröße des mit lokalen Variablen belegten Speicherbereichs auf dem Stack zur Verfügung. Angenehm ist das, da *Symbol* innerhalb der RET-Instruktion als Anzahl vom Stack zu entfernender Bytes angegeben werden kann. Sinn macht das jedoch nur, wenn nicht die »automatischen« Prozedur-Deklarationen mit Angabe von Sprache, Parametern und lokalen Variablen verwendet wird, da in diesem Falle der Assembler die Stackbereinigung automatisch managt.

Expresion2 ist ein Ausdruck, der ebenfalls, wie *expression1*, die Anzahl der Wiederholungen der Allokation des Datums vom Typ *Qualifizierter Typ* unter dem Namen *Name* angibt: Somit ist die Gesamtzahl der Allokationen

```
expression1 · expression2
```

Wozu? Mit *expression2* kann die tatsächlich verwendete Größe des Datums eingestellt werden, mit *expression1* die Anzahl gleicher Daten. So ist der Defaultwert für *expression1* immer 1, für *expression2* auch, es sei denn, der Datentyp ist BYTE. Da die CPU keine einzelnen Bytes auf den Stack schieben kann, ist die Defaulteinstellung hier 2. Für Strings, beispielsweise, könnte sie z.B. auf 100 erhöht werden:

```
LOCAL [10] Strings:Byte:100 = Clear
```

In diesem Fall würden 10 Byte-Arrays der Größe 100 Byte unter dem Namen Strings lokal alloziert. *Clear* hätte dann den »Wert« *expr1 · expr2 · SIZE Typ*, hier also 10 · 100 · SIZE BYTE = 1000 (Bytes).

USES

USES ist eine Direktive, die im Rahmen der Deklarationen von Prozeduren eingesetzt wird (siehe PROC auf Seite 595). Sie dient dazu, den Inhalt von Registern auf den Stack zu retten, die im Rahmen einer Prozedur benutzt werden. Nötig wird dies bei Registern, die aufgrund von Konventionen in Hochsprachen eine bestimmte Bedeutung haben (DS, ES, (E)BX, (E)SI, (E)DI, etc.) und nicht verändert werden dürfen.

Mit USES wird nun innerhalb einer Prozedur im Rahmen des Prologs Code generiert, der die Inhalte der als Argumente angegebenen Regis-

ter auf den Stack rettet und im Rahmen des Epilogs von dort wieder restauriert. USES wird formal wie folgt verwendet:

Verwendung

```
USES Register [ Register] ...
```

wobei mehrere Register durch Leerzeichen getrennt werden müssen.

ARG



Auch ARG ist eine Direktive, die im Rahmen der Deklarationen von Prozeduren eingesetzt wird (siehe PROC auf Seite 595). Sie dient dazu, Symbole für die über den Stack übergebenen Parameter einer Prozedur zu erzeugen, die nur innerhalb der aktuellen Prozedur sichtbar und existent sind und mit denen ein einfacher Zugriff auf die Variablen möglich ist. Wie lokale Variablen liegen auch die Parameter auf dem Stack und werden damit mit dem Entfernen des aktuellen Stack-Rahmens ggf. entfernt. Die Reservierung von Speicher ist für die Parameter nicht erforderlich, da dies die die Prozedur rufende Routine erledigt. Daher dreht es sich hier, anders als bei LOCAL, nur um die Deklaration der Symbole. Sie folgt allerdings der formalen Deklaration der lokalen Variablen:

Deklaration

```
ARG Argument [, Argument] ...
```

wobei Argument folgende Deklaration besitzt:

```
Name [[expression]] [:Qualifizierter Typ]
```

Zu Einzelheiten siehe LOCAL im vorangehenden Abschnitt.

INVOKE



INVOKE ist das Gegenstück zu PROC auf der rufenden Seite. Während PROC die Übergabe von Parametern an die Prozedur über den Stack vereinfacht, vereinfacht INVOKE das Vorbereiten des Stack zu einem Prozeduraufruf inklusive dem Aufruf selbst. Nach Rückkehr aus der Prozedur in den rufenden Teil wird eventuell auch der Stack bereinigt, wenn das nicht schon im Rahmen der Prozedur erfolgte. Mit INVOKE und CALL ist das Aufrufen von Prozeduren genauso einfach wie in Hochsprachen!

Deklaration

Die formale Deklaration von Nutzung von INVOKE erfolgt nach:

```
INVOKE expression [Argumentenliste]
```

Expression ist ein Ausdruck, der eine Adresse oder das Label der Prozedur (oder ihres Prototypen) ergeben muss, die gerufen werden soll. *Expression* muss eine gültige Adresse liefern und kann keine Vorwärts-Referenzen beinhalten.

Argumentenliste ist die Liste der Argumente, die der Prozedur übergeben wird. Die Reihenfolge der Übergabe und somit die Anordnung auf dem Stack entnimmt INVOKE der Deklaration des Prototypen oder der Prozedur und deren Argumenten (z.B., falls angegeben, *Sprache*). Auf diese Weise kann INVOKE auch feststellen, ob nach der Rückkehr aus der Prozedur der Stack zu bereinigen ist (z.B. bei C/C++) oder nicht (z.B. bei Delphi resp. Pascal).

Das TASM-Gegenstück zu INVOKE ist CALL (siehe folgender Abschnitt). Bitte beachten Sie, dass INVOKE und CALL nicht kompatibel sind, weshalb hier auch durch die Wortwahl eine Inkompatibilität des Assemblerquelltextes erzwungen wird.



CALL

CALL ist die TASM-Version von INVOKE und ist damit das Gegenstück zu PROC auf der rufenden Seite des Moduls. CALL vereinfacht den Aufruf von Prozeduren, indem der Direktive die Parameter übergeben werden, die PROC erwartet. CALL legt diese gemäß der Definition von PROC auf den Stack, bevor die Prozedur aufgerufen wird. Nach Rückkehr aus der Prozedur in den rufenden Teil wird eventuell auch der Stack bereinigt, wenn das nicht schon im Rahmen der Prozedur erfolgte. Mit CALL ist das Aufrufen von Prozeduren genauso einfach wie in Hochsprachen!



Verwechseln Sie bitte nicht die *Direktive* CALL mit der *Instruktion* CALL! Bei der *Instruktion* (»Befehl«) CALL handelt es sich um einen CPU-Befehl, der unabhängig vom Assembler ist und eine bestimmte, genau definierte und in Teil 1 dargestellte Aktion der CPU zur Folge hat! Bei der *Direktiven* (»Anweisung«) CALL handelt es sich um eine Assembler-spezifische Anweisung an den Assembler, bestimmte Code-Sequenzen anhand der als Argumente übergebenen Randbedingungen zu erzeugen, die den Stack beladen, eine CALL-Instruktion durchführen und den Stack ggf. wieder bereinigen.



Die formale Deklaration von Nutzung von CALL erfolgt nach: *Deklaration*

```
CALL expression [Sprache] [Argumentenliste]
```

Verglichen mit INVOKE ist bei TASMs CALL noch optional die Angabe der Sprache möglich, nach deren Aufrufkonvention der Stack präpariert wird.

Expression ist ein Ausdruck, der eine Adresse oder das Label der Prozedur (oder ihres Prototypen) ergeben muss, die gerufen werden soll. *Expression* muss eine gültige Adresse liefern und kann keine Vorwärts-Referenzen beinhalten.

Argumentenliste ist die Liste der Argumente, die der Prozedur übergeben wird. Die Reihenfolge der Übergabe und somit die Anordnung auf dem Stack entnimmt CALL der Deklaration des Prototypen oder der Prozedur und deren Argumenten (z.B., falls angegeben, *Sprache*). Auf diese Weise kann CALL auch feststellen, ob nach der Rückkehr aus der Prozedur der Stack zu bereinigen ist (z.B. bei C/C++) oder nicht (z.B. bei Delphi resp. Pascal).

Wird *Sprache* angegeben, so erfolgt die Übergabe der Parameter gemäß dieser Sprachkonvention. Es gelten die in Tabelle 3.5 auf Seite 612 genannten Regeln.



CALL, genauer eine Syntax-Erweiterung von CALL: *CALL ... METHOD*, wird ebenfalls verwendet, um Methoden im Rahmen der Objektorientierten Programmierung (OOP) aufzurufen. Bitte haben Sie Verständnis dafür, wenn ich im Rahmen dieses Buches nicht auf diese Möglichkeiten eingehe.



Unter TASM kann als Sprache auch NOLANGUAGE angegeben werden. In diesem Fall darf CALL kein Argument übergeben werden, da der Assembler sonst eine Fehlermeldung generiert. Nachdem Sie aber PROC auch unter NOLANGUAGE mit Argumenten versehen können, müssen Sie manuell den Stack beladen, wenn Sie CALL unter NOLANGUAGE mit Argumenten nutzen wollen.



TASM erlaubt bei der Deklaration von Prozeduren mittels PROC oder PROTO (= PROCDESC) die Angabe des Schlüsselwortes RETURNS. Die Argumente hinter diesem Schlüsselwort sind Parameter, die ein oder mehrere Ergebnisse der Prozedur (»Funktionsergebnis«) aufnehmen sollen. Die Einrichtung dieser Parameter auf dem Stack sowie ihre Bereinigung nach der Rückkehr aus der Prozedur obliegt in jedem Fall dem rufenden Programmteil, weder PROC/PROTO noch CALL richten sie ein oder entfernen sie! PROC deklariert lediglich lokale Symbole, die die Adressen der entsprechenden Parameter auf dem Stack aufnehmen, und PROTO führt lediglich eine formale Typprüfung durch.

Das MASM-Gegenstück zu CALL ist INVOKE (siehe vorangehender Abschnitt). Bitte beachten Sie, dass INVOKE und CALL nicht kompatibel sind, weshalb hier auch durch die Wortwahl eine Inkompatibilität des Assemblerquelltextes erzwungen wird.



JMP

Analog zu CALL gibt es mit JMP ebenfalls eine TASM-spezifische *Direktive* (!), mit der Prozeduren, die mittels PROC deklariert wurden, bequem angesprungen werden können. Der Unterschied zwischen CALL und JMP besteht darin, dass CALL eine Rücksprungadresse auf den Stack legt, während JMP dies nicht tut. Für weitere Einzelheiten siehe CALL im vorangehenden Abschnitt.



3.2.6 Direktiven zu Scope und Sichtbarkeit

PUBLIC

PUBLIC gestattet es, Labels, Prozeduren, Daten oder Symbole allgemein auch außerhalb des aktuellen Moduls sichtbar zu machen. In der Regel sind nämlich alle diese Elemente nur innerhalb des aktuellen Moduls bekannt. Sollen andere Module auf solche Elemente zugreifen können (z.B. Aufruf einer Prozedur oder Zugriff auf ein Datum), müssen sie durch PUBLIC »veröffentlicht« werden. Die formale Verwendung von PUBLIC erfolgt nach

Verwendung

```
PUBLIC [Sprache] Name [, [Sprache] Name ...]
```

Mit *Sprache* kann gesteuert werden, mit welcher Schreibweise die public gemachten Elemente veröffentlicht werden. Dies ist vor allem dann wichtig, wenn sie von Modulen genutzt werden sollen, die mit *Sprache* erstellt worden sind. Als Argument sind möglich: C, SYSCALL, STDCALL, BASIC, FORTRAN und PASCAL, bei TASM auch CPP (= C++) und NOLANGUAGE (= Assembler). Vergleiche hierzu auch Tabelle 3.5 und die darin dargestellten Konventionen.

Name ist der Name des zu veröffentlichenden (= zu »exportierenden«) Elements (Label, Prozedur, Datum oder mittels EQU deklariertes Symbol).

EXTERN
EXTRN

EXTERN ist das Gegenstück zu PUBLIC. Soll in einem Modul ein Element aus einem anderen Modul verwendet werden, muss dem Assembler dieses bekannt gemacht werden, bevor es verwendet werden kann. Dies kann nur mit veröffentlichten Elementen erfolgen, also solchen, die im betreffenden Modul als PUBLIC deklariert wurden.

Verwendung EXTRN ist ein Synonym zu EXTERN. Beide werden formal wie folgt verwendet:

```
EXTERN Definition [, Definition ...]
```

wobei *Definition* wie folgt realisiert ist:

```
[Sprache] Name:Qualifizierter Typ
```

Mit *Sprache* kann angegeben werden, in welcher Schreibweise die externen Elemente vorliegen. Dies ist vor allem dann wichtig, wenn sie in Modulen deklariert wurden, die mit *Sprache* erstellt worden sind und somit in der durch Sprache definierten Form veröffentlicht wurden. Als Argument sind möglich: C, SYSCALL, STDCALL, BASIC, FORTRAN und PASCAL, bei TASM auch CPP (= C++) und NOLANGUAGE (= Assembler). Vergleiche hierzu auch Tabelle 3.5 und die darin dargestellten Konventionen.

Name ist der Name des zu »importierenden« Elements (Label, Prozedur, Datum oder Symbol).

Qualifizierter Typ ist ein Qualifizierter Typ nach der Definition auf Seite 560.



MASM erlaubt noch eine Erweiterung der *Definition*:

```
[Sprache] Name[(Alias)]:Qualifizierter Typ
```

Mit *Alias* kann ein Name angegeben werden, der den allgemein gültigen Konventionen für die Deklaration von Labels genügt und anstelle des Symbols *Name* verwendet werden kann.



Auch TASM erlaubt spezifische Anpassungen von *Definition*:

```
[Sprache] Name[[Anzahl]]:Qualifizierter Typ:[Anzahl2]
```

[Anzahl1] und *Anzahl2* sind zwei Zähler, die angeben, wie Name zu interpretieren ist. *Anzahl1* ist ein Feldelementezähler und gibt an, wie viele Elemente des Typs *Qualifizierter Typ* zu Name gehören. Defaultwert ist 1. *Anzahl2* dagegen gibt an, wie viele solcher »Felder« unter dem Symbol *Name* angesprochen werden. Der durch *Name* belegte Speicherplatz berechnet sich somit zu *Anzahl1* · *Anzahl2* · *Größe(Qualifizierten Typ)*.

EXTERNDEF Die Direktive EXTERNDEF ist eine Mischung aus EXTERN und PUBLIC: In Modulen, in denen ein Symbol mit EXTERNDEF öffentlich gemacht wird, hat es die Funktion von PUBLIC (vgl. Seite 621), in Mo-

dulen, in denen es verwendet werden soll, die Funktion von EXTERN (vgl. Seite 621).

Verwendung

EXTERNDEF wird wie EXTERN formal wie folgt verwendet:

```
EXTERNDEF Definition [, Definition ...]
```

wobei *Definition* wie folgt realisiert ist:

```
[Sprache] Name:Qualifizierter Typ
```

Mit *Sprache* kann angegeben werden, in welcher Schreibweise die externen Elemente vorliegen. Dies ist vor allem dann wichtig, wenn sie in Modulen deklariert wurden, die mit *Sprache* erstellt worden sind, und somit in der durch Sprache definierten Form veröffentlicht wurden. Als Argument sind möglich: C, SYSCALL, STDCALL, BASIC, FORTRAN und PASCAL, bei TASM auch CPP (= C++) und NOLANGUAGE (= Assembler). Vergleiche hierzu auch Tabelle 3.5 und die darin dargestellten Konventionen.

Name ist der Name des zu »importierenden« Elements (Label, Prozedur, Datum oder Symbol).

Qualifizierter Typ ist ein Qualifizierter Typ nach der Definition auf Seite 560.



EXTERNDEF wird gerne in inclusion files eingesetzt, die ähnlich wie die Header-Dateien in C/C++ von allen Modulen eingebunden werden, die bestimmte Symbole nutzen sollen. In solchen inclusion files stehen dann die Prototypen von Prozeduren und mittels EXTERNDEF veröffentlichten Labels, Variablen und Symbole, die in einem Modul deklariert werden, in anderen benutzt. Auf diese Weise ist sichergestellt, dass in allen Modulen die Symbole gleich deklariert sind.

GLOBAL



Die Direktive GLOBAL ist wie EXTERNDEF eine Mischung aus EXTERN und PUBLIC: In Modulen, in denen ein als GLOBAL markiertes Symbol deklariert wird, hat es die Funktion von PUBLIC (vgl. Seite 621), in Modulen, in denen es verwendet werden soll, die Funktion von EXTERN (vgl. Seite 621). Im Unterschied zu EXTERNDEF kann GLOBAL auch im IDEAL-Modus von TASM eingesetzt werden und verfügt über eine leicht abgewandelte Syntax. Von der Funktion her ist es jedoch mit EXTERNDEF identisch.

Verwendung GLOBAL wird wie die TASM-Version von EXTERN und PUBLIC im MASM- und IDEAL-Modus verwendet:

```
[Sprache] Name[[Anzahl]]:Qualifizierter Typ:[Anzahl2]
```

[Anzahl1] und *Anzahl2* sind zwei Zähler, die angeben, wie Name zu interpretieren ist. *Anzahl1* ist ein Feldelementezähler und gibt an, wie viele Elemente des Typs *Qualifizierter Typ* zu Name gehören. Defaultwert ist 1. *Anzahl2* dagegen gibt an, wie viele solcher »Felder« unter dem Symbol *Name* angesprochen werden. Der durch *Name* belegte Speicherplatz berechnet sich somit zu *Anzahl1* · *Anzahl2* · *Größe(Qualifizierten Typ)*.

LOCALS
NOLOCALS



LOCALS gestattet die Nutzung lokaler Symbole, geht also über die Nutzung lokaler Labels hinaus. Mit LOCALS können zwei Dinge bewerkstelligt werden:

- Freigabe der Nutzung lokaler Symbole (vgl. Seite 602) und
- Definition der Präfixe für lokale Symbole.

NOLOCALS sperrt die Nutzung lokaler Symbole.

Verwendung Formal werden LOCALS und NOLOCALS wie folgt verwendet:

```
LOCALS [Präfix]
NOLOCALS
```

Wird *Präfix* angegeben, so definiert es das Zeichen, das in doppelter Form ein lokales Symbol einleitet. Default ist `LOCALS @`, sodass alle lokalen Symbole mit »@@« beginnen.



Bitte verwechseln Sie nicht die Direktive LOCALS mit der Direktive LOCAL (ohne »S«), die im Rahmen der Deklaration von Prozeduren zum Einsatz kommt (vgl. Seite 616) und dort dafür zuständig ist, lokale Variablen zu allozieren.

COMM Die Direktive COMM erzeugt »kommunale« Variablen. Hierunter werden Variable verstanden, die uninitialisiert und veröffentlicht (»PUBLIC«, vgl. Seite 621) sind, deren Adresse jedoch nicht vom Assembler anhand der Deklaration im Quelltext vergeben wird, sondern vom Linker, also nach der Erstellung des OBJ-Files.

Das bedeutet: Der Assembler hat keinerlei Einfluss auf die Reihenfolge, die Position oder die Initialwerte von kommunalen Variablen. Somit gibt es keinerlei Gewähr, dass sie in der Weise im Speicher angesiedelt sind, in der sie im Assemblermodul deklariert wurden.



Deklaration

Formal erfolgt die Deklaration von kommunalen Variablen nach

```
COMM Definition [, Definition]
```

wobei Definition wie folgt definiert ist:

```
[Sprache] [Distanz] Name: Qualifizierter Typ [:Anzahl]
```

Mit *Sprache* kann angegeben werden, in welcher Schreibweise die externen Elemente vorliegen. Dies ist vor allem dann wichtig, wenn sie in Modulen deklariert wurden, die mit *Sprache* erstellt worden sind und somit in der durch Sprache definierten Form veröffentlicht wurden. Als Argument sind möglich: C, SYSCALL, STDCALL, BASIC, FORTRAN und PASCAL, bei TASM auch CPP (= C++) und NOLANGUAGE (= Assembler). Vergleiche hierzu auch Tabelle 3.5 und die darin dargestellten Konventionen.

Distanz kann die Werte FAR oder NEAR annehmen. Wird dieser optionale Parameter nicht angegeben, wird die Distanz aus der MODEL-Deklaration verwendet. *Name* ist der Name des kommunalen Symbols und Anzahl die Zahl der Elemente, die unter *Name* alloziert werden sollen.

Qualifizierter Typ ist ein Qualifizierter Typ nach der Definition auf Seite 560.

TASM erlaubt aufgrund seines IDEAL-Modus auch eine spezifische Anpassung von *Definition*:



```
[Distanz] [Sprache] Name[[Anzahl]]:Qualif.Typ:[Anzahl2]
```

[Anzahl1] und *Anzahl2* sind zwei Zähler, die angeben, wie Name zu interpretieren ist. *Anzahl1* ist ein Feldelementezähler und gibt an, wie viele Elemente des Typs *Qualifizierter Typ* zu Name gehören. Defaultwert ist 1. *Anzahl2* dagegen gibt an, wie viele solcher »Felder« unter dem Symbol *Name* angesprochen werden. Der durch *Name* belegte Speicherplatz berechnet sich somit zu *Anzahl1* · *Anzahl2* · *Größe(Qualifizierten Typ)*.

INCLUDE INCLUDE ist eine Direktive, die den Assembler veranlasst, beim Assemblieren an der Stelle ihres Auftretens den Inhalt des als Argument übergebenen Files in die Assemblierung einzubinden. Dieser File muss somit ein Textfile sein, der gültige Assembler-Direktiven und Instruktionen beinhaltet. INCLUDE wird formal wie folgt verwendet:

Verwendung
```
INCLUDE filename,
INCLUDE <filename> oder
INCLUDE "filename"
```

Die zweite Version muss unter MASM verwendet werden, wenn *filename* folgende Zeichen enthält: »\«, »;«, »>«, »<«, »'« oder »"«. Im IDEAL-Modus von TASM muss *filename* in Anführungszeichen gesetzt werden.



Sie können INCLUDE beliebig tief schachteln. Das bedeutet: Ein Modul kann mittels INCLUDE ein Modul einbinden, das eine INCLUDE-Anweisung enthält, die ein Modul einbindet, das eine INCLUDE-Anweisung enthält, die ein Modul einbindet, das ...

INCLUDELIB INCLUDELIB informiert den Linker, dass das aktuelle Modul mit einem Modul zusammengebunden (»gelinkt«) werden soll. Die formale Verwendung ist

Verwendung
```
INCLUDELIB libraryname,
INCLUDELIB <libraryname> oder
INCLUDELIB "libraryname"
```

Die zweite Version muss unter MASM verwendet werden, wenn *filename* folgende Zeichen enthält: »\«, »;«, »>«, »<«, »'« oder »"«. Im IDEAL-Modus von TASM muss *filename* in Anführungszeichen gesetzt werden.



Die mit INCLUDELIB eingebundene Datei muss im LIB-Format vorliegen, einer speziellen Form eines OBJ-Files. Mit INCLUDELIB können keine Quelltexte eingebunden werden. Hierzu dient INCLUDE (siehe vorangehenden Abschnitt).

PUBLICDLL



PUBLIC und EXTERN sind lediglich Direktiven, die in einzelnen Modulen deklarierte Symbole auch für andere zugänglich machen. Da das Zusammenführen von Modulen Aufgabe des Linkers ist, muss der Assembler dem Linker die Adressen »öffentlicher« Symbole (PUBLIC) übergeben, die dieser dann verwendet, um nicht aufgelöste Bezüge

(fix-ups) in anderen Modulen aufzulösen. Hat der Linker einmal seine Aufgabe erledigt, benötigt niemand mehr diese Adressen. »Öffentlich« heißt hier also: sichtbar außerhalb der jeweiligen Quell-Module, aber innerhalb des erzeugten Moduls, beispielsweise einer EXE-Datei.

So richtig öffentlich aber wird das Ganze erst, wenn auf solche gelinkten Module tatsächlich »von außen«, also z.B. von Fremdprogrammen oder vom Betriebssystem zugegriffen werden soll. Das ist in DLLs beispielsweise der Fall. Dann muss es eine Liste von »Einsprungpunkten« in diese DLL geben, die anderen Programmen zugänglich ist. TASM unterstützt dies, indem es die Direktive PUBLICDLL zur Verfügung stellt. Sie wird ganz analog zu PUBLIC (vgl. Seite 621) verwendet. Solchermaßen deklarierte Symbole werden im OBJ-File als *dynamic link entry point* eingetragen und entsprechend weiterverwendet.

3.2.7 Vollständige Segmentkontrolle

In Teil 1 dieses Buches haben wir uns bereits ausgiebig mit Segmenten beschäftigt. Als Hochsprachenprogrammierer haben Sie vermutlich bislang wenig mit solchen Segmenten zu tun gehabt, da die Hochsprachencompiler Ihnen die Definition und Verwaltung von Segmenten vollständig abnehmen. Hierzu folgen sie bestimmten Konventionen, die die Compilerbauer irgendwann einmal ins Leben gerufen haben.

Diese Situation ist sehr vorteilhaft, da Sie sich mit dem ganzen Bürokratismus, der eben auch erforderlich ist, wenn Programme oder Programm-Module geschrieben werden, nicht auseinander setzen müssen. Und: Keine Angst, diesen Komfort können Sie in und mit Assembler (in den meisten Fällen) auch nutzen, wie wir im nächsten Kapitel sehen werden.

Dennoch ist es manchmal erforderlich, von solchen Konventionen und vereinfachten Programm-Modellen abrücken und tatsächlich selbst Hand an einzelne Problemkreise anlegen zu können. Und dies ermöglicht Ihnen die Programmierung mit Assembler. Die in diesem Kapitel geschilderten Direktiven gestatten daher eine direkte Einflussnahme auf die Erstellung, Deklaration und Verwaltung von Segmenten.

SEGMENT
ENDS

In Kapitel »Segmenttypen, Gates und ihre Deskriptoren« auf Seite 407 haben wir Segmente detailliert besprochen. Wir haben dort eine Reihe von Feldern kennen gelernt, die Sie vermutlich aus Ihrer täglichen Programmierpraxis mit Hochsprachencompilern heraus nicht kannten. In dem entsprechenden Kapitel habe ich keine Angaben gemacht, wer welche Informationen und wann in diese Deskriptoren einträgt.

Hier die Lösung, zumindest teilweise. Es ist wohl klar, dass solche Felder wie die Flags busy, accessed und available vom Betriebssystem verwaltet werden und vom Programmierer nicht verändert werden können und sollen. Und auch die Lage von Segmenten im Speicher und damit der Eintrag im Feld *base address* ist eine Sache des Betriebssystems. Aber bereits die Segmentgröße (*segment limit*) ist durch Sie in gewissen Grenzen steuerbar. Und was einige Flags betrifft, wie z.B. das D- oder B-Flag oder auch Flags im Type-Feld, so werden Sie gleich staunen. Und selbst base address ist für Sie nicht grundsätzlich tabu ...

Deklaration

Segmente müssen, wie praktisch alles, mit dem der Assembler umgeht, deklariert werden. Die Deklaration eines Segments erfolgt formal und allgemein nach:

```
Name SEGMENT [Attribute]
   Statements
Name ENDS
```



TASM erlaubt aufgrund seines IDEAL-Modus auch die übliche »Verdrehung« der Reihenfolge der Angaben:

```
SEGMENT Name [Attribute]
   Statements
ENDS [Name]
```

Name ist der Name des Segments. Wenn Sie das aktuelle Assemblermodul im Rahmen von größeren Projekten benutzen, sollten Sie *Name* gemäß der Nutzung des Segments vorsichtig auswählen.

Als *Statements* werden alle Assembler-Direktiven und Instruktionen bezeichnet, die sich auf das aktuelle Segment beziehen sollen.

Die *Attribute* fallen in fünf Themengebiete:

1. *combine*: Attribute für die Kombination von Segmenten zu »einem« Segment
2. *class*: Attribute zur Segmentklasse
3. *align*: Attribute zur Segmentausrichtung
4. *size*: Attribute zur Segmentgröße
5. *access*: Attribute zum Segmentzugriff

Segmentkombination

Die Attribute zur Segmentkombination (»*combine*«) sind Informationen für den Linker. Sie steuern, wie dieser verschiedene Segmente mit gleichem Namen *Name* aus verschiedenen Modulen zu einem Segment kombinieren sollen. Hintergrund: Es ist durchaus guter Programmier-

stil, unterschiedliche Programmteile in jeweils eigenen Modulen zu realisieren – das erhöht u.U. die Lesbarkeit erheblich. Delphi geht z.B. mit seinem Unit-Konzept diesen Weg. Dennoch sollte anschließend jedes Segment, das Code enthält, auch im Codesegment wiederzufinden sein, ebenso wie die Daten praktischerweise in *einem* Datensegment aufbewahrt werden: Das spart Aufwand für das Hin- und Herschaufeln von Segment-Selektoren in den Segmentregistern samt Overhead für die Schutzkonzepte. Trotzdem muss pro Modul ein Code- und Datensegment deklariert werden, wenn Sie mit Assembler programmieren wollen. Daher ist es mehr als wahrscheinlich, dass Sie Module kombinieren müssen/wollen. Zur Steuerung gibt es nun folgende Schlüsselworte:

PRIVATEein *private segment* ist das, was das Attribut bereits ausdrückt: privat. Daher werden als PRIVATE deklarierte Segmente nicht mit irgendeinem anderen Segment kombiniert. Dies ist die Defaulteinstellung.

PUBLIC als PUBLIC werden Segmente deklariert, die durchaus mit anderen Segmenten gleichen Namens kombiniert werden können und sollen. Sie bilden im OBJ-File ein einziges, durchgehendes und ununterbrochenes Segment mit Namen *Name*. Die Offsets der einzelnen Labels, Prozeduren oder Daten in den Modulen solcher öffentlichen Segmente werden durch den Linker angepasst.

MEMORY Im Prinzip ist MEMORY nichts anderes als eine PUBLIC-Deklaration. Sie unterscheidet sich nur in einem kleinen, aber wesentlichen Detail: Sie veranlasst den Linker, einen Initialwert für den Stackpointer (E)SP und das Stacksegment (SS) in den EXE-File zu kopieren. Der Initialwert für den Stackpointer ist der Offset des Segment-Endes (= *segment limit*!), also die größte Adresse, die in dem betreffenden Segment möglich ist, der für SS die Basisadresse des Segments. (Vgl. hierzu auch »Stack« auf Seite 385 und »Stacksegmente« auf Seite 415.)

STACK Synonym zu MEMORY

COMMON Das Attribut COMMON veranlasst den Linker, alle Segmente gleichen Namens an die gleiche Adresse zu setzen. Sie belegen somit alle den gleichen Platz in einem gemeinsamen Segment, das so groß ist wie das größte Einzel-Segment, das mittels COMMON gelinkt wird.

AT xxxx — Das Schlüsselwort AT in Verbindung mit einer physikalischen Adresse zwingt den Linker, das entsprechende Segment an dieser Adresse anzusiedeln. Sinnvoll ist dieses Vorgehen im Zeitalter von Multitasking-Systemen nur in seltenen Fällen, z.B. um einen ROM-Bereich ansprechen zu können, der an festen Adressen liegt. Zu Zeiten des guten, alten DOS war auch ein beliebtes Anwendungsgebiet die Ansiedlung von Segmenten im Video-Bereich $B_0000 oder $C_0000, um direkt in den Videospeicher schreiben zu können.

Segmentklasse Unter Attribut für die Segmentklasse (»*class*«) versteht man einen String in Anführungszeichen, der dem Linker bei der Anordnung von Segmenten im Speicher hilft. So werden alle Segmente mit gleicher Segmentklasse zusammen gruppiert. ACHTUNG: Dies ist nicht das Gleiche wie *combine*! Während bei der Segmentkombination aus mehreren Segmenten eines gebildet wird, bleiben alle Segmente einer Segmentklasse unverbunden (so sie nicht mit *combine* kombiniert werden!), werden aber hintereinander und vor Segmenten anderer Klassen im Speicher angeordnet. So wird z.B. häufig die Klasse 'CODE' verwendet, um alle Segmente, die Code enthalten, vor alle Daten- und Stacksegmente zu platzieren, während 'DATA' einen Speicherbereich kennzeichnet, in dem Segmente vorzufinden sind, die irgendetwas mit Daten zu tun haben.

Segmentausrichtung Mit Hilfe des Segmentausrichtungs-Attributes (»*align*«) können Segmente im Speicher an bestimmten Bereichsgrenzen »ausgerichtet« werden. Dies bedeutet, dass die Basisadresse des Segments an Adressen liegt, die ohne Restbildung durch ganzzahlige Potenzen von 2 teilbar sind. Hierbei haben sich einige 2er-Potenzen einen prominenten Platz erobert: Erlaubte Schlüsselworte sind BYTE (2^0), WORD (2^1), DWORD (2^2), PARA (2^4), PAGE (2^8) und MEMPAGE (2^{12}). Defaultwert ist PARA.

Segmentgröße Durch das Segmentgrößen-Attribut (»*size*«) wird dem Linker mitgeteilt, in welcher Umgebung das Segment angesiedelt werden soll: in 16-Bit-Umgebungen mit 16-Bit-Adressen oder in 32-Bit-Umgebungen mit 32-Bit-Adressen. Dementsprechend gibt es zwei Schlüsselworte: USE16 und USE32. Damit sind die Maximalwerte für das Feld *segment limit* der Segmente vorgegeben: USE16 erlaubt max. 64-KByte-Segmente, USE32 4-GByte-Segmente. Hinweis: MASM erlaubt auch das Schlüsselwort FLAT.

Segmentzugriff

Mit den Attributen für den Segmentzugriff (»*access*«) kann festgelegt werden, welcher Art der Zugriff auf ein Segment sein darf. Erlaubte Schlüsselworte sind EXECONLY, EXECREAD, READONLY und READWRITE. Ich glaube, hierzu gibt es nichts weiter zu sagen, die Bedeutung ergibt sich aus den ab Seite 407 geschilderten Informationen zu den Segmentattributen.



Dieses Feature der Steuerung des Segmentzugriffs unterstützt derzeit nur der Linker von Phar Lab, nicht aber die mit MASM und/oder TASM ausgelieferten Linker LINK und TLINK! Somit ist die Nutzung der Schlüsselworte für den Segmentzugriff limitiert.



Ich hoffe, dass Sie in diesem Kapitel gesehen haben, dass mit Assembler eine Reihe von Einstellungen an Segmenten vorgenommen werden können, die in Kapitel 2 so sehr theoretisch ausgesehen haben und die Sie mit Hochsprachen nicht vornehmen können. Wenn Sie sich an dieser Stelle das Speicherabbild eines Segment-Deskriptors ins Gedächtnis rufen (oder schnell auf Seite 396 nachschauen), so werden Sie feststellen, dass viele Einträge in diesem Deskriptor mit den Attributen, die Sie über die Segmentdefinition unter Assembler vornehmen können, beeinflussbar sind:

Segment-Deskriptor-Feld	**Segment-Attribut in Assembler**
base address	combine (AT xxx, COMMON), align
segment limit	combine
granularity	size
D-Flag, B-Flag	size
S-Flag	class
Type	class, access

Konventionen

Aufgrund verschiedener Konventionen, die Hochsprachen nun einmal beinhalten, haben sich für die verschiedenen Attribute bestimmte Standardangaben eingebürgert. Sie sind in Tabelle 3.6 aufgelistet. Es ist guter Programmierstil, wenn man sich von diesen Konventionen nur dann entfernt, wenn man dies für absolut notwendig erachtet.

Segment	Segment-Attribute				
	Name	**Align ***	**Combine**	**Class**	**Group**
Code	_TEXT oder *Name*_TEXT	(D)WORD	PUBLIC	'CODE'	
Konstanten	CONST	(D)WORD	PUBLIC	'CONST'	DGROUP
Daten	DATA	(D)WORD	PUBLIC	'DATA	DGROUP
uninitialisierte Daten	BSS	(D)WORD	PUBLIC	'BSS'	DGROUP
Stack	STACK	PARA	PUBLIC	'STACK'	DGROUP
weitere Daten	FAR_DATA	PARA	PRIVATE	'FAR_DATA'	
weitere uninitial. Daten	FAR_BSS	PARA	PRIVATE	'FAR_BSS'	

* Wenn die Direktiven .8086, .186 oder .286 angegeben werden oder der entsprechende Befehlssatz aktiv ist, wird das Segment an WORD-Grenzen ausgerichtet, andernfalls an DWORD-Grenzen.

Tabelle 3.6: Nach Konventionen üblicherweise benutzte Werte für bestimmte Segmentattribute

ENDS Die Deklaration eines Segments wird abgeschlossen durch die Direktive ENDS. ENDS bewirkt, dass alle nachfolgenden Direktiven und oder Instruktionen nicht mehr das geschlossene Segment betreffen und somit zu Fehlermeldungen führen werden, falls nicht ein anderes Segment noch offen ist oder inzwischen geöffnet wurde. SEGMENT und ENDS klammern einen segmentrelevanten Bereich im Quellcode wie ein BEGIN-END in Delphi oder die geschweiften Klammern { } in C/C++.

GROUP Die Direktive GROUP dient dazu, verschiedene Segmente in einer Gruppe zu gruppieren. Adressen von Daten, die in solchen Gruppen residieren, beziehen sich anschließend nicht mehr auf den Beginn des Segmentes, in dem sie deklariert wurden, sondern auf den Beginn des Segmentes der Gruppe.

Eine häufige Anwendung der Möglichkeit zur Gruppierung ist die Zusammenführung der unterschiedlichen Datensegmente, die ein Modul haben kann (Konstanten, initialisierte und uninitialisierte Daten und Stack), zu einem einzigen »Datensegment«.



Alle Segmente, die in einer Gruppe zusammengefasst werden sollen, müssen über die gleiche »Größe« verfügen. Sie können nicht ein Segment, das 16-Bit-Adressen nutzt, mit Segmenten gruppieren, die 32-Bit-

Adressen nutzen. In 16-Bit-Umgebungen können somit Gruppen nicht größer als 64 KByte werden, in 32-Bit-Umgebungen nicht größer als die magischen 4 GByte.

Verwendung

Die GROUP-Direktive wird formal wie folgt verwendet:

```
Name GROUP SegmentName [, SegmentName ...]
```

wobei *Name* der Name der Gruppe ist und *SegmentName* der Name des aufzunehmenden Segments, das mit SEGMENT deklariert wurde. Falls die Gruppe *Name* bereits existiert, wird *SegmentName* durch diese Verwendung in die Gruppe aufgenommen. Andernfalls wird die Gruppe neu erzeugt.



TASM erlaubt aufgrund seines IDEAL-Modus auch die übliche »Verdrehung« der Reihenfolge der Angaben:

```
GROUP Name SegmentName [, SegmentName ...]
```



Die Verwendung von GROUP ist nicht ganz ungefährlich! In der Vergangenheit (bei älteren MASM-Versionen!) behandelte MASM Symbole, die in Segmenten deklariert und dann in einer Gruppe zusammengefasst wurden, teilweise unterschiedlich, je nachdem, ob und welchem Operator sie übergeben werden. So bezog der Operator OFFSET Daten immer auf das Segment, in dem sie deklariert wurden. Wurde dagegen das Datum selbst (genauer: der Zeiger, der hinter dem Symbol steht, das dem Datum zugeordnet ist) verwendet, erfolgte ein Bezug auf die Gruppe. Schwierig zu verstehen!

Daher ein Beispiel:

```
Data1 SEGMENT DWORD PUBLIC 'DATA'
DWord1 DD 1
Data1 ENDS

Data2 SEGMENT DWORD PUBLIC 'DATA'
DWord1 DD 2
Data1 ENDS

 DGroup GROUP Data1, Data2

Code SEGMENT PARA PUBLIC 'CODE'
ASSUME CS:Code DS:DGroup
Hier: MOV EAX, DWord1
      CMP EAX, DWord2
      JNZ OK
```

```
      CALL PRINT <Fehler! Fehler!>
OK:   MOV EAX, OFFSET DWord1
      CMP EAX, OFFSET DWord2
      JNZ Yeah
      CALL PRINT <So ein Mist!>
      RET
Yeah: CALL Print <Jippijeee!>
      RET
Code ENDS
END Hier
```

Was meinen Sie, was ausgedruckt wurde? »So ein Mist!« Warum?

Analysieren wir! Es werden zwei Datensegmente deklariert, die jedes ein DWORD an Offset $0000_0000 des entsprechenden Segments haben. Durch die Gruppierung nun sollten alle Datensegmente in ein einziges Gruppensegment namens DGroup aufgenommen werden. Somit ist doch anzunehmen, dass der Offset von DWord1 nun $0000_0000 ist und von DWord2 $0000_0004, und zwar bezogen auf das »neue« Gruppensegment. Das ist auch so. Dieses Gruppensegment machen wir nun dem Assembler via ASSUME als Quelle der Daten bekannt.

Verwenden wir nun die Symbole selbst, die ja für die Adressen stehen, so funktioniert alles prima. `MOV EAX, DWord1` lädt in EAX den Wert 1 und vergleicht ihn mit dem Wert »2«, der an der Adresse $0000_0004 steht, die ja nun hinter DWord2 steckt. Da die Daten nicht gleich sind, wird zum Label *OK* verzweigt. DWord2 beinhaltet somit korrekt den Offset des in Data2 an Offset $0000_0000 stehenden Datums, bezogen auf das Gruppensegment DGroup.

Also sollte man annehmen, dass die nächste MOV-Instruktion den Offset $0000_00000 von DWord1 im Gruppensegment in EAX lädt. Der anschließende Vergleich mit CMP sollte also eine Programmverzweigung nach *Yeah* hervorrufen, da ja $0000_0000 ≠ $0000_0004 ist! Genau das aber erfolgt nicht, was nahe legt, dass der Offset von DWord2 $0000_0000 geblieben ist, wenn der Operator OFFSET verwendet wird!! Dies widerspricht dem, was wir bei der direkten Nutzung des Symbols DWord2 festgestellt haben. Der Assembler bezieht also offensichtlich im Rahmen der Tätigkeit des Operators OFFSET die Offsets von Variablen immer auf *das Segment, in dem sie deklariert sind*, unerheblich, ob das Segment Teil einer Gruppe ist, oder nicht!

Dilemma. Was ist zu tun? Früher bestand die einzige Möglichkeit darin, mittels des Operators »:« einen »group override« durchzuführen und

eine *Qualifizierte Adresse* an OFFSET zu übergeben, die neben dem Offset auch das Bezugssegment enthält:

```
      :
      :
      MOV EAX, OFFSET DGROUP:DWord1
      CMP EAX, OFFSET DGROUP:DWord2
      JNZ Yeah
      CALL PRINT <So ein Mist!>
      RET
Yeah: CALL Print <Jippijeee!>
      RET
```

Dies war letztendlich *ein* Grund, warum Borland den IDEAL-Modus von TASM »erfunden« hat. Heute (MASM 6.1x) gibt es die Direktive OPTION (vgl. Seite 679) mit deren Option OFFSET festgelegt werden kann, ob der Assembler die Offsets der Variablen auf das Segment oder die Gruppe bezieht.

Aufpassen! Leider unterstützt TASM (TASM 5.3) diese Option *nicht*! Daher ist im MASM-Modus der oben geschilderte Fall anzunehmen, was zu wirklich schwer aufzufindenden Fehlern führt, da man allzu leicht diesen group override vergisst. In diesem Fall sollte TASMs IDEAL-Modus verwendet werden, der diese Probleme nicht hat.

ASSUME

Wenn auf Daten zugegriffen wird, benötigt die CPU eine vollständige logische Adresse, um eine physikalische Adresse berechnen zu können und das entsprechende Datum aufzufinden (vgl. »Zugriffe auf den Speicher: Von Adressen und Adressräumen« auf Seite 434). Andererseits verwendet der Programmierer im Quelltext Symbole (= Variablennamen), um das Datum zu identifizieren. Diese Symbole stehen aber für einen Offset relativ zum Beginn des Segments, in dem das Datum deklariert ist. Somit handelt es sich bei den Adressen, mit denen im Rahmen der Assemblerprogrammierung umgegangen wird, um die effektiven Adressen.

Um nun die korrekten Befehlssequenzen generieren zu können, muss also der Assembler den Segment-Anteil der logischen Adresse eruieren. Keine Kunst, denken Sie, handelt es sich doch um das Segment, in dem das Datum deklariert worden ist. Richtig. Das Problem ist nur: Der Assembler benötigt als Segment-Anteil ein Segmentregister (CS, DS, ES, FS, GS oder SS)! Und welches nun? Welches Segmentregister hat bereits oder wird spätestens zur Laufzeit des Programms den Selektor aufnehmen, der das aktuelle Datensegment spezifiziert?

Bei der Deklaration von Segmenten wird keinerlei Angabe gemacht, in welchem Segmentregister der Segmentanteil des gerade deklarierten Segmentes steht oder stehen wird. Kann auch gar nicht! Denn wenn z.B. mehrere Datensegmente deklariert werden – woher sollte der Assembler wissen, wann welches Datensegment in welches Segmentregister geladen wird? Das weiß nur der Programmierer, der ja auch die entsprechenden Codesequenzen programmieren muss, die das bewerkstelligen. Daher ist er gefordert, dem Assembler genau das mitzuteilen.

ASSUME dient genau diesem Zweck. Mit ASSUME teilt der Programmierer dem Assembler mit, dass dieser annehmen (»assume«) soll, dass von nun an – bis auf Widerruf! – Segmentregister XX den Segment-Selektor von Segment YY enthält. Dies versetzt den Assembler in die Lage, korrekte Befehlssequenzen zum Adressieren von Daten im Speicher zu generieren. So kann er z.B. darauf verzichten, mit Segment-Override-Präfixen zu arbeiten, wenn das anzusprechende Datum in einem Segment liegt, dessen Selektor sich gerade in DS befindet, da bei Datenzugriffen standardmäßig das DS-Register involviert wird. Andererseits benutzt er den Segment-Override-Präfix ES:, wenn auf ein Datum zurückgegriffen werden soll, dessen Segment-Selektor sich zurzeit im Segmentregister ES befindet.



ASSUME generiert also selbst keinen Code! Es ist jedoch essentiell, wenn Code generiert werden soll, der auf Speicherstellen zurückgreift, da dann Informationen einfließen, die der Programmierer mittels ASSUME gegeben hat. Da somit ASSUMEs nur »Tipps« des Programmierers sind, die der Assembler nicht überprüfen kann, kann auch keine Fehlermeldung generiert werden, wenn der Programmierer versehentlich oder absichtlich »gelogen« hat. Denn ob tatsächlich die Segment-Segmentregister-Beziehung hergestellt wird, die der Programmierer mit ASSUME suggeriert, bleibt dem Assembler verborgen!

Verwendung ASSUME wird formal wie folgt verwendet:

```
ASSUME SegReg:Expression [, SegReg:Expression ...]
```

oder

```
ASSUME NOTHING
```

wobei *SegReg* ein Segmentregister (CS, DS, ES, FS, GS oder SS) darstellt und *Expression* den Namen eines Segmentes oder einer Gruppe oder das Schlüsselwort »NOTHING« ergeben muss. Wird das Schlüsselwort NOTHING in Verbindung mit einem Segmentregister verwendet, wer-

den alle bestehenden Bezüge zwischen diesem Segmentregister und einem Segment oder einer Gruppe gelöscht. *ASSUME NOTHING* löscht alle Bezüge für alle Segmentregister.



MASM und der MASM-Modus von TASM bieten eine Syntaxerweiterung von ASSUME, die sich auf »normale« Register ausdehnt:

```
ASSUME Reg:Expression [, Reg:Expression ...]
```

Reg kann hier ein beliebiges Allzweckregister sein. *Expression* muss entweder einen Qualifizierten Typen (vgl. Seite 560) mit der Größe von *Reg* ergeben oder die Schlüsselworte ERROR oder NOTHING darstellen.

Mit dieser Syntaxerweiterung werden der Nutzung von Allzweckregistern Restriktionen auferlegt, die durch den Qualifizierten Typ definiert werden. Beispiel:

```
:
ASSUME EAX:DWORD              ; In EAX stehen nur DWORDS!
MOV    EBX, [EAX]             ; Fehler: EAX enthält Pointer
ASSUME EBX: PTR DWORD
MOV    EAX, [EBX]             ; OK
MOV    EAX, EBX               ; Fehler: EBX enthält DWord
:
```

Mit `ASSUME EAX:DOWRD` wird dem Assembler mitgeteilt, dass das Register EAX nur »echte« DWORDs aufnehmen darf, nicht aber andere Daten, die DWORD-Größe haben, wie z.B. 32-Bit-Pointer. Die Verwendung von `MOV EBX, [EAX]` muss somit einen Fehler produzieren, da hier der Inhalt von EAX als Adresse auf ein DWORD interpretiert wird, was durch ASSUME ausgeschlossen wurde.

Wird das Schlüsselwort ERROR verwendet, so wird bei jeder impliziten oder expliziten Verwendung des Registers eine Fehlermeldung generiert:

```
:
ASSUME EAX:ERROR
MOV    EAX, EBX               ; Fehler: EAX ist gesperrt!
:
```

Das Schlüsselwort ERROR sperrt auch bei MASM oder im MASM-Modus von TASM Segmentregister.



END END markiert das Ende des Quelltextes. Alle Textzeilen nach END werden vom Assembler ignoriert. END wird formal wie folgt verwendet:

```
END [Startadresse]
```

wobei *Startadresse* optional die Adresse angibt, an der die Programmausführung beginnen soll. Diese Startadresse wird dem Linker als Einsprungsadresse übergeben.



Startadresse macht nur Sinn in Modulen, die zu COM- oder EXE-Files gelinkt werden und somit einen Punkt definieren müssen, dem der Programmlader die Ausführung übergibt. Dann darf jedoch nur eines der Module, die zusammengelinkt werden, diese Startadresse angeben. In Modulen, die im Rahmen von anderen Modulen aufgerufen werden (DLLs, in Hochsprachen eingebundene Units etc.) macht *Startadresse* keinen Sinn und muss somit entfallen! Die Einsprungspunkte werden in diesem Fall mit PUBPLIC o. Ä. deklariert.



Wird END in Modulen verwendet, die die Vereinfachte Segmentkontrolle (siehe nächstes Kapitel) benutzen und mit ihr die .STARTUP-Direktive, darf *Startadresse* ebenfalls nicht angegeben werden, da dies durch .STARTUP erfolgt.

Segmentanordnung Der Linker ist für das Zusammenbinden der verschiedenen Segmente verantwortlich. Im Allgemeinen erfolgt dies sequentiell in der Reihenfolge, in sie im Quelltext erscheinen und/oder (z.B. via INCLUDE) eingebunden werden. Der Assembler stellt jedoch drei Direktiven zur Verfügung, mit der – hauptsächlich aus Kompatibilitätsgründen – diese Reihenfolge verändert werden kann.

.ALPHA So kann mit der Direktive .ALPHA der Linker angewiesen werden, die Module in alphabetischer Reihenfolge zu linken. Ausschlaggebend ist hierbei der jeweilige Segmentname.

.SEQ Mit .SEQ dagegen kann erzwungen werden, dass die Segmente modulweise in der Reihenfolge ihrer Deklaration im Modul gelinkt werden. Dies ist auch die Defaulteinstellung.

.DOSSEG Aus Kompatibilitätsgründen zur unter DOS üblichen Segmentierung gibt es auch die Direktive .DOSSEG, die die Segmente in der Reihenfolge Code-Segmente (*class*: 'CODE') – Nicht-DGROUP-Segmente (*class*: nicht 'CODE') – DGROUP-Segmente linken, wobei bei den DGROUP-Segmenten noch die Reihenfolge Nicht-BSS/Stack-Segmente (initiali-

sierte Daten) – BSS-Segmente (uninitialisierte Daten) – Stack-Segmente (*combine*: STACK) eingehalten wird.

TASM erlaubt aufgrund seines IDEAL-Modus auch die punktlose Version von .DOSSEG.



3.2.8 Vereinfachte Segmentkontrolle

Im vorangehenden Kapitel haben Sie gesehen, dass mit den Direktiven zur Segmentkontrolle sehr viele Einstellungen an Segmenten vorgenommen werden können, die Sie in Hochsprachen entweder gar nicht oder nur sehr umständlich realisieren können. Nun ist es jedoch nicht immer nötig und/oder erwünscht, eine so tief greifende Kontrolle über das Geschehen zu haben. Vielmehr legt man erheblich größeren Wert auf Kompatibilität zu Hochsprachen, da man evtl. Assemblermodule entwickelt, die in Hochsprachen eingesetzt werden sollen. Es wäre somit sehr schön, Schnittstellen nutzen zu können, die die Konventionen der betreffenden Hochsprache einhält und auch ein wenig »Programmierkomfort« zur Verfügung stellt, der einem erforderlichen, aber auf den Assembler abwälzbaren Ballast abnimmt.

MASM und TASM ermöglichen dies durch die Direktiven zur Vereinfachten Segmentkontrolle. Zentrale Direktive hierbei ist die Direktive .MODEL, da sie die Standardwerte einstellt, mit denen die Randbedingungen eingestellt werden und auf die die anderen Direktiven aufbauen.

.MODEL

.MODEL ermöglicht einem, ein bestimmtes »Standardprogrammiermodell« auszuwählen und somit verschiedene Einstellungen automatisch vorzunehmen, die andernfalls über Direktiven zur Segmentkontrolle (siehe vorangehendes Kapitel) und/oder Realisierung von Prozeduren (vgl. »Direktiven zur Deklaration und Nutzung von Prozeduren« auf Seite 610) manuell vorgenommen werden müssten. Hierzu gehören auch eine implizierte Gruppierung mittels GROUP sowie Annahmen über die Segmente mittels ASSUME.

Verwendung

.MODEL wird formal wie folgt verwendet:

```
.MODEL Speichermodell [, Sprache] [, OS] [, Stack]
```

Für das *Speichermodell* gibt es folgende Schlüsselworte: TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE und FLAT, bei TASM auch noch TCHUGE. Die Unterschiede zwischen den dadurch spezifizierten Modellen finden Sie in Tabelle 3.7. Diese Schlüsselworte legen fest, welche

Segmentattribute verwendet werden sollen, wenn Code-, Daten- und Stacksegmente mit den hierzu zur Verfügung stehenden Direktiven (.CODE, .CONST, .DATA, .DATA?, .FARDATA und .FARDATA?; s.u.) eingerichtet werden. Sie legen ferner fest, welche Pointertypen zur Anwendung kommen und welche Annahmen über die Zusammenhänge zwischen Segment und Segmentregister bestehen (ASSUME). Tabelle 3.7 zeigt die Zusammenhänge.

	Segmentanzahl f.					
Modell	**Code**	**Daten**	**Umgebung**	**Code**	**Daten**	**ASSUME**
TINY	zusammen 1		16 Bit	near	near	CS = DS = SS = DGROUP
SMALL	1	1	16/32 Bit	near	near	CS = _TEXT DS = SS = DGROUP
MEDIUM	beliebig	1	16/32 Bit	far	near	CS = *Name*_TEXT DS = SS = DGROUP
COMPACT	1	beliebig	16/32 Bit	near	far	CS = _TEXT DS = SS = DGROUP
LARGE	beliebig	beliebig	16/32 Bit	far	far	CS = *Name*_TEXT DS = SS = DGROUP
HUGE	beliebig	beliebig	16/32 Bit	far	far	CS = *Name*_TEXT DS = SS = DGROUP
FLAT	1	1	32 Bit	near	near	CS = _TEXT DS = ES = SS = DGROUP
TCHUGE	beliebig	beliebig	16/32 Bit	far	far	CS = *Name*_TEXT DS = SS = NOTHING

LARGE und HUGE unterscheiden sich auf Assemblerebene nicht und dienen lediglich der Herstellung von Konsistenzen mit anderen Programmen. TCHUGE ist ein TASM-spezifisches Modell, das ein Interface zum HUGE-Model von Borland C++ herstellt.

Tabelle 3.7: Standardwerte, die Anzahl von Code- und Datensegmenten, Umgebung, Code- und Datenpointer und ASSUME-Argumente in Abhängigkeit des gewählten Programmiermodells

Sprache stellt die im folgenden Modul als Standard anzusehenden Konventionen ein, nach denen Deklarationen und Aufruf von Prozeduren etc. erfolgen sollen. Dieses Argument stellt somit das zu wählende Interface ein, das für die Einbindung von Assemblermodulen in Hochsprachen erforderlich ist. Erlaubte Argumente sind C, SYSCALL, STDCALL, BASIC, FORTRAN und PASCAL, bei TASM auch CPP (= C++) und NOLANGUAGE (= Assembler). Die mit diesem Argument eingestellte Sprache kann in jeder Deklaration von Prozeduren oder ihren Prototypen oder beim Aufruf solcher mittels INVOKE/CALL für den jeweiligen Einzelfall abgeändert werden (vgl. hierzu Tabelle 3.5 auf Seite 612).

OS stellt das Betriebssystem dar, für das das Modul entwickelt wird. Zur Verfügung stehen die Schlüsselworte OS_DOS, OS_NT und OS_OS2. Sie dienen unter anderem der Definition der verwendeten Segmentgrößen und haben somit Einfluss z.B. auf die Spalte »Umgebung« in Tabelle 3.7. Das Modell TINY ist nur mit OS_DOS (= 16-Bit DOS, WINDOWS 3.x) erlaubt, das Modell FLAT nur mit OS_NT und OS_OS2 (= Windows NT, 2000, XP, 95, 98, 98SE, ME, OS/2). Weiteren Einfluss nimmt das als *OS* übergebene Argument auf die Codefolge, die durch .STARTUPCODE und .EXIT generiert werden.

STACK gibt an, ob der Stack Teil der Gruppe der Datensegmente sein soll oder nicht. Wird NEARSTACK als Argument übergeben, was auch der Default ist, so wird das Stacksegment in die Gruppe der Datensegmente aufgenommen und DS = SS. Bei FARSTACK ist dies nicht der Fall. In diesem Fall generiert .STARTUPCODE auch einen anderen Prolog.



TASM bietet mit *MODEL* ein Synonym für *.MODEL* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann. Die Syntax wurde ein wenig erweitert und hat sowohl im MASM- als auch im IDEAL-Modus folgende Form:

```
MODEL [ModelMod] Modell [Name] [, [SprachMod] Sprache]
```

`ModelMod` modifiziert das mit *Modell* angegebene Modell und kann die Schlüsselworte NEARSTACK, FARSTACK, USE16, USE32, OS_DOS, DOS, OS_NT, NT, OS_OS2 und OS2 annehmen, die bereits erläutert wurden oder klar sein dürften.

Name spielt nur dann eine Rolle, wenn ein Modell ausgewählt wird, das mehrere Codesegmente zulässt. In diesem Fall findet sich *Name* als Teil des Codesegmentnamen in der Form *Name*_TEXT wieder (vgl. Tabelle 3.7).

SprachMod ist ein Modifizierer für die gewählte Sprache und wird im Rahmen des Prolog und Epilog erforderlich, wenn Routinen aus dem Modul direkt von Windows aufgerufen werden sollen oder den Borland Overlay Loader nutzen. Gültige Argumente hierfür sind NORMAL, WINDOWS, ODDNEAR und FARNEAR. Vergleiche hierzu Seite 614.



Alle in der .MODEL-Direktive gemachten Angaben können bei der Deklaration von Prozeduren und ihren Prototypen oder bei Export oder Import von Labels, Variablen und Symbolen lokal abgeändert werden.



.MODEL deklariert einige Symbole, die im Quelltext verwendet werden können:

@Model besitzt abhängig vom gewählten Modell die Werte 1 (TINY), 2 (SMALL), 3 (COMPACT), 4 (MEDIUM), 5 (LARGE) oder 6 (HUGE). Leider treten hier Inkompatibilitäten zwischen MASM und TASM auf: Während MASM mit 7 das FLAT-Modell codiert, repräsentiert dieser Wert bei TASM das Modell TCHUGE. Unter TASM ist das Modell FLAT das gleiche wie SMALL und gibt somit den Wert 2 zurück. Somit kann bei Verwendung von MASM zwischen SMALL und FLAT unterschieden werden, bei TASM nicht!

@32Bit Wurde mit der .MODEL-Direktive eine Segmentgröße von 16 Bit eingestellt (USE16), so hat @32Bit den Wert »0«, andernfalls »1«.

@CodeSize @CodeSize gibt als Wert »0« zurück, wenn im gewählten Modell das Codesegment NEAR-Adressen benutzt. Dies ist der Fall bei den Modellen TINY, SMALL, COMPACT, und FLAT. Andernfalls werden FAR-Adressen verwendet und @CodeSize liefert »1«.

@DataSize Analog gibt @DataSize »0« zurück, wenn Daten über NEAR-Pointer angesprochen werden können, was in den Modellen TINY, SMALL, MEDIUM und FLAT der Fall ist. Andernfalls müssen FAR-Pointer verwendet werden, was @DataSize mit dem Wert »1« signalisiert.

@Interface @Interface liefert einen Code für das verwendete Hochsprachen-Interface zurück. Derzeit werden 8 Bits für den Code benutzt: Bit 7 signalisiert das eingestellte Betriebssystem und ist 0 bei DOS und Windows 3.x bzw. 1 bei Windows 95, 98, 98SE, ME, NT, 2000, XP und OS2. Die Bits 6 bis 0 codieren einen Wert für die eingestellte Sprache: NOLANGUAGE (0; TASM) bzw. RESERVERD (0; MASM); C (1); SYSCALL (2); STDCALL (3); PASCAL (4); FORTRAN (5); BASIC (6); PROLOG (7; TASM) bzw. RESERVED (7; MASM) und CPP (C++; 8).

.CODE

.CODE leitet ein Codesegment ein, nachdem mit .MODEL ein Programmiermodell angegeben wurde. .CODE »übersetzt« dann die dort definierten Randbedingungen in die notwendigen Direktiven. Der Abschluss des Segments mittels ENDS ist nicht erforderlich. So sind z.B. folgende Direktivensequenzen absolut identisch:

```
.486
.Model FLAT
.Code
        :
        :
END
```

```
.486
_TEXT SEGMENT WORD PUBLIC 'CODE'
ASSUME CS: TEXT
        :
        :
_TEXT ENDS
END
```

Verwendung

.CODE wird wie folgt verwendet:

```
.CODE [Name]
```



Erlaubt das gewählte Programmiermodell mehrere Codesegmente (MEDIUM, LARGE, HUGE, TCHUGE), setzt sich der standardmäßig für das Codesegment verwendete Name aus dem Modulnamen (gleichbedeutend mit dem Namen des Assemblerfiles) und dem Anhang »_TEXT« zusammen. Bei Vorliegen nur eines Codesegments wird *_TEXT* als Standardname verwendet. Optional können Sie jedoch mit *Name* einen Namen für das Segment vorgeben.

Tabelle 3.8 zeigt die Argumente, die .CODE in Abhängigkeit des gewählten Programmiermodells nutzt.

Modell	Segment-Attribut				
	Name	Align *	Combine	Class	Group
TINY	_TEXT	(D)WORD	PUBLIC	'CODE'	DGROUP
SMALL	_TEXT	(D)WORD	PUBLIC	'CODE'	
MEDIUM	*Name*_TEXT	(D)WORD	PUBLIC	'CODE'	
COMPACT	_TEXT	(D)WORD	PUBLIC	'CODE'	
LARGE	*Name*_TEXT	(D)WORD	PUBLIC	'CODE'	
HUGE	*Name*_TEXT	(D)WORD	PUBLIC	'CODE'	
FLAT	_TEXT	(D)WORD	PUBLIC	'CODE'	
TCHUGE	*Name*_TEXT	(D)WORD	PUBLIC	'CODE'	

* Wenn die Direktiven **.**8086, **.**186 oder **.**286 angegeben werden oder der entsprechende Befehlssatz aktiv ist, wird das Segment an WORD-Grenzen ausgerichtet, andernfalls an DWORD-Grenzen.

Tabelle 3.8: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .CODE



TASM bietet mit *CODESEG* ein Synonym für **.***CODE* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.

.CONST **.**CONST leitet ein Datensegment für initialisierte Daten ein, nachdem mit **.**MODEL ein Programmiermodell angegeben wurde. **.**CONST »übersetzt« dann die dort definierten Randbedingungen in die notwendigen Direktiven. Der Abschluss des Segments mittels ENDS ist nicht erforderlich. So sind z.B. folgende Direktivensequenzen absolut identisch:

```
.486
.Model FLAT
.CONST
        :
        :
END


.486
CONST SEGMENT WORD PUBLIC 'CONST'
DGROUP GROUP CONST
ASSUME DS: DGROUP
        :
        :
CONST ENDS
END
```

.CONST wird wie folgt verwendet: *Verwendung*

```
.CONST
```

Tabelle 3.9 zeigt die Argumente, die .CONST in Abhängigkeit des gewählten Programmiermodells nutzt.

Modell	Segment-Attribut				
	Name	**Align ***	**Combine**	**Class**	**Group**
TINY SMALL MEDIUM COMPACT LARGE HUGE FLAT	CONST	(D)WORD	PUBLIC	'CONST'	DGROUP
TCHUGE	-	-	-	-	-

* Wenn die Direktiven .8086, .186 oder .286 angegeben werden oder der entsprechende Befehlssatz aktiv ist, wird das Segment an WORD-Grenzen ausgerichtet, andernfalls an DWORD-Grenzen.

Tabelle 3.9: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .CONST

TASM bietet mit *CONST* ein Synonym für *.CONST* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.



.DATA

.DATA leitet ein Datensegment für initialisierte Daten ein, nachdem mit .MODEL ein Programmiermodell angegeben wurde. .DATA »übersetzt« dann die dort definierten Randbedingungen in die notwendigen Direktiven. Der Abschluss des Segments mittels ENDS ist nicht erforderlich. So sind z.B. folgende Direktivensequenzen absolut identisch:

```
.486
.Model FLAT
.DATA
        :
        :
END

.486
DATA SEGMENT WORD PUBLIC 'DATA'
DGROUP GROUP _DATA
ASSUME DS: DGROUP
        :
        :
DATA ENDS
END
```

Verwendung .DATA wird wie folgt verwendet:

```
.DATA
```

Tabelle 3.10 zeigt die Argumente, die .DATA in Abhängigkeit des gewählten Programmiermodells nutzt.

Modell	Segment-Attribut				
	Name	Align *	Combine	Class	Group
TINY SMALL MEDIUM COMPACT LARGE HUGE FLAT	_DATA	(D)WORD	PUBLIC	'DATA'	DGROUP
TCHUGE	*Name*_DATA	PARA	PRIVATE	'DATA'	

* Wenn die Direktiven .8086, .186 oder .286 angegeben werden oder der entsprechende Befehlssatz aktiv ist, wird das Segment an WORD-Grenzen ausgerichtet, andernfalls an DWORD-Grenzen.

Tabelle 3.10: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .DATA



TASM bietet mit *DATASEG* ein Synonym für *.DATA* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.

.DATA? .DATA? leitet ein Datensegment für uninitialisierte Daten ein, nachdem mit .MODEL ein Programmiermodell angegeben wurde. .DATA? »übersetzt« dann die dort definierten Randbedingungen in die notwendigen Direktiven. Der Abschluss des Segments mittels ENDS ist nicht erforderlich. So sind z.B. folgende Direktivensequenzen absolut identisch:

```
.486
.Model FLAT
.DATA?
       :
       :
END


.486
_BSS SEGMENT WORD PUBLIC 'BSS'
```

```
DGROUP GROUP BSS
ASSUME DS: DGROUP
        :
        :
_BSS ENDS
END
```

.DATA? wird wie folgt verwendet: *Verwendung*

```
.DATA?
```

Tabelle 3.11 zeigt die Argumente, die **.**DATA? in Abhängigkeit des gewählten Programmiermodells nutzt.

Modell	**Segment-Attribut**				
	Name	**Align ***	**Combine**	**Class**	**Group**
TINY SMALL MEDIUM COMPACT LARGE HUGE FLAT	_BSS	(D)WORD	PUBLIC	'BSS'	DGROUP
TCHUGE	-	-	-	-	-

* Wenn die Direktiven **.**8086, **.**186 oder **.**286 angegeben werden oder der entsprechende Befehlssatz aktiv ist, wird das Segment an WORD-Grenzen ausgerichtet, andernfalls an DWORD-Grenzen.

Tabelle 3.11: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .DATA?

TASM bietet mit *UDATASEG* ein Synonym für **.***DATA?* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.



.FARDATA

.FARDATA leitet ein Datensegment für initialisierte Daten ein, nachdem mit **.**MODEL ein Programmiermodell angegeben wurde. Dieses Datensegment ist FAR deklariert, was bedeutet, dass darauf nur mit FAR-Pointern zugegriffen werden kann. **.**FARDATA »übersetzt« dann die dort definierten Randbedingungen in die notwendigen Direktiven. Der Abschluss des Segments mittels ENDS ist nicht erforderlich. So sind z.B. folgende Direktivensequenzen absolut identisch:

```
.486
.Model FLAT
.FARDATA
        :
        :
END


.486
FAR_DATA SEGMENT PARA PRIVATE 'FAR_DATA'
ASSUME DS: NOTHING
        :
        :
FAR_DATA ENDS
END
```

Verwendung .FARDATA wird wie folgt verwendet:

```
.FARDATA [Name]
```



Da Daten in Segmenten, die mit .FARDATA deklariert werden, über FAR-Pointer angesprochen werden, können mehrere Datensegmente dieses Typs existieren. Falls das der Fall ist, können sie mit *Name* individuell benannt werden. Wird *Name* nicht angegeben, wird *FAR_DATA* verwendet.

Tabelle 3.12 zeigt die Argumente, die .FARDATA in Abhängigkeit des gewählten Programmiermodells nutzt.

Modell	Segment-Attribut				
	Name	Align	Combine	Class	Group
TINY	-	-	-	-	-
SMALL MEDIUM COMPACT LARGE HUGE FLAT TCHUGE	FAR_DATA	PARA	PRIVATE	'FAR_DATA'	

Tabelle 3.12: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .FARDATA

TASM bietet mit *FARDATA* ein Synonym für *.FARDATA* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.



.FARDATA?

.FARDATA? leitet ein Datensegment für uninitialisierte Daten ein, nachdem mit .MODEL ein Programmiermodell angegeben wurde. Dieses Datensegment ist FAR deklariert, was bedeutet, dass darauf nur mit FAR-Pointern zugegriffen werden kann. .FARDATA? »übersetzt« dann die dort definierten Randbedingungen in die notwendigen Direktiven. Der Abschluss des Segments mittels ENDS ist nicht erforderlich. So sind z.B. folgende Direktivensequenzen absolut identisch:

```
.486
.Model FLAT
.FARDATA?
        :
        :
END
```

```
.486
FAR_BSS SEGMENT PARA PRIVATE 'FAR_BSS'
ASSUME DS: NOTHING
        :
        :
FAR_BSS ENDS
END
```

Verwendung

.FARDATA? wird wie folgt verwendet:

```
.FARDATA? [Name]
```



Da Daten in Segmenten, die mit .FARDATA? deklariert werden, über FAR-Pointer angesprochen werden, können mehrere Datensegmente dieses Typs existieren. Falls das der Fall ist, können sie mit *Name* individuell benannt werden. Wird *Name* nicht angegeben, wird *FAR_BSS* verwendet.

Tabelle 3.13 zeigt die Argumente, die .FARDATA in Abhängigkeit des gewählten Programmiermodells nutzt.

Modell	Segment-Attribut				
	Name	Align	Combine	Class	Group
TINY	-	-	-	-	-
SMALL MEDIUM COMPACT LARGE HUGE FLAT TCHUGE	FAR_BSS	PARA	PRIVATE	'FAR_BSS'	

Tabelle 3.13: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .FARDATA?



TASM bietet mit *UFARDATASEG* ein Synonym für **.***FARDATA?* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.

.STACK .STACK leitet ein Stacksegment ein, nachdem mit .MODEL ein Programmiermodell angegeben wurde. .STACK »übersetzt« dann die dort definierten Randbedingungen in die notwendigen Direktiven und sorgt dafür, dass das SS- und (E)SP-Register initialisiert wird. Der Abschluss des Segments mittels ENDS ist nicht erforderlich. So sind z.B. folgende Direktivensequenzen absolut identisch:

```
.486
.Model FLAT
.STACK
        :
        :
END


.486
STACK SEGMENT WORD PUBLIC 'STACK'
DGROUP GROUP STACK
ASSUME SS: DGROUP
        :
        :
STACK ENDS
END
```

Verwendung .STACK wird wie folgt verwendet:

```
.STACK [Size]
```

Optional kann der Direktive die Größe des zu verwendenden Stacks in Bytes angegeben werden. Erfolgt dies nicht, wird 1024 angenommen.



Tabelle 3.14 zeigt die Argumente, die .FARDATA in Abhängigkeit des gewählten Programmiermodells nutzt.

Modell	Segment-Attribut				
	Name	**Align**	**Combine**	**Class**	**Group**
TINY SMALL MEDIUM COMPACT LARGE HUGE FLAT	STACK	PARA	STACK	'STACK'	DGROUP
TCHUGE	STACK	PARA	STACK	'STACK'	

Tabelle 3.14: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .STACK

TASM bietet mit *STACK* ein Synonym für *.STACK* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.



.STARTUP

.STARTUP generiert Code, der zu Beginn des Segments noch vor der ersten durch den Programmierer angegebenen Instruktion ausgeführt wird (»Prolog«). Art und Umfang dieses Codes richten sich nach dem gewählten Betriebssystem (DOS, WINDOWS), nach dem Programmiermodell und nach der Art des Stacks. Darüber hinaus erzeugt .STARTUP ein Label für den Einsprungspunkt (mit der Adresse der ersten ausführbaren Instruktion dieses generierten Codes), das dem Linker als Einstieg übergeben wird, sodass auf die entsprechende Angabe im Rahmen von END verzichtet werden kann. TASM generiert darüber hinaus ein Symbol namens @Startup, das diese Adresse erhält.

TASM bietet mit *STARTUPCODE* ein Synonym für *.STARTUP* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.



.EXIT

.EXIT generiert Code, der am Ende des Segments nach der letzten durch den Programmierer angegebenen Instruktion ausgeführt wird (»Epilog«). Art und Umfang dieses Codes richten sich nach dem gewählten Betriebssystem (DOS, WINDOWS), nach dem Programmiermodell und nach der Art des Stacks. .EXIT kann ein Wert übergeben

werden, den der Epilog dann ggf. als ExitCode an das Betriebssystem oder das rufende Modul zurückgibt:

```
.EXIT [Expression]
```



Wird kein Wert angegeben, wird der in AX stehende Wert verwendet.

TASM bietet mit *EXITCODE* ein Synonym für *.EXIT* (bitte beachten Sie den führenden Punkt!), das sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden kann.

LARGESTACK
SMALLSTACK



Üblicherweise erzeugt die .MODEL-Direktive automatisch Stacks mit der korrekten »Körnung« (16- bzw. 32-Bit-Daten) anhand des gewählten Programmiermodells. Dennoch kann es vorkommen, dass eine andere als die automatisch generierte Körnung des Stacks gewünscht wird, vor allem, wenn auf die Vereinfachte Segmentkontrolle verzichtet wird. Hierzu dienen die Direktiven LARGESTACK (32-Bit-Körnung) und SMALLSTACK (16-Bit-Körnung).

3.2.9 Direktiven zur bedingten Steuerung des Programmablaufs

Im folgenden Abschnitt werden Direktiven aufgelistet, die unter Assembler eine ähnlich einfache Nutzung von IF-THEN-ELSE-, WHILE-DO- oder REPEAT-UNITL-Konstrukten erlauben wie in Hochsprachen.



Verwechseln Sie diese Direktiven, die alle einen Punkt vor dem Direktivennamen haben, nicht mit den punktlosen Zwillingen, die bei der bedingten Assemblierung zum Einsatz kommen! Während bei den punktlosen Direktiven das Ergebnis der Kompilierung abhängig von der Erfüllung einer Bedingung zum Zeitpunkt der Assemblierung ist (= bedingte Assemblierung, vgl. Seite 662), generieren die folgenden Direktiven Code, der den entsprechenden Konstrukten aus Hochsprachen entspricht. Die Bedingungen, die diese Konstrukte prüfen, werden somit erst zur Laufzeit des assemblierten Moduls geprüft und entsprechend reagiert.

.IF
.ELSEIF
.ELSE
.ENDIF

Mit diesen Direktiven sind Konstrukte programmierbar, die in Delphi den Blöcken IF – THEN – ELSE und in C/C++ den Blöcken IF ( ) – ELSE entsprechen. Die zwischen .IF und .ELSE (bzw. .ELSEIF oder .ENDIF) eingeschlossenen Statements werden ausgeführt, wenn eine Bedin-

gung erfüllt ist, die mit .IF definiert wird. Analoges gilt für .ELSEIF und dem nächsten .ELSEIF bzw. einem abschließenden .ELSE oder .ENDIF. Wurde bis zu einem eventuell vorhandenen .ELSE keine wahre Bedingung vorgefunden (und somit die dazugehörigen Statements ausgeführt), werden die zwischen .ELSE und .ENDIF stehenden Statements ausgeführt.

Formal werden diese Direktiven wie folgt verwendet: *Verwendung*

```
.IF Condition
   Statement(s)
[.ELSEIF Condition
   Statement(s)]
:
:
[.ELSE
   Statement(s)]
.ENDIF
```

Hierbei ist *Condition* ein Ausdruck, der als Ergebnis »true« oder »false« ergeben muss. *Condition* kann zur Evaluierung des Ergebnisses »Run-Time-Operatoren« verwenden (vgl. Seite 690).

.WHILE
.ENDWHILE

Mit diesen Direktiven sind Schleifen programmierbar, die in Delphi der Schleife WHILE – DO und in C/C++ der Schleife WHILE ( ) entsprechen. Die zwischen .WHILE und .ENDWHILE eingeschlossenen Statements werden so lange ausgeführt, wie eine Bedingung erfüllt ist, die bei .WHILE definiert wird.

Formal werden diese Direktiven wie folgt verwendet: *Verwendung*

```
.WHILE Condition
   Statement(s)
.ENDWHILE
```

Hierbei ist *Condition* ein Ausdruck, der als Ergebnis »false« oder »false« ergeben muss. *Condition* kann zur Evaluierung des Ergebnisses »Run-Time-Operatoren« verwenden (vgl. Seite 690).

.REPEAT
.UNTIL
.UNTILCXZ

Mit diesen Direktiven sind Schleifen programmierbar, die in Delphi der Schleife REPEAT – UNTIL und in C/C++ der Schleife DO – WHILE ( ) entsprechen. Die zwischen .REPEAT und .UNTIL (.UNTILCXZ) eingeschlossenen Statements werden so lange ausgeführt, bis eine Bedingung erfüllt ist, die mit .UNITL (.UNTILCXZ) definiert wird.

Verwendung Formal werden diese Direktiven wie folgt verwendet:

```
.REPEAT
   Statement(s)
.UNTIL Condition
```

oder

```
.REPEAT
   Statement(s)
.UNTILCXZ
```

Hierbei ist *Condition* ein Ausdruck, der als Ergebnis »false« oder »false« ergeben muss. *Condition* kann zur Evaluierung des Ergebnisses »Run-Time-Operatoren« verwenden (vgl. Seite 690). Wird .UNTILCXZ verwendet, so wird eine Bedingung impliziert: .UNTILCXZ liefert »true« zurück, wenn der Inhalt von (E)CX = 0 ist, andernfalls »false«.

.BREAK
.CONTINUE

.BREAK und .CONTINUE haben in Assembler die gleiche Bedeutung wie BREAK und CONTINUE in C/C++ und Delphi. Sie können in .REPEAT – .UNITL (.UNITLCXZ) – und .WHILE – .ENDWHILE – Schleifen eingesetzt werden, um die Schleife vorzeitig zu verlassen (.BREAK) oder eine unmittelbare Prüfung der Bedingung (.CONTINUE) durchzuführen. .BREAK veranlasst somit einen unbedingten Sprung zu der auf die Schleife unmittelbar folgenden Instruktion, .CONTINUE einen zu der Instruktion, die die Bedingung in der Schleife prüft.

Beide Direktiven können mit einer .IF-Direktive kombiniert werden. In diesem Fall erfolgt die Ausführung von .BREAK oder .CONTINUE nur, wenn die durch .IF definierte Bedingung erfüllt ist.

Verwendung Die formale Verwendung erfolgt demnach wie folgt:

```
.BREAK [.IF Condition]
.CONTINUE [.IF Condition]
```



In verschachtelten Schleifen verlässt .BREAK nur die Schleife, innerhalb der die Direktive steht. Die Ausführung wird in diesem Fall somit innerhalb der nächsten »übergeordneten« Schleife fortgesetzt. Analog springt .CONTINUE zu dem die Bedingung prüfenden Schleifenteil der Schleife, in der die Direktive steht. Es erfolgt in diesem Fall kein Sprung in die »übergeordnete« Schleife!

3.2.10 Makros

Makros sind in Assembler das, was man in Textverarbeitungen gerne »Textbausteine« nennt. Es sind somit Sequenzen aus Direktiven und Instruktionen, die unter einem gemeinsamen Namen zusammengefasst sind. Im Quelltext stehen dann an den Stellen, an denen eine solche Sequenz verwendet werden soll, nur die »Makro-Aufrufe«. Erst während der Assemblierung wird dann der Makro-Aufruf durch den »Textbaustein«, den Inhalt des Makros, ersetzt. Diesen Vorgang nennt man »Makroexpansion«.



Verwechseln Sie auf keinen Fall Unterprogramme wie Prozeduren oder Funktionen mit Makros! Bei Funktionen und Prozeduren existiert der Code nur ein einziges Mal im Modul. Er wird an verschiedenen Stellen durch CALL- oder JMP-Befehle angesprungen. Auch wenn oberflächlich betrachtet ein Makroaufruf ähnlich aussieht, verbirgt sich dahinter jedoch die Makroexpansion, was bedeutet, dass an jede Stelle des Makroaufrufs eine Kopie der im Makro verwendeten Direktiven und Instruktionen eingetragen wird!

Makros sind aus zwei Gründen sehr hilfreich! Zum einen erlauben sie eine sehr große Übersichtlichkeit des Quelltextes, vor allem, wenn sie in inclusion files verbannt werden. Zum anderen ermöglichen sie, richtig programmiert, eine Flexibilität in der Codegenerierung, die sehr oft nicht so einfach »von Hand« realisiert werden kann (vgl. Seite 607). Alleine die Möglichkeit der Nutzung von Makros sind ein Grund, in professionellen Programmen nicht Inline-Assembler, sondern die makrofähigen Stand-alone-Assembler zu nutzen!

Es gibt verschiedene Arten von Makros. Die einfachste Version ist ein Textmakro, bei dem ein Symbol eine Folge von Zeichen repräsentiert. Solche Textmakros kennt der Assembler in Form vordeklarierter Symbole wie @Date (MASM) bzw. ??Date (TASM). Auch Sie können mittels der Direktiven EQU oder TEXTEQU solche Textmakros deklarieren:

```
Hallo EQU <'Hi, Fans!'>
```

Diese Textmakros können dann in Datenallokationen verwendet werden. Sie werden bei der Assemblierung expandiert:

```
Msg   DB Hallo
Today DB @Date
```

Textmakros sind sog. Einzeilenmakros, weil ihre Deklaration innerhalb einer Zeile abgeschlossen wird (werden muss). Wenn Sie so wollen, sind ein weiteres Beispiel von Ein-Zeilen-Makros Makros, die numerische Werte zurückgeben. Auch sie sind in Form vordefinierter Symbole wie @CPU realisiert. Und auch Sie können solche Makros deklarieren. So ist jede Deklaration eines Symbols, das mittels EQU oder = deklariert wird, ein solches Makro, das an einer Stelle deklariert

```
EchtKoelnischWasser = 4711
```

und an einer anderen aufgerufen wird:

```
Deo DW EchtKoelnischWasser
CPU DB @CPU
```

Es sei jedoch nicht verheimlicht, dass solche Makros häufiger als »Symbole« bezeichnet werden. Der Streit hierüber erscheint mir jedoch mehr als akademisch!

Es gibt aber auch Mehrzeilenmakros. Diese besitzen grundsätzlich einen »Makrokörper« (*macro body*) der durch die Direktive MACRO eingeleitet und mit der Direktive ENDM abgeschlossen wird.

Solche Makros sind äußerst flexibel. Sie können ihnen Argumente, sog. »Dummy-Argumente« übergeben, die bei der Makroexpansion durch die Werte substituiert werden, die beim Makroaufruf angegeben werden (weshalb sie auch Dummy-Argumente heißen! Die eigentlichen Argumente sind ja die substituierenden Werte). Sie können Makros erstellen, die die Direktiven und Instruktionen im Makrokörper soundso oft repetieren (REPEAT) oder in Schleifen abarbeiten (WHILE, FOR, FORC). Und sie können Makros definieren, die die Argumente schrittweise abarbeiten und ihre Arbeit einstellen, wenn kein Argument mehr vorhanden ist.



Ein wichtiger Bestandteil von Makros sind Direktiven für die bedingte Assemblierung (vgl. Seite 662). Sie erlauben eine hohe Flexibilität von Makros und sind so gut wie unverzichtbar.

MACRO ENDM

Wie bereits einleitend gesagt: Makros werden mit den Schlüsselworten MACRO und ENDM deklariert:

```
Name MACRO [Parameter [, Parameter ...]]
   macro body
ENDM
```



Wie soll es auch anders sein: der IDEAL-Modus von TASM dreht die Reihenfolge um:

```
MACRO Name [Parameter [, Parameter ...]]
   macro body
ENDM
```

Name ist hierbei wiederum ein im Modul einzigartiger Name, der das Makro identifiziert. Die *Parameter* sind die bereits erwähnten Dummy-Parameter und haben die Form

```
DummyName[:DummyTyp]
```

Falls eine Zeile nicht ausreicht, um alle Parameter zu deklarieren, kann *Parameter* aus dem Zeilen-Fortsetzungszeichen »\« bestehen, was bewirkt, dass die nächste Zeile als Parameter-deklarierende Zeile betrachtet wird:

```
Dummy MACRO DummyName1:Typ1, DummyName2:Typ2 ,\
          DummyName3:Typ3
   macro body
ENDM
```

DummyName ist ein Platzhalter für die aktuellen Argumente bei der Makroexpansion im Makro. *DummyTyp* gibt an, welche Randbedingungen *DummyName* erfüllen muss. Als *DummyTyp* sind erlaubt:

`=<Default>` Falls bei der Makroexpansion *DummyName* durch ein »Null«-Argument (leeres Argument) substituiert wird, wird der Defaultstring *Default* verwendet. *Default* ist ein String, der in spitzen Klammern stehen muss.

`REQ` *REQ*ested erzeugt einen Fehler, falls ein leeres Argument zur Substitution von *DummyName* bei der Makroexpansion verwendet wird.

`VARARG` bewirkt, dass die bis hierhin nicht substituierten Argumente dem aktuellen Dummy-Parameter als Argumentenliste übergeben werden. Der Assembler trennt die Argumente der Liste automatisch mit Kommata und setzt sie in spitze Klammern. Dies kann z.B. mit Hilfe der Direktive FOR ausgewertet werden. *VARARG* darf nur als *DummyTyp* des letzten Parameters verwendet werden.

`REST` (nur bei TASM) wie *VARARG*, jedoch wird der Rest als einfacher Text ohne Kommata und spitze Klammern übergeben.

Beispiel Hier folgt ein Beispiel zur Nutzung der Parameterliste. Zunächst die Deklaration des Makros:

```
Test MACRO Dummy1:=<Hallo,>, Dummy2:REQ, Dummy3, \
     Dummy4:VARARG
        :
        :
ENDM
```

Und hier verschiedene Aufrufe des Makros und das Resultat die Parameter betreffend bei der Makroexpansion:

```
Test 'Hi,', 'Fan!', 2001, 'Dies', 'ist', 'eine', 'Liste'
     ; Dummy1 ← <Hi,>, Dummy2 ← <Fan>, Dummy3 ← 2001,
     ; Dummy4 ← <Dies>, <ist>, <eine>, <Liste>

Test '', 'Sie da!', 2002
     ; Dummy1 ← <Hallo,>, Dummy2 ← <Sie da>, Dummy3 ← 2002,
     ; Dummy4 ← <>

Test '', '', 2003, 4711
     ; FEHLER: Dummy2 muss einen Wert erhalten!

Test '', 'Sie', 2004, 4711
     ; Dummy1 ← <Hallo,>, Dummy2 ← <Sie >, Dummy3 ← 2004,
     ; Dummy4 ← <4711>
```

Als *macro body* schließlich werden alle gültigen und sinnvoll einzusetzenden Direktiven und Instruktionen bezeichnet.



Makro-Prozeduren und Makro-Funktionen (= Mehrzeilenmakros) können bis zu 40 Ebenen geschachtelt werden, Textmakros und Symbole (Einzeilenmakros) bis 20 Ebenen tief.



Es gibt spezielle Operatoren, die die Programmierung von Makros unterstützen. Diese Operatoren sind nur innerhalb von Makros erlaubt. Vergleiche hierzu Seite 706.

PURGE PURGE ist das Gegenteil zu MACRO und löscht das spezifizierte Makro. Makros, die mittels PURGE gelöscht wurden, sind danach nicht mehr verfügbar.

```
PURGE MakroName [, MakroName ...]
```

Sinn macht die Verwendung von PURGE, wenn ein Makro mit einem Schlüsselwort benannt wurde und nach Verwendung des Makros das Schlüsselwort wieder seine ursprüngliche Bedeutung haben soll.



LOCAL

Die Direktive LOCAL wurde bereits im Zusammenhang mit der Deklaration von Prozeduren besprochen (vgl. Seite 616). Sie kann auch in Makros verwendet werden, wenn sie an erster Stelle im Makrokörper genannt wird. Sie wird verwendet gemäß:

```
LOCAL DummySymbol [, DummySymbol ...]
```

Der Assembler generiert mit dieser Direktive automatisch Symbole mit modulweiter Gültigkeit.



Dies ist insbesondere aus dem Grund bedeutend, als Makros, anders als Prozeduren, in der Häufigkeit im Assemblat stehen, in der sie im Rahmen von Makroaufrufen expandiert wurden. Daher müssen alle lokal definierten Symbole einzigartig im Modul sein. Auf der anderen Seite kann jedoch bei der Deklaration von Makros nur ein Symbolname jeweils verwendet werden. Der Assembler muss also die Dummy-Symbole »übersetzen«. Da der Programmierer jedoch nur im Rahmen der Deklaration eines Makros Kenntnis über die jeweiligen Symbolnamen haben muss, genügt es, wenn er die selbst definierten Namen der lokalen Symbole kennt – die durch den Assembler generierten einzigartigen Namen benötigt er nie.



Daher kann der Programmierer unter MASM keinerlei Einfluss auf die Vergabe der generierten Namen nehmen – MASM erstellt Symbole mit Hexadezimalen Zahlen. TASM macht dies auch; jedoch kann der Programmierer hier immerhin vorgeben, ob der mittels LOCALS (vgl. Seite 624) einstellbare Präfix für lokale Symbole verwendet wird (LOCALS mit oder ohne Argument!) oder nicht (NOLOCALS). Im letzteren Fall werden Labels der Form »??XXXX« generiert, wobei XXXX eine einzigartige hexadezimale Zahl ist, andernfalls der Form »YYXXXX«, wobei »Y« der mittels LOCALS eingestellte Präfix (Default: @) ist.

EXITM

EXITM dient dem »vorzeitigen Ausstieg« aus dem Makro. Das bedeutet, dass EXITM bei der Makroexpansion, also der Substitution eines Makroaufrufes durch die Direktiven und Instruktionen des Makros, die Assemblierung des Makros abbricht und mit der Verarbeitung der Direktiven und Instruktionen fortfährt, die dem entsprechenden Makro-

aufruf folgen. EXITM wird üblicherweise im Rahmen der bedingten Assemblierung von Makros verwendet.



MASM, nicht aber TASM, erlaubt eine weitere Verwendung von EXITM mit erweiterter Syntax, wenn EXITM als letzte Direktive des Makrokörpers verwendet wird:

```
EXITM <Text>
```

Hier wird dem das Makro aufrufenden Teil der String *Text* zurückgegeben.

Makrolabels Innerhalb von Makros und REPEAT-, WHILE-, FOR- und FORC-Blöcken können Labels deklariert werden, die nur in diesen Makros oder Blöcken Gültigkeit haben und nur einem Zweck dienen: Ziel einer GOTO-Anweisung (siehe nächster Abschnitt) zu sein. Diese »Makrolabels« oder *tags* haben die Form

```
:Name
```

und sind nur im Rahmen der Makroexpansion von Bedeutung, weshalb sie danach nicht mehr existent sind.

GOTO Die Direktive GOTO veranlasst den Assembler, bei der Expansion von Makros zu einem Makrolabel zu springen und dort mit der Expansion fortzufahren. Die übersprungenen Direktiven und Instruktionen erscheinen nicht an der Stelle der Makroexpansion.

GOTO wird wie folgt verwendet:

```
GOTO TagName
```

REPEAT REPT REPEAT oder sein Synonym REPT leiten ein »Makro im Makro« ein. Das bedeutet, dass REPEAT nur innerhalb von Makros verwendet werden kann und selbst einen Makro-Block deklariert, der jedoch namenlos bleibt:

```
REPEAT Expression
   macro body
ENDM
```

Expression ist hierbei ein Ausdruck, der einen numerischen Wert ergeben muss. Dieser Wert wird im Rahmen des ersten Durchgangs evaluiert. Die zwischen REPEAT und ENDM eingeschlossenen Direktiven und Instruktionen werden dann *Expression*-mal wiederholt.

Mittels EXITM können REPEAT-Blöcke wie »normale« Makros frühzeitig verlassen werden.



Auch WHILE leitet analog zu REPEAT ein »Makro im Makro« ein: **WHILE**

```
WHILE Expression
   macro body
ENDM
```

Expression muss hier einen numerischen Wert ergeben, der als *false* (= 0) bzw. *true* (> 0) interpretiert werden kann, und wird vor jedem neuen Durchgang evaluiert. *Macro body* wird dann so lange wiederholt, bis Expression den Wert *false* liefert.

Mittels EXITM können WHILE-Blöcke wie »normale« Makros frühzeitig verlassen werden.



FOR leitet ein »Makro im Makro« ein, das wie REPEAT den Makrokörper eine bestimmte Anzahl mal wiederholt. Allerdings wird diese Anzahl nicht durch einen Ausdruck bestimmt, sondern durch die Zahl der in einer Parameterliste übergebenen Elemente: **FOR**

```
FOR DummyArgument, ArgumentListe
   macro body
ENDM
```

Hier nimmt *DummyArgument* bei jedem Durchgang das jeweils nächste Element in *ArgumentListe* an. Das Makro wird beendet, wenn es keine zu übergebenden Elemente in *ArgumentListe* mehr gibt.

IRP ist im MASM-Modus von TASM ein Synonym für FOR. Im TASM-Modus ist FOR nicht implementiert, weshalb hier IRP verwendet werden muss. *IRP*



Ein Verwandter von FOR ist FORC, das anstelle einer Argumentenliste einen String erhält und so lange ausgeführt wird, wie noch Zeichen im String sind: **FORC**

```
FORC DummyArgument, String
   macro body
ENDM
```

DummyArgument erhält hier konsekutiv bei jedem Durchgang das jeweils nächste Zeichen in *String*. ACHUNG: Leerzeichen sind auch Zeichen!

IRPC



IRPC ist im MASM-Modus von TASM ein Synonym für FORC. Im TASM-Modus ist FORC nicht implementiert, weshalb hier IRPC verwendet werden muss.

3.2.11 Bedingte Assemblierung

Es gibt Situationen, da wird die Assemblierung von bestimmten Quelltextteilen von einer Bedingung abhängig gemacht. So könnte z.B. die gesamte Assemblierung des Quelltextes davon abhängig gemacht werden, dass die CPU, auf der die Assemblierung erfolgt, einen bestimmten Befehlssatz beherrscht. Andernfalls könnte eine Fehlermeldung generiert werden. So ist z.B. die Assemblierung von SIMD-Befehlen nur dann sinnvoll, wenn die CPU die MMX-, SSE-, SSE2- und/oder 3DNow!-Befehle kennt.

Diese Art der Assemblierung nennt man bedingte Assemblierung, da sie nur dann erfolgt, wenn eine Bedingung erfüllt ist. Und zur Durchführung von bedingter Assemblierung gibt es spezielle Direktiven, die das ermöglichen. Diese Direktiven werden in diesem Kapitel besprochen.



Verwechseln Sie diese Direktiven, die alle *keinen* Punkt vor dem Direktivennamen haben, nicht mit den punktierten Zwillingen, die bei der Programmierung von Schleifen und IF-THEN-ELSE-Konstrukten zum Einsatz kommen (vgl. Seite 652)! Während bei den hier besprochenen Direktiven das Ergebnis der Compilierung abhängig von der Erfüllung einer Bedingung zum Zeitpunkt der Assemblierung ist (= bedingte Assemblierung), generieren die punktierten Direktiven Code, der den entsprechenden Konstrukten aus Hochsprachen entspricht. Die Bedingungen, die diese Konstrukte prüfen, werden somit erst zur Laufzeit des assemblierten Moduls geprüft und entsprechend reagiert.

IF
ELSEIF
ELSE
ENDIF

Der einfachste Fall bedingter Assemblierung ist der, dass die zu assemblierenden Instruktionen zwischen IF und ENDIF stehen. Hierbei definiert der hinter dem Schlüsselwort IF stehende Ausdruck die Bedingung, die erfüllt sein muss, damit die Assemblierung der bis zu ELSEIF, ELSE oder ENDIF stehenden Statements erfolgt. Analoges gilt für ELSEIF. Der zwischen ELSE und ENDIF stehende Teil wird nur dann assembliert, wenn bis an diese Stelle noch keine Bedingung erfüllt wurde.

Das bedeutet: Im Assemblat steht nur ein einzelner Block von Anweisungen: Entweder der zwischen IF und ELSE (ELSEIF, ENDIF) oder der zwischen ELSEIF (so vorhanden) und ELSE (ELSEIF, ENDIF) oder der zwischen ELSE (so vorhanden) und ENDIF – oder eben keiner (falls kein ELSE verwendet wurde). Niemals aber mehr als einer der Blöcke!

Verwendung

Formal werden diese Direktiven wie folgt verwendet:

```
IF Expression
   Statement(s)
[ELSEIF Expression
   Statement(s)]
:
:
[ELSE
   Statement(s)]
ENDIF
```

Hierbei ist *Expression* ein Ausdruck, der als Ergebnis »true« oder »false« ergeben muss. *True* heißt in diesem Zusammenhang: nicht Null, *false*: Null. Das bedeutet, Expression muss einen numerischen Wert zurückgeben.

IFcc
ELSEIFcc

Nicht immer kann Expression jedoch diesen numerischen Wert zurückgeben, da es z.B. ein Symbol referenziert, dessen Existenz oder Schreibweise überprüft werden soll, oder zwei Argumente verglichen werden müssen. Für solche Fälle gibt es mit IFcc- und ELSEIFcc-Variationen der Direktiven IF und ELSEIF, die diesen geänderten Voraussetzungen Rechnung tragen. Sie sind in Tabelle 3.15 aufgelistet.

cc	Bedeutung	Direktive	Gegenspieler	Expression true, wenn
-	-	(ELSE)IF	(ELSE)IFE	numerisch > 0
E	equal	(ELSE)IFE	(ELSE)IF	numerisch = 0
B	blank	(ELSE)IFB	(ELSE)IFNB	leer (uninitialisiert)
NB	not blank	(ELSE)IFNB	(ELSE)IFB	nicht leer (initialisiert)
DEF	defined	(ELSE)IFDEF	(ELSE)IFNDEF	Symbol definiert
NDEF	not defined	(ELSE)IFNDEF	(ELSE)IFDEF	Symbol nicht definiert
DIF	different	(ELSE)IFDIF	(ELSE)IFIDN	Argumente verschieden
DIFI	different	(ELSE)IFDIFI	(ELSE)IFIDNI	wie DIF, jedoch case insensitive
IDN	identical	(ELSE)IFIDN	(ELSE)IFDIF	Argumente gleich
IDNI	identical	(ELSE)IFIDNI	(ELSE)IFDIFI	wie IDN, jedoch case insensitive

Tabelle 3.15: Variationen der Direktiven IF und ELSEIF für die bedingte Assemblierung

Verwendung Formal werden diese Direktiven wie folgt eingesetzt:

```
[ELSE]IF[E]    Expression
[ELSE]IF[N]B   Text
[ELSE]IF[N]DEF SymbolExpression
[ELSE]IFDIF[I] Text1, Text2
[ELSE]IFIDN[I] Text1, Text2
```

Expression ist wie üblich ein Ausdruck, der einen numerischen Wert ergeben muss. *SymbolExpression* ist entweder der Name eines bereits deklarierten Symbols oder ein Ausdruck, bestehend aus Symbolnamen, den Booleschen Operatoren AND, OR und NOT und ggf. (runden) Klammern. Und *Text* ist entweder ein in eckige Klammern (»< >«) eingeschlossener String (Zeichenfolge), ein numerischer Ausdruck, dem ein »%«-Operator vorangeht oder der Name eines Textmakros.



TASM erweitert die Möglichkeiten um zwei weitere IF-Direktiven: IF1 und IF2. Da TASM bei der Assemblierung mehr als einen Assembler-Durchgang (»pass«) erlaubt, kann mit Hilfe der Direktiven IF1 und IF2 eine Assemblierung vom aktuellen Assembler-Durchgang abhängig gemacht werden. So werden auf IF1 folgende Statements nur assembliert, wenn sich TASM in Durchgang 1 befindet und auf IF2 folgende Statements nur im Durchgang 2.

.ERR
.ERRcc

So wie es sinnvoll sein kann, die Assemblierung bestimmter Quelltext-Teile an die Erfüllung einer Bedingung zu knüpfen, kann es ebenfalls sinnvoll sein, in bestimmten Situationen Fehlermeldungen zu generieren. Hierbei unterscheidet man zwei Arten von Fehlermeldungen:

- Unbedingte Fehlermeldungen und
- Bedingte Fehlermeldungen.

Die Direktive .ERR ist eine unbedingt Fehler erzeugende Direktive. Sie erzeugt in jedem Fall einen Fehler, was einen Abbruch der Assemblierung zur Folge hat, und gibt optional eine Fehlermeldung aus.

Analog den Direktiven für die bedingte Assemblierung gibt es Direktiven für die bedingte Fehlergeneration. Sie sind ähnlich den Direktiven zur bedingten Assemblierung aufgebaut: .ERRcc



Es ist unglücklich, ja, aber leider nicht zu ändern! Die Direktiven zur bedingten Assemblierung haben alle *keinen* führenden Punkt, da die Punkt-Varianten für die Generierung von Instruktionen für die Realisierung von Schleifen u. ä. reserviert sind. Die Direktiven für die be-

dingte Fehlergeneration dagegen beginnen alle *mit* einem Punkt. Die punktlosen Varianten sind auf TASM beschränkt!

Tabelle 3.16 listet die verfügbaren Direktiven für die Bedingte Fehlergeneration auf.

cc	Bedeutung	Direktive	Gegenspieler	Fehler, wenn
E	equal	.ERRE	.ERRNZ	Bedingung wahr
NZ	not equal	.ERRNZ	.ERRE	Bedingung falsch (nicht wahr)
B	blank	.ERRB	.ERRNB	Argument leer (uninitialisiert)
NB	not blank	.ERRNB	.ERRB	Argument nicht leer (initialisiert)
DEF	defined	.ERRDEF	.ERRNDEF	Symbol definiert
NDEF	not defined	.ERRNDEF	.ERRDEF	Symbol nicht definiert
DIF	different	.ERRDIF	.ERRIDN	Argumente verschieden
DIFI	different	.ERRDIFI	.ERRIDNI	wie DIF, jedoch case insensitive
IDN	identical	.ERRIDN	.ERRDIF	Argumente gleich
IDNI	identical	.ERRIDNI	.ERRDIFI	wie IDN, jedoch case insensitive

Tabelle 3.16: Variationen der Direktiven .ERR für die Bedingte Fehlergenerierung

Formal werden diese Direktiven wie folgt eingesetzt: *Verwendung*

```
.ERRE      Expression [, Message]
.ERRNZ     Expression [, Message]
.ERR[N]B   Text [, Message]
.ERR[N]DEF SymbolExpression [, Message]
.ERRDIF[I] Text1, Text2 [, Message]
.ERRIDN[I] Text1, Text2 [, Message]
```

Expression ist wie üblich ein Ausdruck, der einen numerischen Wert für true (> 0) oder false (= 0) ergeben muss. *SymbolExpression* ist entweder der Name eines bereits deklarierten Symbols oder ein Ausdruck, bestehend aus Symbolnamen, den Booleschen Operatoren AND, OR und NOT und ggf. (runden) Klammern. Und *Text* ist entweder ein in eckige Klammern (»< >«) eingeschlossener String (Zeichenfolge), ein numerischer Ausdruck, dem ein »%«-Operator vorangeht, oder der Name eines Textmakros.

Im Unterschied zu IF/IFE ist nicht .ERR der Gegenspieler zu .ERRE, sondern .ERRNZ, da .ERR unbedingt (d. h. ohne jede Bedingung) einen Fehler generiert!





TASM bietet auch bei den .ERR-Direktiven Synonyme, die sowohl im MASM-Modus als auch im IDEAL-Modus eingesetzt werden können. So ist .ERR = ERR (bitte beachten Sie die führenden Punkte!), .ERRB = ERRIFB, .ERRDEF = ERRIFDEF, .ERRDIF = ERRIFDIF, .ERRE = ERRIFE, .ERRIDN = ERRIFIDN, .ERRIDN = ERRIFIDN, .ERRIDNI = ERRIFIDNI, .ERRNB = ERRIFNB, .ERRNDEF = ERRIFNDEF und .ERRNZ = ERRIF.

Ferner erweitert TASM die Möglichkeiten um zwei weitere .ERRcc-Direktiven: .ERR1 und .ERR2. Da TASM bei der Assemblierung mehr als einen Assembler-Durchgang (»pass«) erlaubt, kann mit Hilfe der Direktiven .ERR1 und .ERR2 eine Fehlergenerierung vom aktuellen Assembler-Durchgang abhängig gemacht werden. So werden mit .ERR1 nur Fehler generiert, wenn sich TASM in Durchgang 1 befindet und mit .ERR2 nur im Durchgang 2.

3.2.12 Direktiven zur Steuerung von Listings

Listings sind, vor allem bei längeren Assemblerquelltexten, eine angenehme und hilfreiche Angelegenheit. So listet der Assembler in Listings nicht nur den Quelltext auf, sondern protokolliert quasi seine Aktivitäten. Listings können auch Symboltabellen enthalten, die einem den Ort der Deklaration und Benutzung von Symbolen nennen (»cross-reference tables«) oder andere hilfreiche Informationen. Und, was besonders wichtig ist: Listings können Informationen auch »verstecken«. So erscheinen in Listings üblicherweise nur die Instruktionen, die auch tatsächlich in den OBJ-File aufgenommen wurden. Wichtig ist dies bei der bedingten Assemblierung: Teile, die aufgrund einer Bedingung nicht assembliert wurden (»false conditionals«), erscheinen nicht (zwangsweise) in Listings.

Weil Listings also eine wichtige Funktion haben und Quelle vieler Informationen sein können, die der Assemblerprogrammierer nicht nur bei der Fehlersuche benötigt, gibt es Direktiven, die Einfluss auf das nehmen können, was in Listings erscheinen soll – und die auf diese Weise Listings je nach Zweck entweder übersichtlicher machen (wenn z.B. Code, der nicht assembliert wurde, auch nicht erscheint!) oder aber auch das letzte Quäntchen Information ans Tageslicht bringen (wenn z.B. auch die Steuerdirektiven für Listings im Listing erscheinen).

Listings bestehen aus zwei Teilen: dem Listing der assemblierten Instruktionen (»source listing«) und den sich anschließenden Tabellen

(Makro-Tabelle, Tabelle der structures, records und unions, Typen-Tabelle, Segment- und Gruppen-Tabelle, Liste der Prozeduren mit ihren Parametern und lokalen Variablen und die Symbol-Tabelle).

Eine Listing-Seite hat folgenden allgemeinen Aufbau:

```
Header
[Titel]
[Untertitel]
```

Der *Header* besteht in der Angabe des Programms, das das Listing erzeugt hat (also MASM oder TASM), dessen Version, Datum und Uhrzeit der Erstellung, Name des Quelltext-Files und Seitenzahl im Format »*Kapitelnummer - Seitennummer*«. *Titel* und *Untertitel* sind optional und können vom Benutzer frei definiert werden, Kapitelnummer und Seitenzahl steuert der Assembler. Anfangswerte sind »1 - 1«.

Diesem Seitenkopf schließen sich im Falle des source listing die einzelnen Zeilen an, die der Assembler aus den Quelltextzeilen generiert, und zwar in der Form

```
<Tiefe> <Zeilennummer> <Offset> <Opcode> <Quelltext>
```

Tiefe gibt hierbei die Verschachtelungstiefe für Text aus *inclusion files* und Makros an, *Zeilennummer* ist eine optionale Zeilennummer im Listing. Sie ist hilfreich, wenn z.B. *cross-reference tables* generiert werden, in denen die Zeilen der Deklaration und Nutzung von Symbolen verzeichnet sind. Der *Offset* ist der Offset zum Segmentbeginn, den das Datum oder die Instruktion besitzt, die im folgenden *Opcode*, der Bytefolge, die der Assembler aus dem Quelltext für die dazugehörige Instruktion/Datum erzeugt hat, codiert ist. *Quelltext* ist die dazugehörige Zeile im Quelltext.

Bitte beachten Sie, dass nicht alle Assembler dieses Zeilenformat benutzen. So ist das Feld *Tiefe* eine TASM-Spezialität und auch *Zeilennummer* wird nicht immer angegeben. *Offset*, *Opcode* und *Quelltext* sind dagegen Standard.



TITLE
SUBTITLE

Mit TITLE und SUBTITLE kann der Assembler angewiesen werden, jedes Blatt eines Listings mit einem Titel und einem Untertitle zu versehen. TITLE darf unter MASM pro Modul nur einmal angegeben werden, bei TASM wie SUBTITLE beliebig oft. TITLE wird auf jeder Seite unmittelbar nach dem Header linksbündig ausgegeben, SUBTITLE in der folgenden Zeile.

Verwendung TITLE und SUBTITLE werden wie folgt verwendet:

```
TITLE Text
SUBTITLE Text
```

wobei Text jede beliebige Folge von Zeichen sein kann. Bei MASM ist sie auf 60 Zeichen begrenzt.



TASM stellt drei Synonyme zur Verfügung, die sowohl im MASM- als auch im IDEAL-Modus nutzbar sind: %TITLE und %SUBTTL und SUBTTL. SUBTTL ist ein »echtes« Synonym zu SUBTITLE und wird analog verwendet, die Nutzung von %TITLE und %SUBTTL dagegen erfolgt zwar analog zu TITLE und SUBTITLE, jedoch muss der Text in Anführungszeichen eingeschlossen sein:

```
%TITLE "Text"
%SUBTTL "Text"
```

PAGE Mit PAGE kann zum einen ein Seitenvorschub durchgeführt werden, wenn kein Argument angegeben wird. Werden dagegen Argumente angegeben, so definiert PAGE die Anzahl der Zeilen (inkl. Header und Titel/Untertitel) und Spalten einer Seite:

Verwendung

```
PAGE [Zeilen] [, Spalten]
```

Die erlaubte Zahl der Zeilen liegt zwischen 10 und 255 Zeilen, ein Wert von 0 ist erlaubt und bedeutet, dass der Assembler keinen Seitenumbruch generiert. Die erlaubte Zahl an Spalten liegt zwischen 60 und 255 Zeichen, ein Wert von 0 ist ebenfalls erlaubt und bedeutet, dass der Assembler keinen Zeilenumbruch generiert und die Zeilen evtl. abschneidet. Defaultwert für beide Parameter ist 0.

Soll nur die Spaltenzahl geändert werden, darf das Komma nicht vergessen werden:

```
PAGE , 80
```



PAGE kann auch mit einem »+« als Argument aufgerufen werden:

```
PAGE +
```

Dies führt wie PAGE ohne Argument zu einem Seitenumbruch, allerdings wird auch die Nummer des aktuellen Kapitels um 1 erhöht und die Seitenzahl auf 1 gesetzt. Somit ist diese Nutzung der Direktive nur sinnvoll, wenn Listings für die Druckerausgabe generiert werden sollen, da sich die Seitennummerierung aus der Kapitel- und der aktuellen Seitennummer zusammensetzt.

TASM geht auch hier wieder aufgrund des IDEAL-Modus eigene Wege, kennt aber aus Kompatibilitätsgründen auch PAGE. Unter TASM kann im MASM- und IDEAL-Modus %PAGESIZE verwendet werden, das mit einer Ausnahme absolut identisch zu PAGE funktioniert. Die Ausnahme ist, dass die Verwendung von %PAGESIZE ohne Argumente nicht zu einem Seitenumbruch führt. Dies muss durch %NEWPAGE ausgelöst werden. %NEWPAGE hat keine Argumente.



%DEPTH
%LINUM
%PCNT
%BIN
%TRUNC
%NOTRUNC
%TEXT
%TABSIZE

Unter TASM kann das Format einer Zeile im *source listing* verändert werden. Mittels %DEPTH kann die Größe für *Tiefe* eingestellt werden, mit %LINUM die Größe des Feldes für die *Zeilennummer*. %PCNT stellt die Breite des Feldes *Offset* ein, %BIN die für den *Opcode*. Falls der Opcode nicht in das entsprechende Feld mit der spezifizierten Breite passt, wird er abgeschnitten (%TRUNC) oder in die nächste Zeile umgebrochen (%NOTRUNC). %Text definiert die Breite der Spalte, die für *Quelltext* verwendet werden soll (vgl. Seite 667). Mit %TABSIZE schließlich kann eingestellt werden, wie viele Leerzeichen zwischen zwei Tabulatoren liegen sollen.



Verwendung

Die Direktiven werden formal wie folgt verwendet:

```
%DEPTH Breite
%LINUM Breite
%PCNT Breite
%BIN Breite
%[NO]TRUNC
%TEXT Breite
%TABSIZE Breite
```

`Breite` gibt hierbei die Anzahl von Stellen an, die dem Feld zur Verfügung gestellt werden soll.

.LIST
.NOLIST
.LISTIF
.NOLISTIF
.TFCOND
.LISTMACRO
.LISTMACROALL
.NOLISTMACRO
.LISTALL
.CREF
.NOCREF

Handelte es sich bei den bislang beschriebenen Direktiven um solche, die das Erscheinungsbild eines Listings steuern, so folgen nun solche, die den Inhalt beeinflussen.

.LIST ist die Defaulteinstellung und bewirkt, dass alle Quellzeilen des Moduls in das Listing aufgenommen werden, sobald ein Listingfile erstellt wird. Sollen die Quelltextzeilen dagegen nicht in das Listing übernommen werden, so kann das durch die Angabe von .NOLIST erreicht werden.

Im Rahmen der bedingten Assemblierung spielen .LISTIF und .NOLISTIF eine Rolle. Wird .LISTIF angegeben, was auch der Defaultwert ist, so werden auch die Blöcke in das Listing eingebunden, die auf-

grund einer nicht erfüllten Bedingung nicht assembliert wurden. bei .NOLISTIF unterbleibt dies und sorgt damit für kürzere und übersichtlichere Listings, die nur das enthalten, was auch tatsächlich assembliert wurde. Zwischen .LISTIF und .NOLISTIF kann mittels .TFCOND umgeschaltet werden.

.LISTMACRO, .LISTMACROALL und .NOLISTMACRO steuern, welche Makro-Informationen bei der Expansion von Makros in das Listing aufgenommen werden: Bei .LISTMACRO werden nur Statements übernommen, die Code oder Daten erzeugen, nicht aber Kommentare, Symbole oder Segmentdefinitionen. .LISTMACROALL dagegen übernimmt alle Statements. Durch .NOLISTMACRO wird verhindert, dass Makrostatements überhaupt im Listing erscheinen.

.LISTALL ist eine Kombination der Direktiven .LIST, .LISTIF und .LISTMACROALL.

Jedes Listing besitzt eine Symboltabelle, in der Informationen zu den im Modul deklarierten Symbolen stehen. Dies schließt allerdings die Angabe des Ortes ihrer Deklaration und ihrer Benutzung nicht mit ein! Es kann jedoch sehr hilfreich sein, vor allem bei längeren Listings, diese Informationen zur Hand zu haben. Mit .CREF ist dies möglich: Diese Direktive, die standardmäßig eingestellt ist, erzeugt am Ende des Listings eine Tabelle mit den gewünschten Informationen, die sog. *cross-reference list*. Soll sie nicht erstellt werden, kann dies mit .NOCREF erreicht werden.

.NOCREF hat optionale Argumente:

```
.NOCREF [Symbol [, Symbol ...]]
```

Werden bestimmte Symbole als Argumente angegeben, so werden ausschließlich diese vom Eintrag in die Cross-Reference-Liste ausgeschlossen, alle anderen nicht.



Aus Gründen der Abwärtskompatibilität gibt es Synonyme für die angegebenen Direktiven. So ist .XLIST identisch mit .NOLIST, .LFCOND mit .LISTIF, .SFCOND mit .NOLISTIF, .XALL mit .LISTMACRO, .LALL mit .LISTMACROALL, .SALL mit .NOLISTMACRO, .XCREF für .NOCREF .



Der IDEAL-Modus von TASM erfordert auch bei den Listing-Direktiven die Existenz von Synonymen. Und so gibt es sie auch: %LIST ist mit .LIST identisch, %NOLIST mit .NOLIST. Zu .LISTIF und .NoLISTIF hei-

ßen die TASM-Pendants %CONDS und %NOCONDS, zu .LISTMACROALL und .NOLISTMACRO %MACS und %NOMACS, zu .CREF und .NOCREF %CREF und %NOCREF.

Um ein wenig Licht in den Synonym-Dschungel zu bekommen, möge Tabelle 3.17 helfen.

MASM-Direktive	MASM-Synonym und MASM-Modus von TASM	TASM-Synonym im IDEAL- und MASM-Modus
.LIST	.LIST	%LIST
.NOLIST	.XLIST	%NOLIST
.LISTIF	.LFCOND	%CONDS
.NOLISTIF	.SFCOND	%NOCONDS
.TFCOND	.TFCOND	
.LISTMACRO	.XALL	
.LISTMACROALL	.LALL	%MACS
.NOLISTMACRO	.SALL	%NOMACS
.LISTALL	.LISTALL	
.CREF	.CREF	%CREF
.NOCREF	.XCREF	%NOCREF

Tabelle 3.17: Direktiven zur Steuerung von Listings, ihre Synonyme in älteren MASM-Versionen und ihre Entsprechungen im MASM- und IDEAL-Modus von TASM

%CREFALL
%CREFREF
%CREFUREF
%CLTS
%NOCLTS
%SYMS
%NOSYMS

TASM erweitert die Möglichkeiten der Listingsteuerung noch weiter. So gibt es mit %CREFALL, %CREFREF und %CREFUREF drei Direktiven, die eine Feinabstimmung ermöglichen, welche Symbole in die cross-reference table aufgenommen werden: %CREFREF unterdrückt die Aufnahme von unreferenzierten Symbolen, also solchen, die zwar deklariert, aber niemals genutzt wurden. %CREFUREF dagegen unterdrückt die Aufnahme von Symbolen, die deklariert und benutzt wurden, und ist somit der Gegenspieler zu %CREFREF. Sollen beide Symbolvarianten aufgenommen werden, kommt %CREFALL zum Tragen.



Auch die Direktiven, die für die Steuerung des Listings zuständig sind, wie z.B. .LIST, %CREFALL oder %CLTS, können entweder in das Listing aufgenommen (%CLTS) oder davon ausgeschlossen (%NOCLTS) werden.

Und selbst die am Schluss eines Listings angefügte Symboltabelle kann unterdrückt (%NOSYMS) oder generiert (%SYMS) werden.

%INCL
%NOINCL



Unter %NOINCL werden Quelltextzeilen aus inclusion files nicht in das Listing übernommen. Sinnvoll ist dies z.B., wenn solche inclusion files die Funktion von Header-Dateien unter C/C++ haben und lediglich eine Reihe von Symbolen deklarieren, die in anderen Quelltext-Dateien genutzt werden sollen. Wenn deren Deklaration nicht spannend und wichtig ist, kann auf die entsprechenden Zeilen im Listing verzichtet werden, was es überschaubarer macht. Sollen dagegen auch die Quelltexte aus inclusion files in das Listing übernommen werden, muss %INCL verwendet werden. Dies ist auch die Standardeinstellung.

3.2.13 Direktiven zur Anwahl des Befehlssatzes

In Tabelle 3.18 sind die Direktiven angegeben, mit denen der Befehlssatz angewählt werden kann, den der Assembler nutzen soll.

Direktive	verfügbarer Befehlssatz											
	8086	80186	80286	80386	80486	Pentium	8087	80287	80387	FPU	protected mode	privilegiert
.8086	X	O	O	O	O	O	X	O	O	O	X	O
.8087							X	O	O	O		
.186	X	X	O	O	O	O	X	O	O	O	X	O
.286	X	X	X	O	O	O	X	X	O	O	X	O
.286P	X	X	X	O	O	O	X	X	O	O	X	X
.287		X						X	O	O		
.386	X	X	X	X	O	O	X	X	X	O	X	O
.386P	X	X	X	X	O	O	X	X	X	O	X	X
.387							X	X	X	O		
.486	X	X	X	X	X	O	X	X	X	X	X	O
.486P	X	X	X	X	X	O	X	X	X	X	X	X
.586	X	X	X	X	X	X	X	X	X	X	X	O
.586P	X	X	X	X	X	X	X	X	X	X	X	X
.NO87							O	O	O	O		

X bedeutet, dass der entsprechende Befehlssatz durch die Direktive aktiviert wird, bei O wird er deaktiviert. Ein freies Feld bedeutet, dass keine Änderung an der aktuellen Einstellung vorgenommen wird.

Tabelle 3.18: Direktiven zur Anwahl des Befehlssatzes und ihre Auswirkungen auf den verfügbaren Befehlssatz

So kann durch Nutzung der Direktive .8086 z.B. absolute Kompatibilität zum 80286-Prozessor hergestellt werden, da in diesem Fall der Assembler nur den 8086/87-Befehlssatz als gültige Eingabe akzeptiert. Bitte beachten Sie: Die Angabe einer Direktive, die einen bestimmten CPU-Befehlssatz »freischaltet«, aktiviert auch den dazugehörigen Satz an NPX- bzw. FPU-Befehlen. Sollen diese ausgeschlossen werden, so muss die Direktive .NO87 folgen. Mittels .x87 kann dann der Befehlssatz der NPX/FPU wieder dazugeschaltet werden.

Ab dem 80286 gibt es ja den protected mode und damit privilegierte Befehle, die nur dem Betriebssystem im protected mode zugänglich gemacht werden sollen. Daher gibt es ab diesem Prozessor jeweils zwei Direktiven: eine Direktive, die den Befehlssatz ohne privilegierte Befehle freischaltet und für den »normalen« Programmierer gedacht ist, und eine, die die privilegierten Befehle ebenfalls verfügbar macht, und für Betriebssystemkonstrukteure »reserviert« ist. Letztere ist gekennzeichnet durch ein »P«, das an den Namen der Direktive angehängt wird.



Auch bei diesen Direktiven erfordert der IDEAL-Modus von TASM Synonyme. Sie beginnen statt mit einem Punkt mit einem »P« und unterscheiden sich von den ».«-Varianten des MASM-Modus derart, dass es nicht nur die Unterscheidung »nicht-privilegiert« und »privilegiert« gibt, sondern auch noch die Einschränkung auf den real mode. Diese Direktiven zeichnen sich durch ein angehängtes »N« aus. Tabelle 3.19 listet sie auf.

	verfügbarer Befehlssatz											
Direktive	**8086**	**80186**	**80286**	**80386**	**80486**	**Pentium**	**8087**	**80287**	**80387**	**FPU**	**protected mode**	**privilegiert**
P8086	X	O	O	O	O	O	X	O	O	O	O	O
P8087							X	O	O	O	O	
P186	X	X	O	O	O	O	X	O	O	O	O	O
P286	X	X	X	O	O	O	X	X	O	O	X	O
P286N	X	X	X	O	O	O	X	X	O	O	O	O
P286P	X	X	X	O	O	O	X	X	O	O	X	X
P287		X						X	O	O		

Tabelle 3.19: TASM-spezifische Direktiven zur Anwahl des Befehlssatzes und ihre Auswirkungen auf den verfügbaren Befehlssatz

Direktive	verfügbarer Befehlssatz											
	8086	80186	80286	80386	80486	Pentium	8087	80287	80387	FPU	protected mode	privilegiert
P386	X	X	X	X	O	O	X	X	X	O	X	O
P386N	X	X	X	X	O	O	X	X	X	O	O	O
P386P	X	X	X	X	O	O	X	X	X	O	X	X
P387							X	X	X	O		
P486	X	X	X	X	X	O	X	X	X	X	X	O
P486N	X	X	X	X	X	O	X	X	X	X	O	O
P486P	X	X	X	X	X	O	X	X	X	X	X	X
P586	X	X	X	X	X	X	X	X	X	X	X	O
P586N	X	X	X	X	X	X	X	X	X	X	O	O
P586P	X	X	X	X	X	X	X	X	X	X	X	X
PNO87							O	O	O	O		
.286C	X	X	X	O	O	O	X	X	O	O	O	O
.386C	X	X	X	X	O	O	X	X	X	O	O	O
.486C	X	X	X	X	X	O	X	X	X	X	O	O
.487							X	X	X	X		
.586C	X	X	X	X	X	X	X	X	X	X	O	O
.587							X	X	X	X		

X bedeutet, dass der entsprechende Befehlssatz durch die Direktive aktiviert wird, bei O wird er deaktiviert. Ein freies Feld bedeutet, dass keine Änderung an der aktuellen Einstellung vorgenommen wird.

Tabelle 3.19: TASM-spezifische Direktiven zur Anwahl des Befehlssatzes und ihre Auswirkungen auf den verfügbaren Befehlssatz (Forts.)

Doch Borland hat die Liste der Direktiven zur Anwahl des Befehlssatzes darüber hinaus noch erweitert. So gibt es auch ».«-Varianten, die der MASM nicht kennt. Analog den »PxxxN«-Direktiven, die auf den real mode beschränkt sind, gibt es Direktiven im Punkt-Format. Nur ist der Suffix hier »C« und nicht »N«: .xxxC. Und mit .487 und .587 gibt es auch zwei Direktiven, die den Befehlssatz der FPUs des 80486 und Pentiums explizit verfügbar machen.



Mit den Versionen ab MASM 6.11, die parallel zu den entsprechenden Prozessorgenerationen veröffentlicht wurden, haben sich auch einige MASM-spezifische Direktiven ergeben, die in Tabelle 3.20 dargestellt sind.

Direktive	verfügbarer Befehlssatz															
	8086	80186	80286	80386	80486	Pentium	P6-Familie	8087	80287	80387	FPU	MMX	3DNow	XMM	protected mode	privilegiert
.MMX												X				
.686	X	X	X	X	X	X	X	X	X	X	X				X	O
.686P	X	X	X	X	X	X	X	X	X	X	X				X	X
.K3D													X			
.XMM														X		

X bedeutet, dass der entsprechende Befehlssatz durch die Direktive aktiviert wird, bei O wird er deaktiviert. Ein freies Feld bedeutet, dass keine Änderung an der aktuellen Einstellung vorgenommen wird.

Tabelle 3.20: MASM-spezifische Direktiven zur Anwahl des Befehlssatzes und ihre Auswirkungen auf den verfügbaren Befehlssatz

Die Direktiven .686 und .686P wurden mit MASM 6.12 eingeführt, .K3D mit MASM 6.13 und .XMM mit MASM 6.14.



Es ist zu erwarten, dass der mit C++-Builder 6.0 von Borland ausgelieferte TASM diese Direktiven ebenfalls kennen wird, da C++ sicherlich nicht hinter Delphi zurückbleiben soll, der Integrierte Assembler von Delphi in Version 6.0 jedoch bereits die SSE-Befehle und damit OWORDs kennt. Leider kann ich das zum Zeitpunkt der Manuskripterstellung nicht verifizieren, da Borland noch keine Beta-Version des C++-Builders 6.0 zur Verfügung stellt.



3.2.14 Interaktion mit dem Programmierer

Hilfreich für Dokumentation und Verständnis sind Kommentare im Quelltext, die der Assembler nicht berücksichtigt. Solche Kommentare können auf zwei Arten erzeugt werden:

- durch die Direktive ;
- durch die Direktive COMMENT

Ferner kann es sinnvoll sein, während der Assemblierung Meldungen ausgeben zu können, die z.B. zum Verlauf der Assemblierung Auskunft geben. Dies erfolgt mit der Direktive ECHO, die %OUT ein aufgrund Abwärtskompatibilität erforderliches und mit DISPLAY aufgrund des IDEAL-Modus von TASM ein TASM-spezifisches Synonym hat.

Schließlich können neben Fehlermeldungen (vgl. Seite 664) auch Warnungen ausgesprochen werden, die den Assemblerlauf nicht abbrechen. Hierzu dient WARN bzw. NOWARN.

; Der Assembler betrachtet jedes Zeichen, das auf ein »;« folgt, als Kommentar und berücksichtigt es nicht in der Assemblierung. Die Wirkung ist auf die entsprechende Zeile beschränkt. »;« ist somit eine Direktive, die dem Assembler sagt: »Ab hier bis zum Zeilenende: off limits!«

COMMENT Vor allem bei längeren Kommentaren ist es mühlselig, jede Kommentarzeile mit einem »;« zu beginnen. Daher kann mit der Direktive COMMENT zweierlei durchgeführt werden:

- Definition eines Zeichens, das als Ende-Marke des Kommentarbereiches verwendet wird, und
- Definition des Anfangs des Kommentarbereiches.

Verwendung COMMENT wird formal wie folgt verwendet:

```
COMMENT EndeZeichen [Kommentartext]
[Kommentartext]
[Kommentartext] EndeZeichen [Quelltext]
```

Endezeichen ist das Zeichen, das den Kommentarbereich klammert. Verwenden Sie für EndeZeichen nur Zeichen, die Sie nicht für Ihren Kommentar benutzen! Häufig zum Einsatz kommen daher solche Zeichen wie »~«, »§« oder »|«, aber auch »*« oder »#«.

Beispiel

```
COMMENT # ------------------------------------------------
Zwischen den beiden Endzeichen wird jedes Zeichen als Kommen-
tarzeichen interpretiert. So können hier selbst Anweisungen
stehen, die nicht berücksichtigt werden:
       MOV     EAX, EBX
Erst hinter dem zweiten Endzeichen dann beginnt der Assembler
wieder seine Arbeit, sodass die folgende Instruktion assem-
bliert wird:
#      MOV     EAX, EBX
```

ECHO %OUT ECHO zeigt auf dem Bildschirm eine Meldung an. Für jede Zeile, die angezeigt werden soll, ist ein eignes ECHO erforderlich.

Verwendung

```
[%] ECHO Text
```

wobei *Text* jedes der Direktive folgende Zeichen einer Zeile bis zu einem eventuell vorhandenen »;« ist, das einen Kommentar einleitet. Spitze Klammern, die in anderen Direktiven-Text klammern, gelten hier nicht als Klammerzeichen und werden angezeigt. Zur Makroexpansion muss der Operator »%« vor die ECHO-Direktive gestellt werden.

%OUT ist ein Synonym für ECHO.

DISPLAY

DISPLAY ist das TASM-Synonym für ECHO. Der anzuzeigende Text muss in Anführungszeichen stehen:



```
DISPLAY "Text"
```

Verwendung

WARN
NOWARN

TASM generiert Fehlermeldungen, wenn er Unstimmigkeiten im Quelltext vorfindet, die ihm eine korrekte Assemblierung unmöglich machen. Manchmal sind die Unstimmigkeiten jedoch nicht so gravierend, dass er den Assemblierungsvorgang abbrechen muss. Ggf. kann er, wenn er bestimmte Annahmen macht, zu einem korrekt assemblierten Modul kommen. Über die gemachten Annahmen aber muss er den Programmierer informieren. Dies erfolgt über *Warnungen*.





MASM kennt keine Warnklassen, sondern nur Warnstufen. Diese können jedoch nicht über eine Direktive im Quelltext angewählt werden, sondern nur via Kommandozeilen-Schalter. TASM kennt aus Kompatibilitätsgründen diese Warnstufen auch, kann sie jedoch ebenfalls nur über die Kommandozeilenschalter nutzen.

Warnstufe		Bedeutung
Schalter	**Meldung**	
/W0	Fehler (A1xxx – A2xxx)	Fatale und nichtfatale Fehler
/W1	W0 und Warnungen (A4xxx)	z.B. die Verwendung nicht-einzigartiger Members in structures und units
/W2	W0 und Warnungen (A4xxx – A5xxx)	z.B. IF ohne Expression, der Assembler nimmt dann 0 für Expression an
/W3	Alle Fehler und Warnungen	
/WX	kein OBJ-File	Wenn Warnungen bestehen, keine OBJ-File-Erzeugung

Tabelle 3.21: MASMs Warnstufen

Gemäß ihrer Art werden TASMs Warnungen in bestimmte Klassen eingeteilt, die mit einem Drei-Buchstaben-Code codiert werden:

Warnklasse		Bedeutung
Code	**Warnung**	
ALN	Segment alignment	Segment ist nicht richtig ausgerichtet.
BRK	Brackets needed	Zur Herstellung der Eindeutigkeit eines Ausdrucks werden Klammern benötigt.
GTP	Global – symbol type mismatch	Der Symbol-Typ stimmt nicht mit dem global deklarierten Symbol überein.
ICG	Inefficient code generation	Der Code kann effizienter gestaltet werden (z.B. mit dem SMART-Modus).
INT	INT3 generation	
LCO	Location counter overflow	Der location counter befindet sich außerhalb der erlaubten Segmentgrenzen.
MCP	MASM compatibility pass	Es wurde ein TASM-Durchgang im MASM-Kompatibilitätsmodus erforderlich.
OPI	Open IF conditional	Es existiert ein offener IF-Block.
OPP	Open procedure	Eine Prozedurdeklaration wurde nicht abgeschlossen.
OPS	Open segment	Ein segment wurde nicht geschlossen.
OVF	Arithmetic overflow	Es hat ein arithmetischer Überlauf stattgefunden.
PDC	Pass-dependent construction	Die gewählte Konstruktion ist abhängig von der Anzahl der Assembler-Durchläufe.
PRO	Write-to-memory using CS	Es wird versucht, im protected mode in ein Segment zu schreiben, das durch den in CS stehenden Selektor identifiziert wird (Code-Segment).
PQK	Asssuming constant for [constant]	[constant] wurde als Konstante verwendet.
RES	Reserved word	Es wurde ein Schlüsselwort als Symbol verwendet.
TPI	Illegal	Illegale Konstruktion

Tabelle 3.22: TASMs Warnklassen

Verwendung Die Nutzung der verschiedenen Warnklassen erfolgt mit den Direktiven WARN und NOWARN bzw. Kombinationen aus beiden:

```
WARN   [Code]
NOWARN [Code]
```

Wird WARN ohne die Angabe einer Klasse verwendet, werden alle Warnklassen verwendet. Ebenso unterbindet NOWARN ohne Angabe einer Klasse die Verwendung jeglicher Warnklasse.

MULTERRS
NOMULTERRS

Standardmäßig erlaubt TASM nur die Anzeige einer Fehlerquelle pro Quelltextzeile. Es gibt jedoch Situationen, in der eine einzige Zeile Grund für mehrere Fehler ist. In diesem Fall reportet TASM nur den »gravierendsten« Fehler.



Bei der Fehlersuche und -beseitigung kann es jedoch hilfreich sein, alle Fehlergründe zu eruieren. Dies ist durch Angabe von MULTERRS möglich. In diesem Fall können pro Quelltextzeile mehrere Fehlermeldungen generiert werden. NOMULTERRS macht diese »Freigabe« rückgängig.

3.2.15 Assembler-Einstellungen

Direktiven steuern, was der Assembler in einzelnen Situationen konkret zu tun hat. Sie sind somit abhängig von Angaben, die für die einzelnen Situationen zutreffen.

Demgegenüber sollte der Assembler jedoch ein »Standardverhalten« aufweisen, das Default- oder Standardwerte in bestimmten Situationen nutzt, in denen der Programmierer keine konkreten Angaben macht. Diese Standard-Vorgaben kann der Assembler in bestimmten Bereichen selbst vorgeben (z.B. die Standardgröße von Segmenten, die wohl heutzutage immer bei 32 Bit liegen dürfte, wenn die aktuelle CPU das unterstützt, weshalb ab Prozessoren vom Typ 80386 genau dies der Default ist). In anderen Bereichen jedoch sollte der Programmierer das Standardverhalten vorgeben können.

Genau diesem Zweck dienen die Direktiven, die in diesem Kapitel besprochen werden.

OPTION

Mit der Direktive OPTION kann das Standardverhalten des Assemblers in gewissen Situationen definiert werden. So können einerseits bestimmte Defaultwerte festgelegt werden, die der Assembler benutzt, wenn bestimmte Direktiven zum Einsatz kommen, andererseits kann Kompatibilität zu vorangehenden Assemblerversionen hergestellt werden.

Hierzu definiert die Direktive OPTION bestimmte Optionen, die sich am MASM orientieren und im Folgenden genannt werden. Einige von ihnen werden durch TASM unterstützt, andere nicht. Dies ist für jede einzelne Option angegeben.

CASEMAP Die Option CASEMAP steuert die Großschreibung von Symbolen. Sie wird wie folgt verwendet:

```
OPTION CASEMAP:MapType
```

wobei *MapType* die Schlüsselworte ALL (alle Symbole werden in Großbuchstaben dargestellt), NONE (kein Symbol) und NOTPUBLIC (nur nicht veröffentlichte Symbole) verwendet werden dürfen.

Die Option darf nur einmal pro Modul verwendet werden, aktuelle Sprachanwahlen im Rahmen von Prozeduren oder PUBLICs haben Vorrang. Defaulteinstellung: NOTPUBLIC.

DOTNAME NODOTNAME Die Option DOTNAME erlaubt die Verwendung eines führenden Punktes in Variablennamen, Labels, Makros und den MemberNames von structures und unions. Es wird jedoch nicht empfohlen, diese Möglichkeit zu nutzen, da dies zu schwer auffindbaren Problemen führen kann, vor allem, wenn die Nutzung des Dot-Operators (im Rahmen der Qualifizierten Angabe eines Membernames) erforderlich wird. Daher kann mit Option NODOTNAME die Nutzung von führenden Punkten in selbst definierten Symbolen ausgeschlossen werden.

```
OPTION DOTNAME
OPTION NODOTNAME
```

Defaulteinstellung: NODOTNAME.

EMULATOR NOEMULATOR Die Option EMULATOR generiert Instruktionen und spezielle Adresslisten (fixup records) für den Linker, sodass ein Software-Emulator für Fließkommaberechnungen benutzt werden kann. Dies ist dann sinnvoll, wenn davon ausgegangen werden muss, dass das System über keine FPU/NPX verfügt (also heute praktisch nicht mehr, da alle Prozessoren ab dem Pentium eine FPU besitzen!). Erforderlich ist dann aber eine FPU-Bibliothek (z.B. innerhalb einer Hochsprache), mit der das betreffende Assemblermodul gelinkt werden muss. Die Option NOEMULATOR generiert die »echten« FPU-Instruktionen.

```
OPTION EMULATOR
OPTION NOEMLATOR
```

Die Option darf nur einmal pro Modul verwendet werden. TASM stellt auch die analogen Direktiven EMUL und NOEMUL zur Verfügung. Defaulteinstellung: NOEMULATOR.

EPILOGUE

Die Option EPILOGUE ermöglicht es, ein anderes Makro als das standardmäßige EpilogueDef für die Generierung des Epilogs von Prozeduren zu verwenden. Dieses Makro wird dann immer in dem Moment aufgerufen, wenn der Assembler eine RET- oder IRET-Instruktion (ohne Argument!) findet. RETN, RETF und IRETF rufen das Epilog-Makro nicht auf!

```
OPTION EPILOGUE:MakroName
OPTION EPILOGUE:EpilogueDef
```

EXPR16
EXPR32

Die Option EXPR16 wählt 16-Bit-Ausdrücke und -Arithmetik, die Option EXPR32 32-Bit-Ausdrücke und -Arithmetik, was auch Standard ist. Diese Optionen haben nur Einfluss auf Expressions und Arithmetik, die im Rahmen von Assembler-Direktiven eingesetzt werden! Sie haben *keinen* Einfluss auf Segmentgrößen (USE16, USE32!) oder verwendete Register (EAX oder AX!).

```
OPTION EXPR16
OPTION EXPR32
```

Defaulteinstellung: ab 80386: EXPR32, davor: EXPR16.

LANGUAGE

Die Option LANGUAGE stellt die standardmäßig zu verwendende Sprache ein, die mit Direktiven verwendet werden soll, die eine Sprachauswahl ermöglichen (PROC, PUBLIC, etc.).

```
OPTION LANGUAGE:Sprache
```

Defaultwert für *Sprache* ist: keine Sprache, es können C, SYSCALL, STDCALL, BASIC, FORTRAN und PASCAL, bei TASM auch CPP (= C++) und NOLANGUAGE (= Assembler) verwendet werden (vgl. auch Tabelle 3.5 auf Seite 612).

LJMP
NOLJMP

Die Option LJMP ermöglicht die Emulation von bedingten Sprüngen mit Distanzen, die über die –128 bzw. +128 Byte von der aktuellen Adresse hinaus gehen (was ja die Restriktion für bedingte Sprünge ist!). Dies erfolgt, indem der Assembler einen bedingten Sprung mit entgegengesetzter Bedingung hinter einen unbedingten Sprung zum eigentlichen Sprungziel codiert:

```
Aus
        :
        JAE     FarL            ; Fehler: Distanz zu groß!
        :
        :
FarL:   :
```

```
wird
        :
        JB      NearL           ; JB ist Gegenteil von JAE
        JMP     FarL
NearL:  :
        :
FarL:   :
```

Die Option NOLJMP verhindert diese Optimierung.

```
OPTION LJMP
OPTION NOLJMP
```

Default ist LJUMPS. TASM stellt auch die analogen Direktiven JUMPS und NOJUMPS zur Verfügung

M510
NOM510

Die Option M510 stellt Kompatibilität bei den Optionen mit der Version 5.10 von MASM her, indem sie folgende Optionen setzt: CASEMAP:ALL, DOTNAME, EXPR16, OLDMACROS, OLDSTRUCTS und SETIF2:TRUE. Daneben wird SCOPED gesetzt, wenn in einer .MODEL-Direktive eine Sprache angegeben wird, andernfalls NOSCOPED. OPTION NO510 stellt die üblichen Defaultwerte für die angegebenen Optionen ein.

```
OPTION M510
OPTION NOM510
```

Defaulteinstellung: NOM510.

NOKEYWORD

Die Option NOKEYWORD gestattet es, eine Liste mit den Namen von Schlüsselworten (nicht vordefinierte Symbole oder Operanden!) anzugeben, die in MASM als reserviert gelten und »freigegeben« werden sollen, ohne bei ihrer Nutzung als selbst definiertes Symbol eine Warnung oder Fehlermeldung zu generieren.

```
OPTION NOKEYWORD:<Symbol [Symbol ] ...>
```

Beispiel: Nach *OPTION NOKEYWORD:<EXTRN>* kann »EXTRN« zur Deklaration von Variablen, Labels oder Prozeduren verwendet werden. ACHTUNG: Die Freigabe reservierter Symbole kann nicht rückgängig gemacht werden! Defaulteinstellung: keine.

NOSIGNEXTEND

Die Instruktionen AND, OR und XOR verfügen über eine Möglichkeit, einen 16- bzw. 32-Bit-Operanden mit einer 8-Bit-Konstanten zu verknüpfen. Dies erfolgt, indem die Konstante intern durch Vorzeichenerweiterung (»sign extension«) auf 16 bzw. 32 Bit gebracht wird, bevor die Verknüpfung erfolgt. Die (zugegebenermaßen alten!) NEC-Prozessoren V25 und V35 kennen diese Form der Operandennutzung nicht. Daher

kann der Assembler über die Option NOSIGNEXTEND gezwungen werden, keine Befehlssequenz auf der Basis dieser Opcodes ($83 / $84) zu erzeugen.

```
OPTION NOSIGNEXTEND
```

Defaulteinstellung: aus.

OFFSET

Die Option OFFSET steuert, ob der Assembler bei der Benutzung des Operators OFFSET den Offset einer Variablen relativ zum Beginn des Segments, in dem sie deklariert wurde, oder relativ zum Beginn des Gruppensegments berechnen soll, wenn das betreffende Segment Teil einer mittels GROUP erstellten Gruppe ist. (Vergleiche hierzu die Problematik auf Seite 633).

```
OPTION OFFSET:OffsetTyp
```

Als Argumente für *OffsetTyp* können GROUP (Offset relativ zum Gruppensegment) oder SEGMENT (Offset relativ zum Segment) verwendet werden. Defaulteinstellung: GROUP. TASM unterstützt diese Option *nicht*.

OLDMACROS
NOOLD-
MACROS

Unter MASM 5.1x verhalten sich Makros teilweise anders als unter MASM 6.1x. So können z.B. in MASM 5.1x Leerzeichen zur Trennung von Argumenten für Makros verwendet werden, MASM 6.1x verlangt Kommata. Mit OLDMACROS kann Kompatibilität zu MASM 5.1x erzwungen werden.

```
OPTION OLDMACROS
OPTION NOOLDMACROS
```

Defaulteinstellung: NOOLDMACROS.

OLDSTRUCTS
NOOLD-
STRUCTS

Unter MASM 5.1x haben die Namen aller members einer structure und der Name der structure die gleiche »Reichweite« im Modul, was bedeutet, dass sie überall im Modul sichtbar sind. Damit dürfen structures und ihre member nicht die gleichen Namen haben! Dies führt dazu, dass der DOT-Operator die gleiche Funktion hat wie der +-Operator, dieser also in Referenzen auf structure members eingesetzt werden kann. Unter MASM 6.1x dagegen können structures und ihre member durchaus den gleichen Namen haben, da die »Reichweite« der Namen der members nur auf die structure begrenzt ist. Dies erzwingt die Nutzung des DOT-Operators, wenn auf structure members referenziert werden sollen. Mit OLDSTRUCT kann Kompatibilität zu MASM 5.1x erzwungen werden.

```
OPTION OLDSTRUCTURES
OPTION NOOLDSTRUCTURES
```

Defaulteinstellung: NOOLDSTRUCTS.

PROC Mit der Option PROC kann die standardmäßig vergebene Sichtbarkeit definiert werden, wenn eine Prozedur mittels der Direktive PROC deklariert wird.

```
OPTION PROC:Visibility
```

Visibility kann die Werte PRIVATE, PUBLIC oder EXPORT annehmen. Defaulteinstellung: PUBLIC.

PROLOGUE Die Option PROLOGUE ermöglicht es, ein anderes Makro als das standardmäßige PrologueDef für die Generierung des Prologs von Prozeduren zu verwenden.

```
OPTION PROLOGUE:MakroName
OPTION PROLOGUE:PrologueDef
```

READONLY NOREADONLY Mit der Option READONLY können Code-Segmente als *read-only* markiert werden. Dies verhindert z.B. selbst modifizierenden Code oder Variable in Codesegmenten.

```
OPTION READONLY
OPTION NOREADONLY
```

Defaulteinstellung: NOREADONLY.

SCOPED NOSCOPED Mit der Option SCOPED kann die Sichtbarkeit von Labels innerhalb von Prozeduren, die mit der »:«-Direktive erzeugt wurden, auf die Prozedur selbst beschränkt werden. Außerhalb der Prozedur ist das Label dann nicht mehr sichtbar. NOSCOPED dagegen macht grundsätzlich alle mit der »:«-Direktive erzeugten Labels im gesamten Modul sichtbar, unabhängig davon, wo sie deklariert wurden. Labels, die mit der LABEL-Direktive oder »::« erzeugt wurden, bleiben von SCOPED/NOSCOPED unbeeinflusst.

```
OPTION SCOPED
OPTION NOSCOPED
```

Defaulteinstellung: SCOPED.

SEGMENT Die Option SEGMENT erlaubt die Einstellung der maximalen Größe der Segmente.

```
OPTION SEGMENT:Size
```

Size kann die Werte USE16, USE32 oder FLAT annehmen. Bei USE16 werden 16-Bit-Segmente generiert, bei USE32 32-Bit-Segmente. FLAT

erzeugt 32-Bit-Segmente des *flat models*. Defaulteinstellung: USE16/ USE32, je nach vorliegendem Prozessor.

SETIF2

MASM ist in der Version 5.1x ein Zwei-Durchlauf-Assembler, was bedeutet, dass er zur Assemblierung zwei Durchläufe (»passes«) benötigt. MASM 6.1x dagegen erledigt die meisten Dinge in einem Durchlauf, TASM praktisch alle. Zusätzliche Durchläufe erfolgen dann nur, wenn unbedingt erforderlich. MASM 6.1x und TASM kommen somit häufig genug in die Situation, einen zweiten Durchlauf nicht durchführen zu müssen. Dies führt dann aber dazu, dass Direktiven, die auf einem zweiten Assemblerlauf aufsetzen oder einen ersten von einem zweiten unterscheiden (IF1, IF2, ELSEIF1, ELSEIF2, .ERR1, .ERR2) zu Fehlern führen oder gar nicht ausgeführt werden (.ERR2, IF2, ELSEIF2).

Um dies zu verhindern, kann die Option SETIF2 auf TRUE gesetzt werden. Dadurch wird erzwungen, dass Durchgang-2-Direktiven (z.B. .ERR2, IF2, ELSEIF2) in jedem Durchgang ausgeführt werden. Steht SETIF2 auf FALSE, fallen diese Direktiven einem ggf. nicht durchgeführten zweiten Durchgang zum Opfer.

```
OPTION SETIF2:TRUE
OPTION SETIF2:FALSE
```

Defaulteinstellung: TRUE.



Nicht jede TASM-Version unterstützt alle OPTIONen des MASM. Ich habe nicht alle ausprobiert und kann daher lediglich feststellen, dass TASM in seiner Dokumentation zu TASM 5.0 im Anhang die entsprechenden Schlüsselworte als *reserviert* bezeichnet und angibt, dass die entsprechenden Optionen unterstützt werden. Glauben wir's halt, warum sollte es auch nicht so sein?

.RADIX

.RADIX ermöglicht die Angabe, welche Zahlenbasis der Assembler als Standard in Ausdrücken und Symbolen verwenden soll. Zu Beginn jedes Assemblermoduls beträgt die aktuelle Basis 10, sodass alle Zahlen als Dezimalzahlen interpretiert werden, wenn sie nicht mit einem entsprechenden Suffix versehen werden. Als Suffixe, die Vorrang vor .RADIX haben, sind möglich:

B oder *b* (binär; Basis: 2), *O* oder *o* (oktal; Basis: 8), *D* oder *d* (dezimal; Basis: 10), *H* oder *h* (hexadecimal; Basis: 16) oder *R* bzw. *r* (Fließkommazahl; Basis: 10).



Achtung! Wenn Sie mit .RADIX die Standard-Zahlenbasis auf Werte größer als 10 einstellen, sind die Zeichen *B* oder *b* bzw. *D* oder *d* gültige *Ziffern* (!) und können daher nicht mehr als Suffix herangezogen werden. So wird bei .RADIX 16 die Ziffernfolge »33D« als Hexadezimalzahl $33D interpretiert, nicht etwa als dezimale Zahl 33 (mit Vorrang gebendem Suffix »D« für Basis 10)! In diesem Fall müssen anstelle der Suffixe *B*, *b*, *D* und *d* die Suffixe *Y* bzw. *y* für binar*Y* und *T* bzw. *t* für *T*en verwendet werden.

Verwendung Formal wird .RADIX wie folgt verwendet:

```
.RADIX Expression
```

wobei *Expression* die Werte 2 (binär), 8 (oktal), 10 (dezimal) oder 16 (hexadezimal) ergeben muss.

RADIX



RADIX ist TASMs Synonym für .RADIX. Es dient vor allem auch zur Einstellung der zu verwendenden Zahlenbasis im IDEAL-Modus, in dem .RADIX nicht definiert ist.

EMUL
NOEMUL



EMUL und NOEMUL sind Synonyme für OPTION EMULATOR bzw. OPTION NOEMULATOR und veranlassen somit den Assembler, bei FPU-Instruktionen entweder Code zu generieren, der eine dazuzubindende Bibliothek mit Software-emulierten FPU-Befehlen nutzt (EMUL), oder die »harten« FPU-Befehlssequenzen zu erzeugen (NOEMUL). EMUL und NOEMUL dienen vor allem der Nutzung im IDEAL-Modus. Siehe auch Seite 680.

JUMPS
NOJUMPS



JUMPS und NOJUMPS sind TASMs Synonyme für OPTION LJMP und OPTION NOLJMP für den IDEAL-Modus. Mit den Direktiven wird die Generierung von emulierten bedingten Sprungbefehlen über größere Distanzen als –128 bis +127 Bytes ermöglicht (JUMPS) oder verhindert (NOJUMS). Siehe auch Seite 681.

IDEAL
MASM



Die Direktive IDEAL schaltet TASM in den IDEAL-Modus, die Direktive MASM stellt dagegen den MASM-Modus von TASM ein. Zu den Unterschieden siehe die Dokumentation von TASM.

VERSION



VERSION dient dazu, TASM in einen Kompatibilitätsmodus zu schalten, in dem er sich absolut wie die ausgewählte Assemblerversion verhält. Es stehen Kompatibilitätsmodi zu Microsofts MASM wie auch zu eigenen Vorgängerversionen des TASM zur Verfügung.

VERSION wird formal wie folgt verwendet: *Verwendung*

```
VERSION Versions-ID
```

Gültige Argumente für *Versions-ID* sind: M400 (MASM 4.0), M500 (MASM 5.0), M510 (MASM 5.1), M520 (MASM 5.20; Quick MASM), T100 (TASM 1.0), T101 (TASM 1.01), T200 (TASM 2.0), T250 (TASM 2.5), T300 (TASM 3.0), T310 (TASM 3.1), T320 (TASM 3.2), T400 (TASM 4.0), T410 (TASM 4.1) und T500 (TASM 5.0).

MASM51
NOMASM51

MASM51 und NOMASM51 sind »alte« Direktiven, die seit Einführung der Direktive VERSION und der *Version-ID* M510 obsolet geworden sind und nur noch aufgrund der Abwärtskompatibilität implementiert sind.

QUIRKS
SMART
NOSMART

MASM verfügt über einige »seltsame Angewohnheiten« (»*quirks*«). TASM versucht im MASM-Modus, diese *quirks* nicht zu zeigen, indem es ein leicht unterschiedliches Verhalten in bestimmten Situationen an den Tag legt. Zu Einzelheiten siehe die Dokumentation zu TASM. Das führt allerdings zu gewissen Inkompatibilitäten, die sich im schlimmsten Fall darin äußern können, dass der Quelltext, der für MASM geschrieben wurde, mit TASM im MASM-Modus nicht korrekt assembliert wird. In solchen Fällen kann es hilfreich sein, mittels der Direktive QUIRKS TASM eine vollständige Kompatibilität zu MASM aufzuzwingen und die quirks eben auch zu zeigen.



Der SMART-Modus von TASM, der zu allen anderen Betriebsmodi des TASM (IDEAL, MASM, QUIRKS) zuschaltbar ist, ermöglicht, was Borland unter »intelligenter Codegenerierung« versteht. In diesem Modus werden einige Befehlssequenzen, die der Assembler für »optimierbar« hält, »optimiert«. Prominentes Beispiel: Aus LEA wird MOV, wenn lediglich eine einfache Adresse als Argument übergeben wird: `MOV Reg, OFFSET Var` anstelle von `LEA Reg, Var`. Weitere »Optimierungen«: vorzeichenerweiterte Boolesche Operationen wo möglich, also `AND Reg, Const8` anstelle von `AND Reg, Const32`. (Spart immerhin drei Bytes in der Befehlssequenz. Na gut!) Und Ersatz eines `CALL FAR` *Adresse* durch `PUSH CS, CALL NEAR` *Adresse*, wenn Ausgangspunkt und Ziel des Sprungs im gleichen Segment stehen. (Auch gut. Mein Tipp jedoch: In diesem Fall gleich `CALL NEAR`! Spart den ganzen Overhead der Schutzkonzept-Prüfungen beim Laden des CS-Registers.).

SMART ist standardmäßig eingestellt und kann mit NOSMART abgestellt werden.

PUSHCONTEXT POPCONTEXT



PUSHCONTEXT und POPCONTEXT kann ein Argument übergeben werden, mit dem ein »Kontext« gesichert oder restauriert werden kann. Hilfreich ist das z.B., wenn in Makros oder Prozeduren ein anderer Kontext eingestellt werden soll als in anderen Programmteilen.

```
PUSHCONTEXT Kontext
POPCONTEXT  Kontext
```

Wobei *Kontext* eines der folgenden Schlüsselworte sein kann: ASSUMES (aktuelle Annahmen zur Register-Segment-Beziehung), RADIX (aktuelle Zahlenbasis), LISTING (Einstellungen für Listig-Steuerungen), CPU (gewählter Befehlssatz), ALL (alle möglichen Kontexte). Hierzu stellt MASM einen Stack mit der Tiefe 10 zur Verfügung.

PUSHSTATE POPSTATE



TASM stellt einen Stack mit einer Tiefe von 16 Einträgen zur Verfügung, der die aktuellen Einstellungen des Assemblers aufnimmt. Hierzu zählen die aktuelle Emulations-Version (siehe VERSION), der Betriebsmodus (IDEAL, MASM, QUIRKS, MASM51), die FPU-Emulation (EMUL), der aktuelle Befehlssatz (.8086 bis .686), MULTERRS, SMART, RADIX, JUMPS, LOCALS und das Zeichen für lokale Labels (LOCALS).

Mit PUSHSTATE können diese Einstellungen auf den Stack geschoben und mit POPSTACK wieder zurückgeladen werden. Hilfreich ist das z.B., wenn in Makros oder Prozeduren eine andere »Umgebung« eingestellt werden soll als in anderen Programmteilen.



PUSHSTATE / POPSTATE sind in gewisser Weise die TASM-Versionen von MASMs PUSHCONTEXT / POPCONTEXT! Da TASM jedoch über verschiedene Betriebsmodi und zusätzliche Direktiven verfügt, werden auch diese berücksichtigt. Lediglich für die Einstellungen zu Listings gibt es eigene Direktiven (vgl. nächster Abschnitt).

%PUSHLCTL %POPLCTL



Analog MASMs PUSHCONTEXT LISTING und seinem POP-Zwilling können auch unter TASM die Einstellungen der Listing-Schalter gesichert und wieder restauriert werden. Dies erfolgt ebenfalls über einen 16-Stufen-Stack und die Direktiven %PUSHLCTL und %POPLCTL



Es können nur »Schalterstellungen« (%LIST, %CONDS, etc.) gesichert werden, nicht aber Einstellungen der Felder eines Listings (%DEPTH, %LINUM, %PCNT, %BIN, %TEXT).

3.2.16 Verschiedenes

FASTIMUL



FASTIMUL hat seine Glanzzeiten bereits erlebt! Wie die anderen »Optimierungsdirektiven« in diesem Kapitel existiert die Direktive seit TASM 3.0 und simuliert eigentlich nur die Drei-Operanden-Form des IMUL-Befehls, die erst mit dem 80386 aufkam. FASTIMUL hat somit nur Bedeutung bei 8086erm und 80286ern – also heute praktisch keine mehr! Es wird absolut analog zum IMUL-Befehl eingesetzt (vgl. Seite 48):

```
FASTIMUL Reg, Reg/Mem, Const
```

FASTIMUL erzeugt eine Sequenz der Instruktionen IMUL, MOV, NEG, SHL, ADD und SUB.

RETCODE
RETF
RETN



Der Befehl RET erzeugt je nach Kontext unterschiedliche Opcodes, je nachdem, ob er eine Prozedur abschließt, die NEAR oder FAR deklariert wurde. Ferner kann es auch Befehlssequenzen erzeugen, je nachdem, ob bei Verwendung der Direktive MODEL ein Epilog erforderlich ist oder nicht. RETCODE, RETF und RETN sind Direktiven, die in keinem Fall einen Epilog erzeugen und entweder ein near return (RETN) oder ein far return (RETF) erzeugen. RETCODE erzeugt bei den Modellen TINY, SMALL, COMPACT und TPASCAL ein near return, bei den Modellen MEDIUM, LARGE, HUGE und TCHUGE ein far return. Mehr ist nicht zu sagen!



Das heißt: Vielleicht doch! Nutzen Sie RETCODE, RETF und RETN nur dann, wenn Sie wissen, was Sie tun! Eine FAR deklarierte Prozedur mit RETN abzuschließen wird Ihnen genauso viel Freude bereiten wie der Abschluss einer NEAR deklarierten Prozedur mit RETF. Viel Spaß!

GETFIELD
SETFIELD



SETFIELD und GETFIELD sind Direktiven, die eine bestimmte Sequenz aus den Instruktionen XOR, XCHG, ROL, ROR, OR und MOVZX bilden, mit denen das gezielte Setzen oder Auslesen von Feldern in Records möglich ist. Die Instruktionen werden wie folgt benutzt:

```
SETFIELD FeldName Reg/Mem, Reg
GETFIELD FeldName Reg, Reg/Mem
```

wobei *FeldName* der Name des gewünschten Feldes in einem mit RECORD deklarierten Feld ist, *Reg/Mem* ein Register oder eine Speicherstelle der Größe 8, 16 oder 32 Bit, die dem deklarierten RECORD entspricht, und *Reg* ein Register, das kleiner oder gleich groß sein muss wie

Reg/Mem und den Wert des Feldinhaltes aufnimmt. (Falls *Reg* ein Byte-Register ist, kann es nur die »L«-Variante sein: AL, BL, CL oder DL!) SETFIELD setzt dann in *Reg/Mem* an der Stelle *FeldName* den Wert aus *Reg*, während GETFIELD in *Reg* den durch *FeldName* definierten Feldinhalt aus *Reg/Mem* einträgt.

SETFLAG
FLIPFLAG
MASKFLAG
TESTFLAG



SETFLAG, FLIPFLAG, MASKFLAG und TESTFLAG sind verkappte logische Befehle. So führt SETFLAG eine OR-Instruktion aus, FLIPFLAG ein XOR, MASKFLAG ein AND und TESTFLAG ein TEST. Der Unterschied zu den bestehenden Instruktionen besteht darin, dass die xxxFLAG-Direktiven den jeweils »optimierten« Code erzeugen.

Beispiel: Wenn Sie in Register AX Bit Nummer 8 setzen wollen, so können Sie dies tun mit

```
OR      AX, 0100h
```

Dies kann jedoch »effizienter« codiert werden, wenn lediglich das Byte-Register angesprochen wird:

```
OR      AH, 01h
```

Diese Optimierung erledigt SETFLAG. FLIPFLAG, MASKFLAG und TESTFLAG arbeiten analog.



Ich bin mir nicht sicher, ob diese Art der »Optimierungen« heute noch wirklich wichtig und gerechtfertigt ist. So ist die Ausführungszeit in beiden Fällen identisch und ob das eine eingesparte Byte so viel bewirkt, ist fraglich! Aufgrund fehlender Kompatibilität zu MASM rate ich von der Benutzung der Direktiven ab. Aber sei es drum.

3.3 Operatoren

3.3.1 Operatoren in Ausdrücken

? Der Operator »?« dient als »Initialisierer« im Rahmen der Allozierung von Daten. Er bewirkt, dass ein Datum des in der Deklaration angegebenen Typs alloziert, nicht aber mit einem bestimmten Wert initialisiert wird. Mit Hilfe des ?-Operators allozierte Daten nennt man daher »uninitialisierte« Daten.

() Die runden Klammern werden verwendet, um eine Priorität des umklammerten Ausdrucks festzulegen. So hat der »*«-Operator eine höhe-

re Priorität als der »+«-Operator und ein Ausdruck der Form 3 * 4 + 5 ergäbe daher 17. Durch die Verwendung der Klammern erhält der geklammerte Ausdruck eine höhere Priorität, sodass zunächst die Summe gebildet und mit 3 multipliziert wird, was zum Ergebnis 27 führt.

*****

Der »*«-Operator besitzt zwei Operanden, zwischen denen er steht. Mit ihnen führt er eine Multiplikation durch.

+

Der »Plus«- oder »+«-Operator kommt in einer unären Form (ein Operand) und in einer binären Form (zwei Operanden) vor. In der unären Form steht er vor dem Operanden und hat keinerlei Auswirkungen, da er den folgenden Operanden als positiven Wert kennzeichnet. Dies ist aber die Standardeinstellung für alle numerischen Werte. In der binären Form verknüpft er die beiden Operanden, zwischen denen er steht, durch eine Addition.

-

Auch der »Minus«- oder »-«-Operator kommt in einer unären Form (ein Operand) und in einer binären Form (zwei Operanden) vor. In der unären Form steht er vor dem Operanden und negativiert damit den Wert des Operanden. In der binären Form verknüpft er die beiden Operanden, zwischen denen er steht, durch eine Subtraktion.

/

Der »/«-Operator besitzt zwei Operanden, zwischen denen er steht. Mit ihnen führt er eine Division durch.

.

Der »DOT«- oder ».«-Operator wird üblicherweise im Rahmen von structures und unions verwendet, um einen Member dieser Strukturen zu selektieren. Handelt es sich bei dem selektierten Member selbst um eine Struktur, wird ein weiterer DOT-Operator notwendig, um deren Members anzusprechen. Die allgemeine Verwendung erfolgt somit nach

```
Expression.Member[.Member[.Member ...]]
```

Expression ist hierbei ein Ausdruck, der als *Qualifizierten Typen* eine structure oder eine union repräsentiert. In solchen Konstrukten repräsentieren die Member Offsets des Datums zum Beginn der Struktur. Expression liefert den Offset der Struktur zum Segment- oder Gruppenbeginn. De facto addiert somit der DOT-Operator den Offset eines Members zum Offset der dazugehörigen Struktur, was in der Adresse des Members im Segment bzw. in der Gruppe resultiert.

Der DOT-Operator kann jedoch auch im Rahmen des type casting bei der indirekten Adressierung eingesetzt werden. *Expression* hat dann die Form `(StructName PTR Variable)`:

```
MyStruct STRUCT
   AByte DB ?
   AWord DW ?
ENDS

       MOV     AX, (MyStruct PTR [EBX]).AWord
```

: Der »COLON«- oder »**:**«-Operator wird im Rahmen des *segment overriding* eingesetzt (vgl. auch »segment override« auf Seite 135). Dadurch wird der Assembler veranlasst, Offsets relativ zum angegebenen Segment und nicht zum Standardsegment zu interpretieren. Dies ist bei der Berechnung von Labels (Codesegment) und Daten (Datensegment) im Rahmen der Generierung der Befehlssequenzen erforderlich.

Die formale Verwendung erfolgt nach

```
Segment:Expression
```

wobei als *Segment* ein Segmentregister, der Name eines mittels SEGMENT (vgl. Seite 627) deklarierten Segments oder einer mittels GROUP (vgl. Seite 632) deklarierten Gruppe verwendet werden kann. In den letzten beiden Fällen muss dann allerdings mittels ASSUME (siehe Seite 635) eine Beziehung zwischen dem angegebenen Segment (oder der Gruppe) und dem zu verwendenden Segmentregister hergestellt worden sein, da der Assembler zur Generierung der Befehlssequenz ein Segmentregister angeben muss. *Expression* ist ein Ausdruck, der eine Adresse im Speicher ergibt.



Bitte verwechseln Sie den »**:**«-Operator nicht mit der »**:**«-Direktive zur Deklaration eines Labels (vgl. Seite 601)!

[ ] Die eckigen Klammern werden als »Indexoperator« oder zur Darstellung indirekter Adressierung benutzt:

```
Expression1[Expression2]
[Register]
```

Expression1 muss in diesem Fall eine Adresse ergeben, zu der ein Offset addiert werden kann, der durch den Index *Expression2* und den aktuellen Typ berechnet werden kann. *Register* ist ein Allzweckregister des Prozessors.

Beispiele: Im ersten Fall wird dem Assembler mitgeteilt, dass im Register EBX eine Adresse steht, die auf ein Word im Speicher zeigt. Dieses Word ist in EAX zu kopieren. Register EBX wird somit zur indirekten Adressierung verwendet.

```
        MOV     EAX, WORD PTR [EBX]

Drei            EQU 1 + 1               ; weil Index »1« = Offset »0«
MyDoubles       DD 1, 2, 3, 4, 5, 6
        MOV     EAX, MyDoubles[Drei]
```

Im zweiten Beispiel liefert *MyDoubles* die Adresse des ersten einer Folge von sechs DWORDs zurück. Durch die Angabe des Symbols *Drei*, das den numerischen Wert 2 erzeugt, als Index auf *MyDoubles* wird der Assembler veranlasst, zu der Adresse von *MyDoubles* noch zweimal die Größe eines DWords (also 8) zu addieren. Dies ist in der Tat die Adresse des DWords mit dem Wert »3«.

Die Operatoren »"« und »'« werden im Rahmen der Allozierung von Strings eingesetzt (vgl. Seite 561). Sie sind gleichbedeutend, können also wahlweise eingesetzt werden. Einzige Einschränkung: Wer mit »"« eröffnet, muss auch mit »"« schließen und umgekehrt. " '

Strings dürfen nicht länger als 255 Zeichen sein und müssen innerhalb einer Zeile deklariert werden. Ist das nicht möglich, so kann der Operator @CatStr oder die Direktive CATSTR verwendet werden.

Sollen die Zeichen »"« oder »'« als Zeichen im String verwendet werden, müssen sie verdoppelt werden:

```
Msg DB "Man sagt: ""Entweder man hat''s, oder nicht."""
```

ADDR kann bei der Direktive INVOKE dazu benutzt werden, die Adresse einer Variablen als Pointer zu übergeben: **ADDR**



```
MyString DB      "Dies ist ein String"
FPTR     TYPEDEF FAR PTR BYTE

MyProc PROC NEAR APointer:FPTR
:
ENDP

INVOKE MyProc, ADDR MyString
```

AND Mit dem AND-Operator lässt sich im Rahmen von Ausdrücken eine UND-Verknüpfung zweier Ausdrücke erreichen:

```
Expression1 AND Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben. Das Ergebnis der Verknüpfung hat die gleiche Größe wie Expression1 bzw. Expression2.

CODEPTR 

Der Operator CODEPTR gibt die Defaultgröße der Adressen zurück, mit denen die im Operanden übergebene Prozedur arbeitet. Diese Adressengröße ist z.B. vom gewählten Modell abhängig, aber auch davon, ob 32-Bit- oder 16-Bit-Umgebungen verwendet werden (z.B. .8086 bzw. .386).

```
CODEPTR Expression
```

CODEPTR kann damit z.B. im Rahmen von bedingter Assemblierung sinnvoll eingesetzt werden, wenn in Abhängigkeit der Adressgröße unterschiedlicher Code verwendet werden soll.

DATAPTR 

Der Operator DATAPTR ist das Daten-Pendant zu CODEPTR. Er gibt die Standard-Operandengröße zurück, mit der ein Zugriff auf Daten erfolgt. Diese ist ebenfalls vom gewählten Modell und der aktuellen Umgebung abhängig.

```
DATAPTR Expression
```

DATAPTR kann damit z.B. im Rahmen von bedingter Assemblierung sinnvoll eingesetzt werden, wenn in Abhängigkeit der Operandengröße unterschiedlicher Code verwendet werden soll.

DUP Der Operator DUP wiederholt die Deklaration, die im Ausdruck Expression angegeben wird, count-mal:

```
Count DUP Expression
```

So führt z.B. die Zeile

```
Str     255 DUP ?
```

zur Deklaration von 255 nicht initialisierten Bytes, die unter dem Namen Str angesprochen werden können.

EQ Mit dem EQ-Operator lässt sich im Rahmen von Ausdrücken eine Prüfung auf Gleichheit zweier Ausdrücke oder Labels erreichen:

```
Expression1 EQ Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben oder valide Labels sein. Das Ergebnis des Vergleichs ist 0 (= false) für »nicht gleich« bzw. –1 (= true) für »gleich«.

GE

Mit dem GE-Operator lässt sich im Rahmen von Ausdrücken eine Prüfung erreichen, ob zwei Ausdrücke oder Labels wertmäßig gleich sind oder *Expression 1* größer als *Expression2*:

```
Expression1 GE Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben oder valide Labels sein, die dann alphabetisch verglichen werden. Das Ergebnis des Vergleichs ist 0 (= false) für »nicht größer oder gleich« bzw. –1 (= true) für »größer oder gleich«.

GT

Mit dem GT-Operator lässt sich im Rahmen von Ausdrücken eine Prüfung erreichen, ob *Expression 1* größer als *Expression2* ist:

```
Expression1 GT Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben oder valide Labels sein, die dann alphabetisch verglichen werden. Das Ergebnis des Vergleichs ist 0 (= false) für »nicht größer « bzw. –1 (= true) für »größer«.

HIGH

Der Operator HIGH ermittelt aus einem Ausdruck das high byte, also das obere, »most significant« Byte eines Words.

```
HIGH Expression
```

Expression muss eine Konstante oder einen OFFSET ergeben.

HIGH

Im IDEAL-Modus von TASM kann dem Operator auch ein Typ vorangestellt werden:



```
Type HIGH Expression
```

In diesem Fall ermittelt HIGH die dem *Type* entsprechende Anzahl der »oberen« Bits von *Expression*. Beispiel: *Word HIGH MyDoubleWord* ermittelt aus *MyDoubleWord* die oberen 16 Bits (*Word*).

HIGHWORD

Der Operator HIGHWORD ermittelt aus einem Ausdruck das high word, also das obere, »most significant« Word des Ausdrucks.




```
HIGHWORD Expression
```

Expression muss eine Konstante oder einen OFFSET ergeben.

LARGE Der Operator LARGE bewirkt, dass *Expression* 32-bittig interpretiert wird, wenn der ausgewählte Prozessortyp mindestens .386 ist.



```
LARGE Expression
```

So kann z.B. die Angabe

```
JMP     [DWORD PTR xxx]
```

auf zwei Weisen interpretiert werden: FAR-JMP mit 16-Bit-Selektor und 16-Bit-Offset (= 32 Bit in DWORD) oder NEAR-JMP mit 32-Bit-Offset (in DWORD). Mittels LARGE kann eine Entscheidung für die letzte der beiden Möglichkeiten getroffen werden:

```
JMP     LARGE [DWORD PTR xxx]
```

erzwingt die Interpretation als NEAR-JMP mit 32-Bit-Offset.

LE Mit dem LE-Operator lässt sich im Rahmen von Ausdrücken eine Prüfung erreichen, ob zwei Ausdrücke oder Labels wertmäßig gleich sind oder *Expression 1* kleiner als *Expression2*:

```
Expression1 LE Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben oder valide Labels sein, die dann alphabetisch verglichen werden. Das Ergebnis des Vergleichs ist 0 (= false) für »nicht kleiner oder gleich« bzw. –1 (= true) für »kleiner oder gleich«.

LENGTH Der Operator LENGTH ermittelt die Anzahl von Einheiten oder Elementen (»entities«, »elements«), die mit *Symbol* verbunden sind.

```
LENGTH Symbol
```

Wichtig ist hierbei, dass LENGTH bei MASM und TASM im MASM-Modus nur diejenigen Elemente als Teil einer Multi-Elementstruktur interpretiert, die mit dem (»äußersten«) DUP-Operator erzeugt wurden:

```
Msg            DB 'Dies ist ein String'
3DArray DD 20 DUP (10 DUP (1, 2, 3, 4, 5))
Sequence       DD 1, 2, 3, 4, 5
Field          DD 5 DUP (0)

LMsg      = LENGTH(Msg)       ; =  1: kein DUP-Operator
L3DArray  = LENGTH(3DArray)   ; = 20: 20 DUP
LSequence = Length(Sequence)  ; =  1: kein DUP
LField    = Lenght(Filed)     ; =  5: 5 DUP
```

Im IDEAL-Modus hat LENGTH die Funktion von LENGTHOF. Beachten Sie bitte auch, dass es bei MASM diesen Operator nur noch aus Kompatibilitätsgründen gibt. Er wurde durch LENGTHOF ersetzt.

LENGTHOF

Im Gegensatz zu LENGTH gibt LENGTHOF tatsächlich die Anzahl aller Einheiten zurück, die mit Symbol verbunden sind, unabhängig davon, ob sie mit oder ohne DUP erzeugt wurden.



```
LENGTHOF Variable
```

LENGTHOF kann nur Symbole auswerten, die die Adressen von Variablen darstellen.

```
Msg            DB 'Dies ist ein String'
3DArray DD 20 DUP (10 DUP (1, 2, 3, 4, 5))
Sequence       DD 1, 2, 3, 4, 5
Field          DD 5 DUP (0)

LMsg      = LENGTH(Msg)              ; =   19
L3DArray  = LENGTH(3DArray)          ; = 1000
LSequence = Length(Sequence)         ; =    5
LField    = Lenght(Filed)            ; =    5
```

LOW

Der Operator LOW ermittelt aus einem Ausdruck das low byte, also das untere, »least significant« Byte eines Words.

```
LOW Expression
```

Expression muss eine Konstante oder einen OFFSET ergeben.

LOW

Im IDEAL-Modus von TASM kann dem Operator auch ein Typ vorangestellt werden:



```
Type LOW Expression
```

In diesem Fall ermittelt LOW die dem *Type* entsprechende Anzahl der »unteren« Bits von *Expression*. Beispiel: *Word LOW MyDoubleWord* ermittelt aus *MyDoubleWord* die unteren 16 Bits (*Word*).

LOWWORD

Der Operator LOWWORD ermittelt aus einem Ausdruck das low word, also das untere, »least significant« Word des Ausdrucks.



```
LOWWORD Expression
```

Expression muss eine Konstante oder einen OFFSET ergeben.

LROFFSET 

LROFFSET hat im Prinzip die gleiche Funktion wie OFFSET, nur dass eine Auflösung des Ausdrucks erst durch den Linker erfolgt (*linker resolved offset*). Dies wird erforderlich, wenn unter Windows Real-Mode-Segmente Verwendung finden sollen.

LT Mit dem LT-Operator lässt sich im Rahmen von Ausdrücken eine Prüfung erreichen, ob *Expression 1* kleiner als *Expression2* ist:

```
Expression1 LT Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben oder valide Labels sein, die dann alphabetisch verglichen werden. Das Ergebnis des Vergleichs ist 0 (= false) für »nicht kleiner « bzw. –1 (= true) für »kleiner«.

MASK MASK erzeugt aus dem Record oder dem RecordField eine Bitmaske.

```
MASK Record
MASK Recordfield
```

In dieser Maske ist jedes Bit gesetzt, das im gesamten Record oder im einzelnen Recordfeld gesetzt ist. Alle anderen Bits sind gelöscht.

MOD Der MOD-Operator ermittelt den Modulus zweier Ausdrücke, also den Rest, der bei einer Integer-(= Ganzzahl-)Division von *Expression1* durch *Expression2* übrig bleibt:

```
Expression1 MOD Expression2
```

Unnötig zu sagen, dass beide Ausdrücke bei der Auflösung eine Integer ergeben müssen.

NE Mit dem NE-Operator lässt sich im Rahmen von Ausdrücken eine Prüfung auf Nicht-Gleichheit zweier Ausdrücke oder Labels erreichen:

```
Expression1 NE Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben oder valide Labels sein. Das Ergebnis des Vergleichs ist 0 (= false) für »nicht verschieden« bzw. –1 (= true) für »verschieden«.

NOT Mit dem NOT-Operator lässt sich im Rahmen von Ausdrücken eine logische Negierung des Ausdrucks erreichen:

```
NOT Expression
```

Expression muss einen numerischen Wert ergeben. Das Ergebnis der Negierung hat die gleiche Größe wie Expression.

OFFSET

Der OFFSET-Operator ermittelt die Adresse von *Expression*, das üblicherweise eine Variable oder ein Label darstellt, innerhalb des Segmentes oder, falls das Segment Teil einer Gruppe ist, innerhalb der Gruppe. Das bedeutet, dass solche Adressen immer relativ zum Segment- oder Gruppenbeginn zu betrachten sind (eben Offsets!).

```
OFFSET Expression
```

Expression muss ein Ausdruck sein, der entweder eine Konstante ergibt oder sich auf ein deklariertes Symbol bezieht, das eine Speicherstelle darstellt. Register oder Segmente können durch OFFSET nicht adressiert werden.

Trotzdem akzeptiert OFFSET auch Segmente als Operanden. In diesem Fall wird die Adresse des letzten Bytes des Segments ermittelt, also die Größe des Segmentes selbst. Bitte beachten Sie jedoch, dass dies relativ zum Gruppenbeginn erfolgt, falls das Segment Teil einer Gruppe ist.

Mit Hilfe der *OPTION OFFSET* und/oder der *OPTION M510* kann bewirkt werden, dass OFFSET die Adresse auch im Fall einer Gruppenzugehörigkeit des Segments auf den Segmentbeginn bezieht.

OPATTR

Der Operator OPATTR (operand attribute) ist ein Operator, der in neueren MASM-Versionen den Operator .TYPE ersetzt. Er gibt ein Bit-codiertes Word zurück, das Auskunft über den Typ und die Sichtbarkeit des in *Expression* angegebenen Ausdrucks gibt.



```
OPATTR Expression
```

Zu Einzelheiten siehe .TYPE

OR

Mit dem OR-Operator lässt sich im Rahmen von Ausdrücken eine ODER-Verknüpfung zweier Ausdrücke erreichen:

```
Expression1 OR Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben. Das Ergebnis der Verknüpfung hat die gleiche Größe wie Expression1 bzw. Expression2.

PROC

PROC ist in TASMs IDEAL-Modus ein Synonym für PTR mit dem *QualifiedType* PROC. Der Operator dient dazu, den in *Expression* übergebenen Ausdruck einem type casting mit dem Typ RPOC zu unterwerfen.



```
PROC Expression
```

PTR Der Operator PTR kann dazu benutzt werden, den in *Expression* übergebenen Ausdruck so zu behandeln, als wäre er vom Typ *QualifiedType*. Dies nennt man im Allgemeinen »type casting«.

```
QualifiedType PTR Expression
```

Auf diese Weise kann PTR bei Symbolen, die (noch) nicht deklariert wurden (»forward references«), oder bei Adressen ohne Label (»indirekte Adressierung«) eingesetzt werden, dem Assembler Angaben zu Größe und/oder Typ des Datums zu machen oder zu seiner Distanz (im Falle von »echten« Labels). Diese Angaben braucht er für Prüfzwecke, aber vor allem für die Zusammenstellung der korrekten Befehlssequenz. Ferner kann PTR in dieser Form dazu verwendet werden, deklarierte Daten in einem neuen Kontext zu interpretieren:

```
MyDWord      DD 012345678h

      MOV    AX, WORD PTR MyDWord
      MOV    BX, WORD PTR [MyWord + 2]
```

Als QualifiedType können alle Typdeklarationen eingesetzt werden, die Informationen der entsprechenden Art vermitteln (vgl. Seite 560). Darüber hinaus sind selbstverständlich auch eigene Typdeklarationen erlaubt.

In einer zweiten Funktion kann PTR auch eingesetzt werden, um einen qualifizierten Typen zu deklarieren:

```
[Distanz] PTR [QualifiedType]
```

Er ist dann Teil einer Typdeklaration:

```
PByte TYPEDEF FAR PTR BYTE
```

SEG SEG ist der Operator, der den zweiten Teil einer logischen Adresse ermittelt: das Segment. Er ist damit der Counterpart zu OFFSET.

```
SEG Expression
```

SEG gibt den Selektor des Segmentes oder der Gruppe zurück, die in Expression übergeben wird. SEG kann damit dazu verwendet werden, ein Segmentregister mit dem korrekten Selektoren zu beladen:

```
Test SEGMENT WORD PUBLIC
       TestDate       DD 0
ENDS

.CODE
```

```
ASSUME DS:Test
       MOV     AX, SEG TestDate
       MOV     DS, AX
       MOV     EAX, OFFSET TestDate
       MOV     EBX, [EAX]
```

Gemäß seiner Aufgabe kann der SEG-Operator nur Ausdrücke akzeptieren, die zu Adressen auf Variable, Labels, Segment- oder Gruppennamen oder Speicheradressen evaluiert werden können. Die Angabe von Konstanten oder anderen Symbolen ist nicht möglich.

SHL

Mit dem SHL-Operator lässt sich eine logische Bit-Verschiebung um *Count* Bits nach links erreichen:

```
Expression SHL Count
```

Expression muss einen numerischen Wert ergeben.

SHR

Mit dem SHR-Operator lässt sich eine logische Bit-Verschiebung um *Count* Bits nach rechts erreichen:

```
Expression SHL Count
```

Expression muss einen numerischen Wert ergeben.

SIZE

Der Operator SIZE ermittelt die Größe von *Symbol*:

```
SIZE Symbol
```

nach der Formel:

```
SIZE = LENGTH * TYPE
```

SIZE ist somit eng an die Operatoren LENGTH und TYPE gebunden.

Im IDEAL-Modus hat SIZE die Funktion von SIZEOF, das ansonsten bei TASM nicht definiert ist. Beachten Sie bitte auch, dass es bei MASM diesen Operator nur noch aus Kompatibilitätsgründen gibt. Er wurde durch SIZEOF ersetzt.

SIZEOF

Der Operator SIZEOF ermittelt die Größe von *Variable* oder *Type*:



```
SIZEOF Variable
SIZEOF Type
```

Auch SIZEOF ist abhängig von LENGTHOF:

```
SIZEOF = LENGTHOF * TYPE
```

SMALL Der Operator SMALL bewirkt, dass *Expression* 16-bittig interpretiert wird, wenn der ausgewählte Prozessortyp mindestens .386 ist.



```
SMALL Expression
```

So kann z.B. die Angabe

```
JMP     [DWORD PTR xxx]
```

auf zwei Weisen interpretiert werden: FAR-JMP mit 16-Bit-Selektor und 16-Bit-Offset (= 32 Bit in DWORD) oder NEAR-JMP mit 32-Bit-Offset (in DWORD). Mittels SMALL kann eine Entscheidung für die erste der beiden Möglichkeiten getroffen werden:

```
JMP     SMALL [DWORD PTR xxx]
```

erzwingt die Interpretation als FAR-JMP mit 16-Bit-Selector und 16-Bit-Offset.

SYMTYPE SYMTYPE ist ein Alias für .TYPE und nur bei TASM und nur im IDEAL-Modus deklariert. Er gibt ein Bit-codiertes Byte zurück, das Auskunft über den Typ und die Sichtbarkeit des in *Expression* angegebenen Ausdrucks gibt.



```
SYMTYPE Expression
```

Zu Einzelheiten siehe .TYPE

THIS Der Operator THIS gibt einen Operanden des Typs QualifiedType zurück, dessen Segment und Offset dem Wert des aktuellen location counters entspricht. Bei Codesegmenten ist dies die Adresse der aktuellen Befehlszeile, bei Datensegmenten die Adresse des aktuellen Datenbytes.

```
THIS QualifiedType
```

THIS NEAR ist äquivalent zu $.

```
.data

FieldStart              DD 20 DUP ?
FieldEnd        EQ THIS BYTE
FieldSize       EQ FieldEnd - FieldStart

.code
```

```
Lbl1:   :
        :
        :
        MOV     EAX, OFFSET THIS BYTE
        SUB     EAX, OFFSET Lbl1
```

.TYPE

.TYPE ist ein Operator, der bei TASM im MASM-Modus und bei MASM nur noch zwecks Abwärtskompatibilität implementiert ist. Er wurde durch OPATTR (operand attribute) ersetzt. Dies erklärt auch den Sinn der beiden Operanden: Sie geben ein Bit-codiertes Byte bzw. Word zurück, das Auskunft über den Typ und die Sichtbarkeit des in Expression angegebenen Ausdrucks gibt.

```
.TYPE Expression
OPATTR Expression
```

.TYPE nutzt nur die 8 Bits eines Bytes zur Codierung. Sie erfolgt gemäß folgender Tabelle.

Bit	Bedeutung
0	Der Ausdruck codiert ein code label
1	Ausdruck, der eine Speicherstelle adressiert, ggf. auch reloziertes Datenlabel
2	Konstanten-Ausdruck (untypisiertes Symbol)
3	Der Ausdruck benutzt direkte Adressierung
4	Der Ausdruck codiert ein Register
5	Der Ausdruck ist fehlerfrei und referenziert keine undeklarierten labels
6	Der Ausdruck ist eine Speicherstelle, die im Stacksegment angesiedelt ist
7	Der Ausdruck referenziert ein externes Label

OPATTR dagegen definiert auch die Bits 15 bis 8, wobei zurzeit jedoch nur die Bits 10 bis 8 verwendet werden gemäß folgender Tabelle:

Bit 8	Bit 9	Bit 10	Eingestelltes Sprachinterface
0	0	0	Keine Sprache
0	0	1	C / C++
0	1	0	SYSCALL
0	1	1	STDCALL
1	0	0	Pascal
1	0	1	Fortran
1	1	0	Basic
1	1	1	reserviert (vermutlich FLAT, jedoch keine Angaben gefunden)

TYPE Der Operator TYPE gibt einen Wert zurück, der je nach *Expression* eine Größe oder eine Distanz darstellt.

```
TYPE Expression
```

Bei Variablen wird die Größe der Elemente der Variablen in Bytes zurückgegeben, bei structures die Gesamtgröße der structure. Im Falle von Konstanten wird als Code 0 zurückgegeben, bei Labels die Distanz des Labels (FAR oder NEAR). Wird ein Register als Expression verwendet, wird die Größe des Registers in Bytes zurückgegeben.

WIDTH Der Operator WIDTH gibt die Breite in Bits zurück, die ein im Operanden genannter record oder ein record field besitzt.

```
WIDTH Record
WIDTH RecordField
```

XOR Mit dem XOR-Operator lässt sich im Rahmen von Ausdrücken eine ausschließende ODER-Verknüpfung zweier Ausdrücke erreichen:

```
Expression1 XOR Expression2
```

Expression1 und *Expression2* müssen numerische Werte ergeben. Das Ergebnis der Verknüpfung hat die gleiche Größe wie Expression1 bzw. Expression2.

3.3.2 Operatoren für Strings

! Verschiedene Zeichen haben unter Assembler eine bestimmte Bedeutung, z.B. als Operator oder vordefiniertes Symbol (z.B. »$«, »"«, »'«, »<«, »>«, »%«, »,« und »;«). Diese Zeichen können daher nicht ohne weiteres innerhalb von Strings eingesetzt werden. Um sie jedoch auch nutzbar zu machen, kann der Operator ! verwendet werden. Er bewirkt, dass das unmittelbar folgende Zeichen als Zeichen und nicht als Operator interpretiert wird.

```
!Zeichen
```

Falls das Ausrufezeichen selbst verwendet werden soll, muss es dupliziert werden: »!!«.

3.3.3 Run-Time-Operatoren

Run-time-Operatoren dienen, wie der Name schon sagt, dazu, Operationen durchzuführen, die erst im Rahmen der Laufzeit, also nach erfolgter Assemblierung erfolgen können. Sie werden daher in Verbin-

dung mit Direktiven verwendet, die ebenfalls erst zur Laufzeit zur Geltung kommen, wie z.B. die »Direktiven zur bedingten Steuerung des Programmablaufs« auf Seite 652.

Der Gleichheitsoperator == ergibt TRUE, wenn die beiden Ausdrücke gleich sind: **==**

```
Expression1 == Expression2
```

Der Ungleichheitsoperator != ergibt TRUE, wenn die beiden Ausdrücke nicht gleich sind: **!=**

```
Expression1 != Expression2
```

Der Vergleichsoperator > ergibt TRUE, wenn *Expression1* größer als *Expression2* ist: **>**

```
Expression1 > Expression2
```

Der Vergleichsoperator >= ergibt TRUE, wenn *Expression1* größer als oder gleich *Expression2* ist: **>=**

```
Expression1 >= Expression2
```

Der Vergleichsoperator < ergibt TRUE, wenn *Expression1* kleiner als *Expression2* ist: **<**

```
Expression1 < Expression2
```

Der Vergleichsoperator <= ergibt TRUE, wenn *Expression1* kleiner als oder gleich *Expression2* ist: **<=**

```
Expression1 <= Expression2
```

Der Verknüpfungs-Operator || verknüpft zwei Ausdrücke über eine logische ODER-Verknüpfung: **||**

```
Expression1 || Expression2
```

Der Verknüpfungs-Operator && verknüpft zwei Ausdrücke über eine logische UND-Verknüpfung: **&&**

```
Expression1 && Expression2
```

Der Bit-Test-Operator & gibt TRUE zurück, wenn in *Expression1* Bit Nummer *Expression2* gesetzt ist: **&**

```
Expression1 & Expression2
```

! Der Negationsoperator ! unterwirft Expression einer logischen Negierung (NOT):

```
! Expression
```

CARRY? Der Operator CARRY? gibt TRUE zurück, falls das carry flag gesetzt ist:

```
CARRY?
```

OVERFLOW? Der Operator OVERFLOW? gibt TRUE zurück, falls das overflow flag gesetzt ist:

```
OVERFLOW?
```

PARITY? Der Operator PARITY? gibt TRUE zurück, falls das parity flag gesetzt ist:

```
PARITY?
```

SIGN? Der Operator SIGN? gibt TRUE zurück, falls das sign flag gesetzt ist:

```
SIGN?
```

ZERO? Der Operator ZERO? gibt TRUE zurück, falls das zero flag gesetzt ist:

```
ZERO?
```

3.3.4 Operatoren in Makros

Die in diesem Kapitel angegebenen Operatoren sind nur im Zusammenhang mit Makros definiert.

& Der »Substitutionsoperator« & darf nur in Makros und REPEAT-Blöcken verwendet werden, in Strings, die in Anführungszeichen stehen oder in spitzen Klammern. Er dient dort der Substitution des durch ihn markierten Symbol(-teils) mit einem Dummyparameter.

```
NewDate MACRO Dummy
   &Dummy&Var DB 'Dies ist ein String mit dem Wort &Dummy '
ENDM

NewDate Single
NewDate Double
; Diese Makroaufrufe expandieren das Makro zu:
; SingleVar DB 'Dies ist ein String mit dem Wort Single '
; DoubleVar DB 'Dies ist ein String mit dem Wort Double '
```

Der zu substituierende *Parameter* muss zwischen zwei »ampersands« (= &) stehen:

```
&Parameter&
```

es sei denn, *Parameter* wird durch ein Leerzeichen abgeschlossen:

```
&Parameter
```

Dann darf der zweite ampersand entfallen.



Da Ampersands auch innerhalb von Strings stehen dürfen, um dort eine Substitution zu gestatten, muss das string escape character (»!«) verwendet werden, wenn durch »&« nicht eine Substitution erfolgen soll, sondern das Zeichen selbst verwendet werden soll:

```
Logo DB "MeineFirma GmbH !& Co. KG"
```

<> Die spitzen Klammern werden verwendet, um Werte einer Parameterliste eines Makros als einen einzigen Parameter zu übergeben:

```
Test MACRO Dummy1, Dummy2, Dummy3
  macro body
ENDM

Test 1, 2, 3
; erzeugt: Dummy1 = "1", Dummy2 = "2", Dummy3 = "3"

Test <1, 2, 3>
; dagegen erzeugt Dummy1 = "1, 2, 3"; Dummy2 = ""; Dummy3 = ""
```

! Das Ausrufezeichen »**!** « sorgt dafür, dass das sich anschließende Zeichen nicht als Direktive, Operator oder Symbol interpretiert wird, sondern buchstäblich als Zeichen. Auf diese Weise können z.B. ampersands oder die spitzen Klammern in Strings verwendet werden. Wird das Ausrufezeichen selbst benötigt, muss es verdoppelt werden.

% Das Prozentzeichen (Expansionsoperator) ist ein Operator, der Ausdrücke berechnet. Es steht dafür, dass der sich anschließende Ausdruck evaluiert und das Ergebnis als Text behandelt wird:

```
%Expression
```

Expression kann hier ein Ausdruck sein, der einen numerischen Wert ergibt, aber auch ein Textmakro oder eine Makrofunktion (das ist ein

Makro, das mit Hilfe der Direktive EXITM und einem dazugehörigen Argument einen Wert zurückgibt):

```
Test MACRO Dummy
   Lbl&Dummy&:
ENDM

TextMakro EQU <Test>

Test %TextMakro
Test %3*4+5
Test %@WordSize
; Diese Makroaufrufe expandieren das Makro zu:
; LblTest:
; Lbl17:
; Lbl4
```



Der Expansionsoperator kann auch am Anfang einer Zeile stehen. In diesem Fall veranlasst es die Auswertung und Substitution aller Textmakros dieser Zeile, einschließlich derjenigen, die in spitzen Klammern stehen:

```
Tag   EQU 30
Monat EQU 09
Jahr  EQU 2001

% <Datum: Tag.Monat.Jahr>
; Datum: 30.09.2001
```

;; Das doppelte Semikolon leitet wie das einfache auch einen Kommentar ein. Alle Zeichen, die nach dem doppelten Semikolon bis zum Zeilenende folgen, werden als nicht zu berücksichtigender Kommentar behandelt.

Der Unterschied zum einfachen Semikolon liegt darin, dass bei der Makroexpansion Kommentare, die durch ein Semikolon eingeleitet werden, als Kommentarzeile übernommen und (z.B. im Listing) angefügt werden. Kommentarzeilen mit doppeltem Semikolon dagegen werden bei der Expansion nicht berücksichtigt:

```
Test MACRO Dummy
   Lbl&Dummy&:         ; Deklaration des Labels
        MOV EAX, EBX   ;; Kommentar wird nicht expandiert
ENDM
```

```
Test 4711
; expandiert zu:
; Lbl4711:                    ; Deklaration des Labels
;       MOV EAX, EBX
```

3.4 Vordefinierte Symbole

Als vordefinierte Symbole werden Symbole bezeichnet, die der Assembler selbst deklariert und die in der Regel Werte repräsentieren, die eine Aussage über den aktuellen Status des Assemblers oder des bearbeiteten Quelltextes geben.

3.4.1 Vordefinierte String-Symbole (Textmakros)

Die vordefinierten Textmakros enthalten die Information als String und können daher überall dort eingesetzt werden, wo Strings erwartet werden, z.B. bei der Direktive TEXTEQU.

@code

@code gibt den Namen zurück, den das aktuelle Codesegment hat und der über die Direktive .CODE eingestellt wurde. Wurde das Speichermodell TINY gewählt, enthält @code den Wert DGROUP.

@curseg

@curseg gibt den Namen des aktuellen, geöffneten Segments zurück, der mit einer vereinfachten Segmentanweisung oder der Direktive SEGMENT angegeben wurde.

@data

@data gibt den Namen zurück, den das aktuelle Datensegment bzw. die Gruppe hat, zu der das Datensegment gehört und der über die Direktive .DATA bzw. GROUP eingestellt wurde.

@Date
??date

@Date ist ein MASM-Makro und ??date das korrespondierende TASM-Makro, das das aktuelle Datum im Format mm/dd/yy zurückgibt.

@Environ

@Environ wird mit Argument in der Form

```
@Environ(envar)
```



verwendet, wobei envar eine Environment-Variable sein muss. @Environ gibt dann den Inhalt dieser Environment-Variablen zurück:

```
Command TEXTEQU @Environ(COMSPEC)
; Command hat nach Expansion des Textmakros z.B. den Wert
; "C:\COMMAND.COM"
```

@fardata @fardata gibt den Namen zurück, den das aktuelle Segment vom Typ FARDATA hat und der z.B. über die Direktive .FARDATA eingestellt wurde.

@fardata? @fardata? gibt den Namen zurück, den das aktuelle Segment vom Typ FARDATA? hat und der z.B. über die Direktive .FARDATA? eingestellt wurde.

@FileCur 
@FileCur gibt den Namen des aktuellen Quelltext- oder Inclusion-Files zurück, und zwar mit relativem oder absolutem Pfad. Das bedeutet, dass innerhalb von Quelltexten der Name des dazugehörigen Quelltextfiles zurückgegeben wird, innerhalb des Quelltextes von inclusion files der des inclusion files.

@FileName ??filename @FileName gibt wie @FileCur den Namen des aktuellen Quelltext-Files zurück, und zwar mit relativem oder absolutem Pfad. Im Unterschied zu @FileCur gibt @FileName in Inclusion-Files jedoch den Namen des Files zurück, der den Inclusion-File aufgerufen hat. ??filename ist ein TASM-Synonym für @FileName.

@stack @stack gibt den Namen zurück, den das aktuelle Stacksegment bzw. die Gruppe hat, zu der das Stack-Segment gehört und der über die Direktive .STACK bzw. GROUP eingestellt wurde.

@Time ??time @Time ist das MASM-Makro und ??time das korrespondierende TASM-Makro, das die aktuelle Systemzeit im 24-Stunden-Format hh:mm:ss zurückgibt.

3.4.2 Vordefinierte Symbole (Numerische Makros)

Die in diesem Abschnitt genannten Symbole sind vordefinierte Symbole, die einen numerischen Wert zurückgeben.

$ Mit dem Dollarzeichen wird der aktuelle Inhalt des *location counters* abgerufen, also der aktuelle Offset zum Segment. $ ist äquivalent zum Operator THIS NEAR.

In Datensegmenten gibt $ den Offset des nächsten zu deklarierenden Datums zurück, in Codesegmenten die Adresse des nächsten Befehls:

```
.MODEL FLAT
.DATA
Table       DD  20 DUP (?)
TableLength EQU $ - Table
```

@32Bit

@32Bit gibt einen Codewert zurück, der angibt, ob 16- oder 32-Bit-Segmente vorliegen: Bei 16-Bit-Segmenten wird 0 zurückgegeben, bei 32-Bit-Segmenten 1. Dieser Operator ist nur in TASM definiert. Das Makro wird durch Angabe der Direktive .MODEL deklariert.



@CodeSize

@CodeSize gibt an, ob das Codesegment, das mit der .MODEL-Direktive gewählt wurde, NEAR-Adressen (TINY, SMALL, COMPACT, FLAT; Wert = 0) oder FAR-Adressen (MEDIUM, LARGE, HUGE; Wert = 1) verwendet. Das Makro wird durch Angabe der Direktive .MODEL deklariert.

@CPU

@CPU gibt ein Bitfeld zurück, das den gewählten Befehlssatz (vgl. Seite 672) codiert:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	.687	.587	.487	.387	.287		.8087	priv.	.686	.586	.486	.386	.286	.186	.8086

@DataSize

@DataSize gibt an, ob das Datensegment, das mit der .MODEL-Direktive gewählt wurde, NEAR-Adressen (TINY, SMALL, MDEIUM, FLAT; Wert = 0) oder FAR-Adressen (COMPACT, LARGE, HUGE; Wert = 1) verwendet. Das Makro wird durch Angabe der Direktive .MODEL deklariert.

@Interface

@Interface ist ein Makro, das durch Angabe der Direktive .MODEL deklariert wird und Ihnen Informationen zum gegenwärtig eingestellten Sprachinterface und zum gewählten Betriebssystem gibt. Dazu wird ein Bit-codiertes Byte zurückgegeben, dessen Bits 3 bis 6 zurzeit reserviert sind. Die Bits 2 bis 0 codieren folgende Einstellungen:

000_b	kein Sprachinterface	100_b	Pascal
001_b	C	101_b	Fortran
010_b	SYSCALL	110_b	Basic
011_b	STDCALL	111_b	reserviert

Bit 7 ist gelöscht, wenn MS-DOS oder 16-Bit-Windows als Betriebssystem gewählt wurde, und gesetzt, wenn es 32-Bit-Betriebssysteme wie Windows 95, 98(SE), ME, NT, 2000 oder OS/2 sind.

Vorsicht: MASM und TASM gehen hier etwas eigene Wege. TASM ordnet der Bitkombination 111_b die Sprache PROLOG zu und nutzt darüber hinaus auch Bit 3 für die Sprache CPP (C++): 1000_b.



@Line



@Line gibt die aktuelle Zeilennummer im Modul zurück. Falls @Line in einem inclusion file steht, bezieht sich diese Zeilennummer auf den Beginn des inclusion files. Innerhalb eines Makros wird die Nummer relativ zum Beginn des Quelltextes angegeben. Das Makro ist sinnvoll, wenn die Zeilennummer eines benutzerdefinierten Fehlers angegeben werden soll.

@Model

Das Makro @Model gibt einen Code zurück, der Auskunft über das Speichermodell gibt, das mit der Direktive .MODEL gewählt wurde. Es gilt:

0	reserviert	4	MEDIUM
1	TINY	5	LARGE
2	SMALL	6	HUGE
3	COMPACT	7	FLAT

Das Makro wird durch Angabe der Direktive .MODEL deklariert.



Leider gehen MASM und TASM auch in diesem Fall jeweils eigene Wege. So ist unter TASM FLAT gleichbedeutend mit TINY, weshalb FLAT auch mit dem Wert 1 codiert wird. Den freigewordenen Wert 7 verwendet TASM (aus Gründen der Abwärtskompatibilität) für das längst überholte Modell TCHUGE und den Wert 0 für das ebenfalls überholte TPASCAL.

@Startup



Das Makro @Startup ist nur deklariert, wenn die Direktive .STARTUP oder STARTUPCODE angegeben wurde. Es enthält die (NEAR-)Adresse des ersten Bytes des Startup-Codes, der durch die Direktive erstellt wurde, und zeigt damit auf den Einsprungpunkt des Programms.



In der Regel erzeugen die Direktiven nur Startup-Code, wenn das Speichermodell TINY und das Betriebssystem MS-DOS gewählt wurde. Nur in diesem Fall macht auch das Makro Sinn.

@Version



Das Makro gibt einen Drei-Ziffern-String mit der Versionsnummer des aktuell verwendeten Assemblers zurück. Vor MASM 5.x ist dieses Makro nicht deklariert, sodass unter MASM nur Werte > »500« zu erwarten sind.

??Version

??Version ist das TASM-Äquivalent zu MASMs @Version, das dieselbe Information in Form eines Vier-Zeichen-Strings zurückgibt. Die möglichen Strings, die das Makro liefern kann, sind unter der Direktive VERSION auf Seite 686 aufgelistet.



@WordSize

@WordSize ist ein Makro, das die Größe der aktuellen Segmente zurückliefert. So wird bei 16-Bit-Segmenten der Wert 2 zurückgegeben (ein »Word« ist in diesem Falle 2 Byte breit), bei 32-Bit-Segmenten der Wert 4 (ein »Word« hat hier die Breite 4 Bytes). Bitte beachten Sie, dass der Begriff »WordSize« tatsächlich nur auf die Segmente bezogen ist, nicht etwa auf die Breite eines Datums vom Typ WORD!

3.4.3 Makros zur Verwaltung von Strings



String-Verwaltungsmakros gibt es nur unter MASM. Ihre Bedeutung ist schon deshalb limitiert, weil die gleichen Ergebnisse auch durch entsprechende Direktiven erzeugt werden können. Daher werden die entsprechenden Makros hier auch lediglich kurz vorgestellt und ansonsten auf die Besprechung der entsprechenden Direktiven verweisen.

@CatStr

Dieses nur in MASM verfügbare Textmakro erfüllt die gleiche Funktion wie die Direktive CATSTR (vgl. Seite 599) mit dem Unterschied, dass es nur dort eingesetzt werden kann, wo ein Textstring erwartet wird.

```
@CATSTR(String1[, String2, ...])
```

Das Textmakro expandiert keine Makros oder Ausdrücke, falls nicht der Operator »%« eingesetzt wird.

@InStr

Dieses nur in MASM verfügbare Textmakro erfüllt die gleiche Funktion wie die Direktive INSTR (vgl. Seite 599) mit dem Unterschied, dass es nur dort eingesetzt werden kann, wo ein numerischer Wert erwartet wird.

```
@INSTR([Position, ]String1 , String2)
```

Das Makro expandiert keine Makros oder Ausdrücke, falls nicht der Operator »%« eingesetzt wird.

@SizeStr

Dieses nur in MASM verfügbare Textmakro erfüllt die gleiche Funktion wie die Direktive SIZESTR (vgl. Seite 599) mit dem Unterschied, dass es nur dort eingesetzt werden kann, wo ein numerischer Wert erwartet wird.

```
@SIZESTR(String1)
```

Das Makro expandiert keine Makros oder Ausdrücke, falls nicht der Operator »%« eingesetzt wird.

@SubStr Dieses nur in MASM verfügbare Textmakro erfüllt die gleiche Funktion wie die Direktive SUBSTR (vgl. Seite 599) mit dem Unterschied, dass es nur dort eingesetzt werden kann, wo ein Textstring erwartet wird.

```
@SUBSTR(String , Start[, Length])
```

Das Textmakro expandiert keine Makros oder Ausdrücke, falls nicht der Operator »%« eingesetzt wird.

3.4.4 TASM-Symbole für OOP



Die folgenden vordefinierten Symbole sind nur in TASM und nur im Rahmen der OOP-Erweiterungen definiert. Sie werden deklariert, sobald die erweiterte STRUC-Direktive verwendet wird. Auf eine weitergehende Besprechung wird verzeichnet, da OOP-Programmierung unter Assembler in diesem Buch nicht behandelt werden kann.

@Object @Object ist ein Textmakro, das den Namen des aktuellen Objektes enthält. Es kann dazu verwendet werden, das derzeitig aktive Objekt zu referenzieren.

@Table @Table ist ein Symbol vom Typ TableData, das die Methodentabelle des als Operanden übergebenen Objekts enthält.

```
@Table ObjectName
```

@TableAddr @TableAddr ist ein Label, das die Adresse der virtuellen Methodentabelle des als Operanden übergebenen Objekts enthält.

```
@TableAddr ObjectName
```

3.5 Assemblermodule in Hochsprachen

Nachdem wir nun wissen, mit welchen Instruktionen, Direktiven, Operatoren und vordefinierten Symbolen wir unter Assembler arbeiten können, geht es nun darum, dies zu nutzen. Ich werde mich im Rahmen dieses Buches darauf beschränken, zu schildern, wie man Assemblermodule in Hochsprachen einbindet; die Erzeugung vollständig lauffähiger Assembler-Anwendungen unter 32-Bit-Windows ist sicherlich die Kür, gehört aber nicht zur hier zu vermittelnden Pflicht.

Die Einbindung von Assemblermodulen erfolgt – unabhängig von der verwendeten Hochsprache – in drei Schritten:

- Erzeugung des Assembler-Quelltextes
- Assemblierung zum Objekt-File
- Einbindung des Objekt-Files in den Hochsprachen-Quelltext

3.5.1 Erzeugung des Assembler-Quelltextes

Die Erzeugung des Assemblertextes ist einfach: Starten Sie einen beliebigen Editor, z.B. das mit Windows ausgelieferte NOTEPAD.EXE, und geben Sie den Quelltext ein. Dieser sollte wie folgt strukturiert sein:

- Kopf
- Deklaration von Daten
- Programmierung der Instruktionen
- Abschluss

Kopf

Die ersten Zeile eines jeden Assemblerquelltextes sollte die Angabe des Befehlssatzes sein, der im folgenden Quelltext »bekannt« sein soll. Dies erfolgt über die Direktiven, die Sie in Tabelle 3.18 auf Seite 672 finden (je nach verwendetem Assembler kommen natürlich auch die Tabelle 3.19 oder Tabelle 3.20 in Frage).



Gehen Sie an dieser Stelle nicht allzu unkritisch vor! Die Wahl des Befehlssatzes hat durchaus weit reichende Folgen. So können Sie z.B. nur dann die 32-Bit-Register verwenden, wenn Sie mit *.386* »mindestens« den 80386-Befehlssatz aktiviert haben. Auch lässt sich nur dann das Speichermodell »flat« nutzen, das Sie für die Einbindung in 32-Bit-Windows-Programme benötigen.



Mein Tipp an dieser Stelle: So Sie nicht irgendwelche »exotischen« Dinge wie die Freischaltung privilegierter Instruktionen (die Sie wahrscheinlich sowieso nicht nutzen können!) oder die Nutzung von SIMD-Befehlen benötigen, ist *.386* durchaus OK. Weniger geht wegen Windows nicht, mehr ändert an den gegenseitigen Abhängigkeiten nichts!

.MODEL

Der Angabe des Befehlssatzes sollte dann die *.MODEL*-Direktive (siehe Seite 639) folgen. Sie ist insofern wichtig, als Sie hier das verwendete Speichermodell angeben können (müssen!) – bei 32-Bit-Windows: immer »flat«! – sowie die Konventionen zum Aufruf der Routinen.

Schließlich wollen Sie ja die im Assembler-Modul realisierten Routinen aus Hochsprachen heraus aufrufen. Und dies hat nach bestimmten Konventionen zu erfolgen, die wir weiter unten nochmals ansprechen werden.



Die .MODEL-Direktive ist auch erforderlich, um die Direktiven zur »Vereinfachte Segmentkontrolle« (vgl. Seite 639) nutzen zu können.

Deklaration der Daten

Falls Sie nun im Rahmen des Assemblermoduls auf Daten zurückgreifen müssen, sollte nun die Deklaration dieser Daten erfolgen. Hierbei gibt es drei Möglichkeiten:

- Die Daten wurden/werden »außerhalb« des Assemblermoduls deklariert und sollen hier nur genutzt werden. Solche Daten nennt man »externe« Daten.
- Die Daten werden im Assemblermodul deklariert und sollen auch nur hier verwendet werden. Das sind »lokale« Daten, die nur im entsprechenden Assemblermodul sichtbar sind.
- Die Daten werden im Assemblermodul deklariert und sollen auch »außerhalb« des Assemblermoduls sichtbar sein. Diese Daten heißen »öffentlich«.



Nich alle Complier erlauben die Einbindung von öffentlichen Daten. Delphi z.B. macht erheblichen Ärger, wenn unter Assembler ein Datensegment deklariert wird, in diesem ein öffentliches Datum definiert und alloziert wird und von Delphi aus dann benutzt werden soll. Offensichtlich ist der von Delphi benutzte Linker nicht willens, die Adresse des Datensegments in die entsprechenden Bereiche im Assemblermodul nachzutragen. Ausweg: Deklarieren Sie das Datum im Delphimodul und greifen Sie im Assemblermodul via EXTRN-Direktive darauf zu!

.DATA

Die Datendeklaration selbst ist nun denkbar einfach. Falls Sie sich das Leben durch die Angabe der *.MODEL*-Direktive im Kopf erheblich vereinfacht haben, brauchen Sie sich um die eigentliche Deklaration des Datensegmentes und seine Kombination mit dem/den Datensegment(en) der Hochsprache nicht zu kümmern. Geben Sie einfach ».DATA« ein, um auf ein gemeinsames Datensegment zugreifen zu können. Andernfalls müssen Sie mittels einer SEGMENT-Direktive ein Datensegment deklarieren, es korrekt ausrichten, benennen und mittels GROUP zu den Datensegmenten der Hochsprache binden. Denn wie

Sie ja wissen, unterstützen die Hochsprachen in der Regel nur ein Datensegment, das den gesamten virtuellen Adressraum einnimmt.

EXTRN

Machen Sie nun dem Assembler alle (in anderen Modulen wie Hochsprachenmodulen, aber auch anderen Assemblermodulen deklarierten) Daten bekannt, wenn Sie auf sie zurückgreifen wollen. Dies erfolgt einfach, indem Sie vor die Deklaration des Datums das Schlüsselwort EXTRN setzen. Dadurch wird kein Datum alloziert (weil es ja woanders schon besteht!), aber dem Assembler die erforderlichen Informationen gegeben, die er zur Erstellung der Befehlssequenzen benötigt (Typ, Größe etc.).

Lokale Daten

Die Deklaration lokaler Daten erfolgt unspektakulär mit Hilfe der bereits bekannten Direktiven (vgl. »Direktiven zur Datendeklaration« auf Seite 561).

PUBLIC

Genauso unproblematisch ist die Deklaration »öffentlicher« Daten. Sie erfolgt, indem lediglich das Schlüsselwort PUBLIC vor den Namen des deklarierten Datums gestellt wird.



Wie Sie wissen, erfolgt in allen drei Fällen die eigentliche Deklaration des Datums gleich und mit Hilfe der bereits genannten Direktiven. Der Unterschied zwischen den einzelnen Fällen ist der, dass im Falle EXTRN-deklarierter Daten der Assembler die Adresse des Datums im Datensegment nicht selbst bestimmt, da es ja irgendwo bereits besteht. Vielmehr veranlasst er den Linker, der dann das Assemblermodul mit den anderen Modulen zusammenbindet, die Adresse an die entsprechenden Stellen in den Befehlsinstruktionen einzusetzen. Solche Adressen sind in Assembler-Listings, die Sie bei jedem Assemblerlauf anfertigen lassen sollten, in der Regel mit einem kleinen »r« (für relocate) als Adresssuffix markiert. Im Falle PUBLIC-deklarierter Daten dagegen trägt der Assembler dieses Datum in eine Liste ein, auf die der Linker beim Linken wiederum dann zugreift, wenn nun in anderen Modulen auf das hier deklarierte Datum zugegriffen werden soll.



Das heißt also: Sobald ein Datum nicht lokal im Assemblermodul verwendet wird (werden soll), wird es mindestens zweimal deklariert. Am Ort der eigentlichen »Erzeugung« hat es dann als PUBLIC gelabelt zu werden, am Ort der Verwendung als EXTRN. Dies gilt auch für Hochsprachen, wie wir weiter unten noch sehen werden!

Sehr viel mehr gibt es zur Deklaration der Daten nicht mehr zu sagen.

Programmierung der Instruktionen

Auch für das Code-Segment und seine Definition können Sie sich die Arbeit erheblich erleichtern, wenn Sie die .*MODEL*-Direktive verwendet haben. Analog zu .DATA können Sie dann einfach den Teil, in dem Sie programmieren wollen, mit .CODE beginnen. Alles andere, wie z.B. das Zusammenbinden der einzelnen Code-Segmente, erfolgt dann »automatisch«.



Denken Sie auch hier daran, was ich eben zur EXTRN/PUBLIC-Deklaration gesagt habe. Falls Sie Routinen benutzen wollen, die außerhalb des aktuellen Assemblermoduls deklariert und realisiert wurden/werden, müssen Sie diese Routinen dem Assembler dennoch »vorstellen«. Dies erfolgt z.B. sehr einfach über die Deklaration von »Prototypen« z.B. mit der Direktive PROTO (vgl. Seite 614), die auch gleichzeitig die EXTERN/PUBLIC-Deklaration übernimmt. Falls Sie nicht mit PROTO arbeiten wollen, so müssen Sie jedoch die Routinen mittels EXTRN einbinden oder mittels PUBLIC nach außen verfügbar machen, so sie nicht lokal bleiben sollen.



Wichtig – um nicht zu sagen: extrem wichtig! – ist bei der Deklaration von Routinen, dass das Assemblermodul und die externen Module »die gleiche Sprache sprechen«, was Parameter und Funktionsergebnisse betrifft! Wie bereits an anderer Stelle mehrfach geäußert und weiter unten nochmals ausführlich diskutiert besitzt jede Hochsprache bestimmte Übergabekonventionen, die unbedingt einzuhalten sind, soll Chaos vermieden werden. Diese Notwendigkeit wird dadurch nicht gerade erleichtert, dass moderne Hochsprachen von heute selbst über Direktiven verfügen, mit denen solche Konventionen angepasst werden können. So unterscheiden sich (nicht nur) in dieser Hinsicht C++ und Delphi (= Pascal) erheblich voneinander. Beide, C++ und Delphi, lassen sich aber mit entsprechenden Direktiven so einstellen, dass sie auch Module nutzen können, die in der jeweils anderen Sprache erstellt wurden. Und jetzt kommt auch noch der Assembler hinzu!



Daher mein Tipp: Einigen Sie sich auf eine Übergabe-Konvention, die Sie stur in allen Modulen durchhalten! Ich persönlich verwende hierzu gerne STDCALL. Wenn Sie also bei .MODEL, .PROTO, der Deklaration von Routinen im Assembler-Modul und in den Hochsprachen, die diese Module nutzen, grundsätzlich STDCALL verwenden, kann nicht mehr viel schief gehen! Abweichen davon sollten Sie nur in einem Fall: in dem, dass Sie Routinen verwenden wollen (müssen), die Sie nicht programmiert haben und die daher einer bestimmten, vorgegebenen Konvention folgen.

Abschluss

Dies ist wirklich das Trivialste an einem Assemblermodul! Der Abschluss erfolgt durch ein einfaches »END« am Ende des Quelltextes. Alles, was danach kommt, behandelt der Assembler als Kommentar und lässt es bei der Assemblierung unberücksichtigt.



Der Assembler meckert in der Regel, falls ein Quelltext nicht mit einem END abgeschlossen werden sollte. Das hat jedoch keine Auswirkungen auf den Assemblierungsvorgang und das Ergebnis. Er meckert halt nur, weil er ja nicht sicher sein kann, dass mit der letzten Zeile, die er liest, tatsächlich Schluss sein soll und daher warnt, es könnte etwas fehlen. Immerhin könnte ja die Quelldatei beschädigt sein. Findet er dagegen ein END, weiß er, dass das nicht der Fall ist.

3.5.2 Assemblierung zum OBJ-File

Dieser Schritt ist wirklich sehr einfach. Sobald Sie den Quelltext vollständig über einen Editor erstellt haben, rufen Sie Ihren externen Assembler auf und übergeben ihm neben dem Namen der Quelltext-Datei auch ggf. Parameter, die den Assemblierungslauf steuern (»Assemblerschalter«). Alternativ können Sie aber die meisten Funktionen dieser Schalter auch im Quelltext über Direktiven realisieren.

Falls Sie alles richtig gemacht haben, ist das Ergebnis dieser Aktion ein OBJ-File mit dem Objektcode, den der Assembler aus Ihrem Quelltext erzeugt hat. Dieser File kann unmittelbar in eine Hochsprache eingebunden werden.

In der Regel aber wird das nicht alles auf Anhieb klappen und der Assembler erzeugt Warnungen und/oder Fehlermeldungen. Diese werden zwar auf dem Bildschirm ausgegeben; aber je nach Umfang des Quelltextes und Anzahl Ihrer Irrungen und Wirrungen kann die Liste schnell so groß werden, dass sie über den Bildschirm flitzt.



Nicht nur aus diesem Grund empfehle ich eindringlich, Assembler-Listings erstellen zu lassen. Auch wenn Sie vor allem am Anfang nicht davon überzeugt sind, helfen sie doch ungemein bei der Fehlersuche!



Bitte beachten Sie, dass die auf dem Markt befindlichen (oder nicht mehr vertriebenen) Stand-Alone-Assembler in der Regel DOS-Programme sind. Das bedeutet, Sie müssen sie aus einer DOS-Box (WIN95 bis NT) oder einer DOS-Emulation (WIN 2000, XP) heraus starten.

3.5.3 Einbindung in Hochsprachen

Die Einbindung von Assemblermodulen erfolgt recht problemlos, sobald einmal ein OBJ-File vorliegt. Sie erfolgt in zwei Stufen:

- Deklaration der einzubindenden Routinen des Assemblermoduls
- Anweisung an den Compiler, beim Linken das Modul zu berücksichtigen (Einbindung des Moduls)

Das Prinzip ist hierbei in allen Hochsprachen das gleiche, es unterscheiden sich lediglich die Direktiven.



In Delphi gibt es die Direktive *external*, die den Anweisungsblock einer Routine ersetzt. Das bedeutet, dass jede eingebundene Assembler-Routine wie eine Delphi-Routine deklariert wird. Allerdings steht anstelle der Anweisungen das Schlüsselwort *external*:

```
function ASMFunc(var Param: LongInt):LongInt; stdcall;
  external;

procedure ASMProc(Var Param: LongInt); stdcall;
  external;
```



Bitte beachten Sie, dass die Direktive STDCALL hier nur vermeintlich überflüssig ist! Zwar regelt STDCALL die Reihenfolge der Übergabe von Parametern über den Stack, könnte hier also auf den ersten Blick weggelassen werden, weil ja nur ein Parameter übergeben wird. Doch, wie gesagt, stimmt das nur oberflächlich betrachtet! Denn die heutigen Hochsprachen-Compiler optimieren, was das Zeug hält! Und eine Optimierungsart ist, Parameter möglichst über Register und nicht über den Stack zu übergeben (was die Defaulteinstellung nicht nur bei Delphi, sondern auch bei C++-Builder und Visual C++ ist!!), wo immer möglich. Wenn Sie nun nicht exakt wissen, welchen Parameter die rufende (Delphi-/C-/C++-) Routine der gerufenen (Assembler-)Routine über welches Register übergibt, erzeugen Sie bereits oben erwähntes Chaos!



Daher kann ich Ihnen nur dringend empfehlen, auch in vermeintlich »klaren« Situationen wie der eben geschilderten ganz konsequent bestimmte Spielregeln einzuhalten. Solche Spielregeln sind z.B. die Nutzung definierter Schnittstellen, wie sie durch Direktiven wie STDCALL definiert werden. Sie ersparen sich dadurch Frust, der vor allem in der Lernphase schnell die Freude am Assembler vergessen lässt.

Die Direktive STDCALL oben erzwingt die Übergabe des (hier einzigen) Parameters über den Stack. Das bedeutet, dass der Compiler Code erzeugt, der vor dem Aufruf der externen Assemblerroutine den Parameter auf den Stack legt (und bei Funktionen nach Rückkehr aus der Assemblerfunktion das Ergebnis »behandelt«. Soll nun die Assembler-Routine mit der Deklaration im Delphi-Modul harmonieren, was dringend zu empfehlen ist, muss im Assemblermodul als *Sprache* z.B. in der .MODEL-Direktive ebenfalls STDCALL verwendet werden.

{$L}
{$LINK}

Fehlt noch die eigentliche Einbindung des OBJ-Files, den der Assembler erzeugt hat. Hierzu stellt Delphi den Compilerschalter {$L} (oder {$LINK} zur Verfügung, dem der Name (und Pfad) der Datei übergeben wird:

```
{$L C:\Project\ASM\MyObject.OBJ}
{$LINK MyOtherObjec.OBJ}
```



Auch in C gibt es eine Direktive, die das Einbinden extern deklarierter Routinen ermöglicht. Sie heißt hier *extern* und nimmt die gleiche Funktion wie in Delphi war:

```
extern int _stdcall ASMFunc (int param)

extern void _stdcall ASMProc (int param)
```

Zur Bedeutung des »Modifizierers« _stdcall (oder _ _stdcall) kann nur wie im Abschnitt »Delphi« eben geschildert auf die Optimierungen verwiesen werden, die moderne C/C++-Compiler heute haben. Ohne den Modifizierer könnten einige (oder alle) Parameter über Register übergeben werden.



C++ verfügt über einige Konventionen, die C und Delphi nicht kennen. So erzeugt der Compiler bei der Kompilierung von C++-Modulen Funktionsnamen, in denen der Typ der Funktionsargumente verschlüsselt enthalten ist (»name mangling«). Dies ermöglicht unter anderem die Verwendung überladener Funktionen und ist insofern von Bedeutung, als dieses *name mangling* beim Einbinden externer Routinen nicht nur unerwünscht ist, sondern gar zu Fehlern führen würde.

Name Mangling

Da der Assembler das »Namen-Mangeln« nicht unterstützt, müssen Sie zu einem Trick greifen, wenn Sie es nutzen wollen. Erstellen Sie hierzu eine Dummy-Routine mit den gewünschten Parametern und/oder Funktionsergebnissen und lassen Sie sie den C++-Compiler in ein ASM-File schreiben (C++-Builder: Option –S). Den so erhaltenen Code

sollten Sie dann exakt in Ihren Assemblerquelltext übernehmen. So erzeugt der Compiler auf diese Weise z.B. aus

```
void test(int)
{
}
```

ein Output mit dem ASM-Quelltext

```
@test$qi proc near
  push ebp
  mov  ebp, esp
  pop  ebp
  ret
@test$qi endp
```

Zwischen die Zeilen *MOV EBP, ESP* (Ende Aufbau Stack-Rahmen) und *POP EBP* (Beginn Abbau Stack-Rahmen) können Sie dann Ihre Instruktionen einfügen.



Falls Ihnen das zu umständlich ist, wird in C++-Quelltexten eine etwas abgewandelte Form der Deklaration externer Routinen erforderlich:

```
extern "C" int _stdcall ASMFunc (int param)

extern "C" void _stdcall ASMProc (int param)
```

Das zusätzliche *"C"* in der Deklaration der einzubindenden Routine bewirkt, dass der Compiler bei dieser Routine keine Namenserweiterung vornimmt, sich also in dieser Hinsicht wie ein C-Compiler verhält.

Ich persönlich halte das für einfacher und bevorzuge es daher. Aber das ist Geschmackssache!



Diese Deklaration können Sie auch blockweise für mehrere Routinen vornehmen:

```
extern "C" {
  int _stdcall ASMFunc (int param)
  void _stdcall ASMProc (int param)
}
```

und gar auf Prototypen anwenden, die in Header-Dateien stehen:

```
extern "C" {
  #include "mylib.h"
}
```

Was Delphi sein *{$L}*, ist C/C++ seine Direktive *#pragma link*. Sie weist den Linker an, die genannte Datei beim Linken zu berücksichtigen:

#pragma link

```
#pragma link C:\Project\ASM\MyObject.OBJ
```

Dadurch wird das OBJ-Modul, das die deklarierte externe Routine enthält, eingebunden.

3.5.4 Aufrufkonventionen

Wie Sie gesehen haben, ist die Einbindung von Assemblermodulen in Hochsprachen nicht schwer und vermutlich einfacher, als Sie dachten. Doch sollte fairerweise nicht verschwiegen werden, dass die Tücke, wie so häufig, im Detail liegt. Auf Assembler angewendet heißt das:



Achten Sie darauf, dass Assemblermodul und Hochsprachenteil »die gleiche Sprache sprechen«. Dies erfolgt, indem bestimmte Konventionen, die die Hochsprachen vorgeben, eingehalten werden müssen.

C/C++	Delphi	MASM/TASM	Übergabe via	Reihenfolge	Bereinigung	führende Unterstriche	Großschreibung [2]	C++ name mangling
__cdecl	cdecl	C	S	r → l	A	ja	nein	nein
__pascal	pascal	PASCAL	S	l → r	E	nein	ja	nein
__stdcall	stdcall	STDCALL [1])	S	r → l	E	nein	nein	ja
		SYSCALL, CPP	S	r → l	A	nein	nein	nein
__fastcall	register		R	l → r	E	@	nein	ja
	savecall		S	r → l	E	nein	ja	-
		NOLANGUAGE	S	l → r	E	nein	nein	-

A: aufrufender Code; E: Epilog der Routine; l: links; r: rechts; R: Register; S :Stack; @: anstelle des Unterstrichs wird das @-Zeichen vorangestellt.

[1]) Im Assembler entsprechen die Konventionen bei STDCALL denen von PASCAL, wenn die Routine keine variable Argumentenliste enthält, andernfalls denen von SYSCALL;

[2]) Der Assembler erzeugt defaultmäßig Symbolnamen in Großbuchstaben.

Tabelle 3.23: Konventionen verschiedener Sprachinterfaces

Die Aufrufkonventionen, nach denen die Deklaration und Nutzung von Routinen in Assembler- und Hochsprachenmodulen erfolgt, spielen eine entscheidende Rolle. Daher sind sie in Tabelle 3.23 einmal gegenübergestellt.



Bitte beachten Sie, dass der Assembler standardmäßig alle Symbolnamen im Quelltext in Großbuchstaben konvertiert. Das führt zu Problemen mit allen Hochsprachen (C, C++) bzw. Hochsprachen-Interfaces (cdecl, stdcall in Delphi), die eine Unterscheidung zwischen Groß- und Kleinbuchstaben vornehmen (vgl. in der Tabelle: __cdecl, __stdcall, _fastcall bzw. cdecl, stdcall, register). Um die Interfaces zwischen Assembler und Hochsprache einander anzupassen und somit gemeinsame Voraussetzungen für den Linker zu schaffen, können jedoch »Schalter« in der Kommandozeile des Assemblers bzw. Compilers verwendet werden, die die Unterscheidung herbeiführen (TASM: /ml, /mx; MASM: /Ml, /Mx; ML: /Cx, /Cp; C++-Builder: /c+) bzw. die Konversion in Großbuchstaben hervorrufen (TASM: /mu; MASM: /Mu; ML: /Cu; C++-Builder: /c-). Im Quelltext kann dies durch die Direktive OPTION CASEMAP (Assembler) definiert werden.



Der C++-Builder von Borland/Inprise ruft bei der Kompilierung TASM mit der Kommandozeile /ml auf.

Wenn Sie die Tabelle genauer studieren, stellen Sie fest, dass es eigentlich nur drei Interfaces gibt, die sinnvoll eingesetzt werden können, wenn es um die Zusammenarbeit zwischen verschiedenen Modulen geht: C (CPP), PASCAL und STDCALL. Nur diese drei Konventionen kommen in allen drei »Sprachformen« vor, sodass eine relativ problemlose Einbindung unterschiedlicher Module möglich ist, da sowohl der Assembler als auch der Hochsprachen-Compiler die gleichen Formalismen bei der Namensvergabe (Unterstrich, Großschreibung), dem Weg der Übergabe von Parametern an die Routinen (Stack), der Reihenfolge der Ablage auf dem Stack sowie der Bereinigung des Stacks bei der Rückkehr aus der Routine verwenden.

NOLANGUAGE sollte daher nur in Assemblermodulen verwendet werden, um innerhalb des Moduls Routinen aufzurufen, und dann auch nur, wenn Sie die Übergabekonventionen im Einzelfall vorgeben. Ähnliches gilt für SYSCALL. Und was _ _*fastcall/register* und *savecall* betrifft: Das sind Hochsprachen-interne Lösungen, die zwar sehr schnellen Code und gute Optimierungen erlauben (weil häufig genug kein Overhead durch Pro- und Epilog erforderlich wird und die Daten bereits da sind, wo sie sein sollten: in einem Register), die aber wenig geeignet sind, mit »fremden« Modulen zusammenzuarbeiten.



Bitte beachten Sie jedoch, dass sowohl in C/C++ als auch in Delphi heutzutage die Einstellung »register« Standard ist, die Übergabekonventionen nach dem Modell _ _*fastcall* bzw. *register* verwenden. Somit darf bei der Deklaration der externen prozedur die Verwendung des entsprechenden Schlüsselwortes (stdcall, pascal bzw. cdecl) in keinem Fall versäumt werden.

3.5.5 Übergabekonventionen

Falls Sie die »vereinfachte« Segmentkontrolle aktiviert haben (.MODEL), sind der Aufruf von Routinen und die Übergabe von Parametern kein Problem. Deklarieren Sie im Hochsprachenmodul und im Assemblermodul lediglich die gleichen Routinen-Deklarationen (inklusive korrespondierender Parameterdeklarationen) gemäß der eben vorgestellten Kriterien und achten Sie auf das gewählte Sprachmodell. Dann sollte es keine Probleme geben. Tabelle 5.13 auf Seite 816 kann Ihnen dabei helfen, unter Assembler und den Hochsprachen die jeweils entsprechenden Datentypen zu identifizieren.

Dennoch möchte ich noch eine Tabelle anfügen, die Ihnen bei der Programmierung vor allem von Funktionen helfen soll. Denn falls Sie in einem Assemblermodul eine Funktion realisieren, die in ein Hochsprachenmodul eingebunden werden soll, so müssen Sie wissen, an welcher Stelle die Hochsprache dieses Funktionsergebnis erwartet.

C++	Delphi	Rückgabe über
(unsigned) char	Byte/ShortInt	AL
(unsigned) short / int	Word/SmallInt	AX
(unsigned) long	LongWord/LongInt	EAX
	Int64	EDX:EAX
enum	Aufzählungstyp	AL / AX / EAX [1])
float, double, long double	Single/Double/Extended	ST(0)
Zeiger	Zeiger	EAX
	String	Zeiger auf dem Stack

[1]) Je nach Anzahl der Elemente

Tabelle 3.24: Übergabekonvention von Funktionswerten



Bitte beachten Sie, dass diese Tabelle nur in Verbindung mit 32-Bit-Betriebssystemen gilt, in denen das Speichermodell »flat« gewählt wird. In 16-Bit-Welten werden 32-Bit-Werte (wie z.B. long/LongInt) in DX:AX übergeben, wobei in DX die »oberen« 16 Bit liegen. Near-Zeiger werden

in AX zurückgegeben, Far-Zeiger in DX:AX, wobei das 16-Bit-Segment in DX und der 16-Bit-Offset in AX liegt.

3.5.6 FAR und NEAR – eine Frage des Standpunktes

Es fällt einem auf Anhieb schwer, einen Adressraum von 4 GByte Größe als Raum zu empfinden, in dem NEAR-Adressen eine Rolle spielen. Diejenigen, die noch die gute, alte 16-Bit-Zeit kennen, haben gelernt, immer dann von FAR zu reden, wenn man ein 64-kByte-Segment verlassen wollte. Und nun das!

Doch ist die andere Seite der Betrachtung durchaus richtiger: Als FAR wird unabhängig von der tatsächlichen Distanz eine Adresse bezeichnet, wenn sie zur Darstellung eine Segmentangabe und einen Offset in dieses Segment benötigt. NEAR ist die Adresse immer dann, wenn der Offset ausreicht, unabhängig davon, wie groß die entsprechenden Segmente sind.

Da nun aber unter 32-Bit-Betriebssystemen der gesamte Adressraum nur über einen 32-Bit-Offset angesprochen werden kann, sind generell alle Adressen in diesem Speichermodell NEAR!



Behalten Sie daher bitte immer im Hinterkopf, dass Sie bei der Verwendung von Zeigern – sei es Zeigern auf Daten (Variable!), sei es Zeigern auf Instruktionen (Routinen!) – unter 32-Bit-Windows mit NEAR-Zeigern umgehen! Somit werden alle Parameter, die Sie Routinen via *call by reference* übergeben, in Form eines 32-Bit-Zeiger-Offsets übergeben. Das aber bedeutet auch, dass der Wechsel von Datensegmenten nicht erforderlich ist und daher die Ausnahme und für bestimmte Speziallösungen reserviert bleibt.

3.5.7 Tabus

Bedingt durch Konventionen der jeweiligen Hochsprache ist die Verwendung von bestimmten Registern in Assemblermodulen tabu. Na ja – nicht ganz!



Delphi belegt intern die Register EDI, ESI, ESP, EBP und EBX. In DS findet sich die Adresse des Datensegments, in SS die des Stacksegments. Ggf. kann auch in ES ein Selektor auf ein Datensegment stehen. FS kann im Rahmen der strukturierten Ausnahmebehandlung (siehe nächstes Kapitel) eine Rolle spielen und sollte daher auch nicht benutzt werden.

Die Register EAX, ECX und EDX und das Segmentregister GS werden nicht verwendet und stehen dem Programmierer zur Verfügung.



Die Register ESP und EBP, nicht aber EBX, sowie die Segmentregister DS und SS sind auch unter C++-Builder in Verwendung. Ggf. kann auch hier in ES ein Selektor auf ein Datensegment stehen. ESI und EDI werden unter C++-Builder als »Registervariable« benutzt, was bedeutet, dass sie nur dann als tabu zu gelten haben, wenn die Übergabekonvention »_ _fastcall« benutzt wird. Andernfalls können sie benutzt werden ohne gesichert oder restauriert werden zu müssen. FS kann auch hier im Rahmen der strukturierten Ausnahmebehandlung (siehe nächstes Kapitel) eine Rolle spielen und sollte daher auch nicht benutzt werden.

Die Register EAX, EBX, ECX und EDX sowie GS stehen dem Programmierer somit vollständig, ESI und EDI eingeschränkt zur Verfügung.



Dennoch ist es sinnvoll, auch unter C++ die Register EBX, ESI und EDI als reserviert zu betrachten und, wie bei Delphi obligatorisch, auch im Rahmen des Pro- und Epilogs zu sichern und zu restaurieren. Man schließt auf dieses Weise eine potentielle, schwer zu findende Fehlerquelle aus, vor allem da z.B. EBX durch das Betriebssystem belegt sein könnte.



Achtung! Bei Verwendung der Register FS und GS sollten Sie berücksichtigen, dass das Schreiben in diese Register eine Prüfung der Zugriffsrechte im Rahmen der Schutzkonzepte mit sich bringt. FS und GS können daher nur für bestehende Datensegmente verwendet werden und machen somit nur Sinn in Modulen, die Datensegmente definieren, die nicht im Rahmen des FLAT-Modells in die Gruppe _DATA eingebunden werden (sollen).

Das alles bedeutet, dass innerhalb von Assembler-Routinen, die in Hochsprachen-Programme eingebaut werden sollen, die »verbotenen« Register nur dann verwendet werden dürfen, wenn sie vor Gebrauch gesichert und danach wieder restauriert werden. Dies erfolgt am sinnigsten innerhalb des Pro- und Epilogs. Unterbleibt das, führt das mit an Sicherheit grenzender Wahrscheinlichkeit zu schwerwiegenden Problemen, die letztendlich in einem Programmabbruch enden dürften.

Die Assembler unterstützen die Sicherung/Restauration durch die USES-Direktive im Rahmen einer PROC-Deklaration (vgl. Seite 611).

```
ASMProc PROC
   USES  EBX ESI EDI, Param1:DWORD, Param2:DWORD ...
   LOCAL Var1:DWORD, Var2:DWORD
   :
   :
   ret
ASMProc ENDP
```



Bitte beachten Sie, dass die Register auch restauriert werden, wenn aus einer Assembler-Routine eine Hochsprachen-Routine aufgerufen werden soll.

```
ASMProc PROC
   USES  EBX ESI EDI, Param1:DWORD, Param2:DWORD ...
   LOCAL Var1:DWORD, Var2:DWORD
   :
   :
   POP  EDI             ; Register in umgekehrter Reihenfolge
   POP  ESI             ;   des PUSHens durch USES wieder
   POP  EBX             ;   restaurieren
   :                    ; ggf. Parameter auf den Stack
   CALL HighLevelProc   ; HighLevelProc muss EXTRN sein!
   :                    ; ggf. Aufräumen (Ergebnis, Stack)
   PUSH EBX             ; Register in der in USES angegebenen
   PUSH ESI             ;   Reihenfolge wieder auf den Stack!
   PUSH EDI
   :
   :
   ret
ASMProc ENDP
```



USES muss nicht notwendigerweise verwendet werden. Es ist auch möglich, die Register unmittelbar vor und nach Gebrauch zu sichern/restaurieren:

```
ASMProc PROC
   Param1:DWORD, Param2:DWORD ...
   LOCAL Var1:DWORD, Var2:DWORD
   :
   :
   PUSH EDI             ; Register in beliebiger Reihenfolge
   PUSH ESI             ;   auf den Stack PUSHen
   PUSH EBX
   :                    ; Instruktionen, die Verwendung von
   :                    ;   den gesicherten Registern machen
```

```
   :
   POP  EBX             ; Register in der umgekehrten Reihen-
   POP  ESI             ;   folge wieder vom Stack POPpen!
   POP  EDI
   :
   :
   ret
ASMProc ENDP
```

3.6 Assembler und die strukturierte Ausnahmebehandlung (SEH)

Im Kapitel »Exceptions und Interrupts« auf Seite 486 haben wir nur die Interrupt-Behandlung auf der Ebene der Hardware betrachtet. Doch der protected mode wäre nicht der protected mode, wenn nicht das Betriebssystem ein Wörtchen mitzusprechen hätte. Und das tut es auch, in Form der Schutzkonzepte. Um es gleich vorwegzunehmen: Der Programmierer hat nicht mehr viele Möglichkeiten, in die Interrupt-Behandlung einzugreifen: Er kann die IDT nicht verändern, da ein Zugriff auf das IDTR und die IDT ohne die erforderlichen Privilegien, die ihm das Betriebssystem mit Sicherheit nicht geben wird, unmöglich ist. Somit kann er nicht, wie noch zu Zeiten des guten, alten DOS, Interrupts auf eigene Handler »verbiegen« oder eigene der IDT hinzufügen. Er kann sich auch nicht in bestehende Interrupthandler einklinken, da dies auch nur mit den entsprechenden Privilegien möglich ist. Somit kann er eigentlich nur noch per INT-Befehl die entsprechenden Interrupts auslösen. Mit allen daraus folgenden, teilweise bereits besprochenen Konsequenzen. Und das war es schon.

Halt – noch eine kleine Möglichkeit hat er. Und die nennt sich »strukturierte Ausnahmebehandlung« (SEH, *structured exception handling*) und wird vom Betriebssystem unter Win32 zur Verfügung gestellt, um dem Anwender bzw. Anwendungs-Programmierer die Möglichkeit zu geben, auf Exceptions sinnvoll reagieren zu können. Wenn Sie die Hochsprache C gut kennen, kennen Sie den Begriff *Strukturierte Ausnahmebehandlung*. Und genau diese strukturierte Ausnahmebehandlung, die in C realisiert ist, basiert auf der von Windows (in allen 32-Bit-Versionen!) zur Verfügung gestellten Ausnahmebehandlung. Wir werden gleich noch einmal darauf zurückkommen.



Leider (weil es wirklich interessant ist!) kann ich an dieser Stelle nicht detaillierter auf die Realisierung der SEH unter Windows eingehen. Doch hierfür gibt es hervorragende Bücher von hochkarätigen Autoren, die den Ablauf sehr detailliert darstellen. Ich möchte mich darauf beschränken, Ihnen kurz den Ablauf zu skizzieren, wobei ich die Vorgänge bei Windows 2000 schildere. Analog läuft das aber auch in den anderen Varianten (9.x/SE/ME/NT) ab.

Als »*trap*« bezeichnet Microsoft den Vorgang, dass ein (Anwendungs-) Programm aufgrund eines Interrupts oder einer Exception unterbrochen wird und die Behandlung des Interrupts (bzw. der Exception) erfolgen soll. Zuständig in erster Instanz: das Betriebssystem. (Bitte verwechseln Sie also Microsofts *trap* nicht mit dem weiter oben kennen gelernten Exception-Typ *trap* des Prozessors!) Zu diesem Zweck stellt das Betriebssystem »Traphandler« zur Verfügung, die im Kernel (und damit unter höchster Privilegstufe) realisiert sind. Das Betriebssystem kennt vier »*trap handler*«:

- einen Handler für Systemdienste; das sind Software-Interrupt-Aufrufe, die bestimmte Funktionen des Betriebssystems abrufen. So wird beispielsweise der IDT-Eintrag $2A bei der Betriebssystem-Funktion GetTickCount verwendet, $2D für DebugServices und $2E für System-Services allgemein (z.B. im Rahmen von WriteFile). Dieser Systemdienst-Handler ist vergleichbar mit dem Funktionsdispatcher $21 unter DOS.
- einen Handler für Seitenfehler (#PF) und die virtuelle Speicherverwaltung (#NP); dieser Handler ist extrem wichtig und damit absolut abgeschottet, da über ihn die gesamte virtuelle Speicherverwaltung inklusive Paging-Mechanismus, Auslagerungsdatei und weiß der Geier was abläuft. Er sorgt dafür, dass ausgelagerte Seiten oder nicht vorhandene Segmente bei Bedarf nachgeladen werden oder Platz für nachzuladende Seiten/Segmente im physikalischen Speicher geschaffen wird.
- einen Handler für Hardware- und Softwareausnahmen (Exceptions); dieser Handler soll uns im Folgenden ein wenig mehr interessieren, da über ihn die strukturierte Ausnahmebehandlung erfolgt; und
- einen Handler für Interrupt-Dienstroutinen; dieser Handler ist für alle Hardware- und Software-Interrupts zuständig, die nicht in eine der drei anderen Klassen fallen.

Betrachten wir den Exception-Handler. Im Kernel wurde ein »*exception dispatcher*«, ein Ausnahmeverteiler, realisiert, der die Aufgabe hat, einen Handler für die spezifische Ausnahme zu suchen und, wenn er gefunden wurde, die zu behandelnde Ausnahme an diesen zur Bearbeitung weiterzuleiten. Darüber hinaus »filtert« er auch Exceptions heraus, die der Anwender gar nicht bearbeiten soll: Bei #DB oder #BP beispielsweise, also der *debug* oder *break point exception*, wird unmittelbar der Debugger aufgerufen, bei #PF oder #NP, der *page fault* oder *segment not present exception*, wird der Speicherverwaltungs-Handler aufgerufen. Andere Ausnahmen behandelt er, indem er dem Anwendungsprogramm, das die Exception ausgelöst hat, einen Fehlercode zurückliefert.

SEH und das Betriebssystem

Was übrig bleibt, sind Ausnahmen, die der Benutzer behandeln soll. Zu solchen Ausnahmen gehören die Division durch Null (#DE) oder der arithmetische Überlauf (#OF), die FPU-Exceptions (#MF) oder die SIMD-Exceptions (#XF). Solche Ausnahmen werden vom Betriebssystem nicht oder nur teilweise (#MF wegen der MMX-Befehle und der Umgebungssicherung bei task switches) behandelt. Um nun die Verantwortung der Exception-Behandlung an das Anwendungsprogramm weitergeben zu können, erwartet der Ausnahmeverteiler eine ganz bestimmte Struktur (daher der Begriff »strukturierte« Ausnahmebehandlung), nämlich einen »rahmenbasierten« Ausnahme-Handler auf Seiten des Anwendungsprogramms. Rahmenbasiert heißt in diesem Zusammenhang, dass der Ausnahmeverteiler einen Stack-Rahmen (»stack frame«) erwartet, der die notwendigen Informationen enthält, die zum Aufruf des anwenderseitigen Ausnahmehandlers erforderlich sind.

SEH und C

Kommen wir zurück zu C. Die Hochsprache C realisiert genau diese Erwartungen des Betriebssystems. (Wen wundert's, nachdem Windows zum größten Teil in C geschrieben ist.) Hierzu gibt es die C-Anweisungen »__try« und »__except«.



Bitte beachten Sie, dass es neben »__except« auch ein »__finally« gibt, mit dem man den »__try«-Teil beenden kann. Streng genommen handelt es sich bei »try – finally« jedoch nicht um eine strukturierte Ausnahmebehandlung, wenn man als »Ausnahme« das Auftreten von Exceptions und als Ausnahmebehandlung die Behandlung solcher Exceptions versteht. Denn der »finally«-Teil wird *immer* abgearbeitet, also auch dann, wenn im »try«-Teil *keine* Ausnahme aufgetreten ist, sondern wenn er »zu früh«, also nicht regulär z.B. über ein »goto« oder »return« beendet wird. Die »Ausnahme« ist hier also durchaus auch

eine bewusste Programmverzweigung. Da mit diesen Konstrukten keine Exceptions *behandelt*, sondern lediglich sichergestellt werden soll, dass *auch* beim Auftreten von Exceptions bestimmte Codeteile abgearbeitet werden, sollte man »try-finally«-Blöcke nicht mit dem Begriff SEH in Verbindung bringen – auch wenn viele renommierte Autoren und Fachleute von Microsoft das tun.

Es fehlt hier leider der Platz, weiter auf den »kontrollierten Ausstieg«, also die »Ende-Behandlung« mittels try – finally einzugehen. Es bleibt aber festzuhalten, dass diese Blöcke dennoch einen Einfluss auf die strukturierte Ausnahmebehandlung haben und daher nicht ganz losgelöst betrachtet werden dürfen: in Form sog. *local* und *global unwinds*:

local unwind Für einen local unwind streut der Compiler Anweisungen in die Befehlsfolge ein, die sicherstellen, dass nach aufgetretenen Exceptions, aber auch nach anderen Auslösern eines verfrühten »Ausstiegs« (return, break, halt, goto etc.), auch tatsächlich der im Finally-Teil stehende Code ausgeführt wird. Die Abarbeitung dieses Codes – und damit des Finally-Teils – nach einem irregulären Verlassen des Try-Teils nennt man *local unwind*. (Das reguläre Abarbeiten des Finally-Teils ist somit *kein* local unwind!)

global unwind Stehen nun Abbruchursache und resultierende Ende-Behandlung nicht in unmittelbarem Zusammenhang, nennt man die irreguläre Abarbeitung eines Finally-Teils *global unwind*. Falls z.B. in einem Try-Block der Aufruf einer Routine erfolgt und in dieser Routine eine nicht behandelte Exception erfolgt, so muss der den Routinenaufruf enthaltende Try-Teil irregulär verlassen werden: Ein *unwind* ist die Folge. Da aber der Grund dafür nicht im Try-Teil liegt, sondern an räumlich anderer Stelle in einer anderen Umgebung in der Routine, ist er nicht mehr lokal, sondern global. Global ist ein notwendiger unwind auch dann, wenn ein Try-Execpt-Block in einen Try-Finally-Block eingebettet ist und eine Exception auftritt: Die Ursache für den unwind liegt im Try-Teil des Try-Except-Blocks, nicht im Try-Block des »übergeordneten« Try-Finally-Blocks.

Genau dieser Sachverhalt bedingt, dass im Rahmen der Strukturierten Ausnahmebehandlung auch unwinds berücksichtigt werden müssen. Und das verkompliziert den Mechanismus hinter der SEH doch deutlich! Daher muss ich es an dieser Stelle damit bewenden lassen, Sie aufzumuntern, sich mittels des integrierten Debuggers und der CPU-Anzeige selbst Einblick zu verschaffen, falls Sie Details wissen wollen.

SEH und OOP

Nun gibt es zwischen den Programmiersprachen C und C++ einige »kleine« Unterschiede, die die Sache nicht gerade vereinfachen. Und einer der Unterschiede ist: C++ basiert auf dem Konzept der objektorientierten Programmierung (OOP). Das bedeutet, dass die Exception-Behandlung mit und durch Exception-Objekte erfolgt. Den gleichen Weg geht Delphi. Und damit dürfte klar sein, dass es in diesen Programmiersprachen nicht einfach damit getan ist, einen Programmteil zu realisieren, der den rahmenbasierten Exception-Handler darstellt, auf den das Betriebssystem zurückgreifen könnte. Vielmehr muss es Mechanismen geben, die die Verbindung zwischen der strukturierten und der objektorientierten Ausnahme-Behandlung herstellen.

Sie können sich vorstellen, dass dies nicht ganz einfach ist. Und es würde daher weit über das hinausgehen, was im Rahmen eines Assemblerbuches geschildert werden kann, wenn ich nun diese Mechanismen detailliert besprechen würde. Es ist eigentlich auch gar nicht notwendig, sondern würde lediglich die Neugier befriedigen. Notwendig ist lediglich, herauszuarbeiten, wie man die Ausnahme-Behandlung, die die modernen Programmiersprachen zur Verfügung stellen, in eigenen Assemblermodulen nutzen kann. Und dazu reicht es eigentlich aus, zu eruieren, was für Code der Compiler erzeugt, wenn er die Schlüsselworte kompiliert, mit denen die objektorientierte Ausnahme-Behandlung realisiert wird.



In Delphi dienen hierzu die Schlüsselworte »try« und »except«, die die »Try-Except-End«-Konstrukte ermöglichen. Hierbei stehen zwischen *try* und *except* die Anweisungen, die überwacht werden sollen, und zwischen *except* und *end* die, die abgearbeitet werden sollen, wenn eine Exception auftritt. Bitte beachten Sie hierbei, dass try und except *nicht* die Delphi-Versionen der C-Schlüsselworte __try und __except sind! Daher ist der »Stack-Rahmen«, den diese Anweisungen erzeugen und verwenden, nicht der gleiche, den das Betriebssystem zur rahmenbasierten Ausnahmebehandlung erwartet!

Da Try-Except-Blöcke und somit die zum Einsatz kommenden Handler ineinander verschachtelt werden können, existiert für jede »Ebene« eine Datenstruktur auf dem Stack mit identischem Aufbau: Basisadresse des Stack-Rahmens der Routine, in der die Ausnahmebehandlung erfolgt; die Adresse des zu verwendenden Anwendungshandlers und ein Link zu der nächst»höheren« Ebene. Neben dieser Datenstruktur auf dem Stack gibt es eine Adresse, die den »Einstiegspunkt« in die ver-

kettete Liste der Stack-Rahmen beinhaltet. Es handelt sich um FS:$0000_0000. Hier steht die Stackadresse, an der die Datenstruktur der Standard-Behandlungsroutine steht. Wurde vom Anwendungsprogrammierer keine Exception-Behandlung programmiert, wird somit die Standard-Behandlungsroutine, die Delphi zur Verfügung stellt, aufgerufen. Anwendungsprogrammierer können sich nun in diesen Mechanismus mittels zweier Konstrukte einklinken: mit einem

- Try-Except-End-Block, bei dem zwischen *except* und *end* Anweisungen stehen, die die Exception behandeln. Hier kann keine Fallunterscheidung anhand der aufgetretenen Exception gemacht werden. Will man dies, so muss man die andere Form verwenden: einen
- Try-Except-End-Block, bei dem zwischen *except* und *end* »On-Do-Anweisung«-Blöcke und ein optionaler Else-Teil stehen.

try Beide Formen unterscheiden sich nicht unerheblich, wie wir noch sehen werden. Zunächst soll am einfachsten Fall,

```
try
   überwachte Anweisungen
except
  Anweisungen im Ausnahmefall
end;
```

gezeigt werden, was der Compiler aus *Try-Except-End* macht:

```
       XOR     EAX, EAX
       PUSH    EBP
       PUSH    @HANDLER
       PUSH    DWORD PTR FS:[EAX]
       MOV     FS:[EAX], ESP
; hier folgen die überwachten Anweisungen
```

Als Erstes wird mittels XOR EAX, EAX das EAX-Register gelöscht, um den Offset in das mittels FS selektierte Segment zu erstellen, an der das Ende der Link-Einträge steht. Wie bereits gesagt, zeigt der Inhalt dieser Speicherstelle auf die Stackadresse, an der die Datenstruktur des jeweils aktuellen, »innersten« Try-Except-Blocks verzeichnet ist. An dieser Stelle ist das noch die von Delphi vorgesehene Standardbehandlung.

Dann wird der aktuelle Zeiger auf die Stackbasis (EBP) auf den Stack geschoben. Dadurch wird ein Zeiger auf den Stack-Rahmen gerettet, der zu der Routine gehört, die den neuen »innersten« Exception-Handler enthält. Dies erfolgt unter anderem aufgrund der Notwendigkeit

zum local und global unwind sowie aufgrund der Möglichkeit des Unterprogramm-Aufrufs, damit jeder Teil der Behandlungsroutinen zu jedem Zeitpunkt den Stack-Rahmen der Ausnahme-Behandlungsroutinen feststellen kann – schließlich richten ja in solchen Blöcken, aber auch im Rahmen der Exception-Behandlung aufgerufene Unterprogramme einen eigenen Stack-Rahmen ein und verändern damit EBP.

Als Nächstes wird dann die Adresse des Exception-Handlers auf den Stack gebracht, der aufgerufen werden soll, wenn ein Fehler auftritt. Genauer gesagt: Es wird die Adresse gePUSHt, an der ein JMP-Befehl auf eine Routine steht, die Delphi zur Verfügung stellt.

Schließlich wird noch die zurzeit in FS:$0000_0000 befindliche Adresse der auf dem Stack befindlichen Datenstruktur des nun »übergeordneten« Try-Except-Blocks auf den Stack gerettet und mit dem aktuellen Inhalt des ESP-Registers, also der aktuellen Stackspitze, belegt. Dadurch wurde der Link zwischen der nun eingerichteten, »inneren« Behandlungsroutine mit der »nächsthöheren«, hier der Standardroutine, hergestellt.

Soweit die Definition und »Verkabelung« eines Try-Blocks in eine rekursive Exception-Hierarchie. Bevor wir sehen werden, wie eine solche Hierarchie auf dem Stack aussieht, schauen wir uns erst einmal an, wie es im »einfachen« Fall eines nicht verschachtelten Blocks weitergeht. Es schließt sich unmittelbar der Block mit den überwachten Anweisungen an, die dann ausgeführt werden.

Das sich an die überwachten Anweisungen anschließende »except« übersetzt der Compiler zu

except

```
        XOR     EAX, EAX
        POP     EDX
        POP     ECX
        POP     ECX
        MOV     FS:[EAX], EDX
        JMP     ENDE
HANDLER:JMP     @HANDLEANYEXCEPTION
; hier folgen die im Ausnahmefall auszuführenden Anweisungen
```

»Except« erzeugt also zunächst einmal Code, der den regulären Ablauf des Programms abschließt, also dann abgearbeitet wird, wenn innerhalb des Blocks überwachter Ausdrücke *kein* Fehler aufgetreten ist: Mittels XOR EAX, EAX wird das EAX-Register gelöscht, nur dient das in diesem Fall dazu, den Code »0« in EAX zurückzugeben. Der durch *try*

angelegte Stackbereich wird nun abgebaut: Die gerettete Adresse der Datenstruktur des »übergeordneten« Try-Except-Blocks wird zunächst in EDX gePOPpt um dann zwei Befehle weiter an die bekannte Speicheradresse FS:$0000_0000 zurückgespeichert zu werden. Die folgende Handleradresse und der Inhalt von EBP werden via POPpen in ECX in die Ewigen Jagdgründe geschickt: Sie werden nicht mehr gebraucht. Schließlich wird »hinter« das »end« gesprungen, das den Except-Block abschließt, und dieser Block damit umgangen. Dies erfolgt durch einen unbedingten Sprung.

Der eigentliche Except-Teil nun wird durch einen JMP-Befehl mit Ziel @HandleAnyException eingeleitet. Die Adresse dieses Befehls wurde ja in die Datenstruktur aufgenommen, die auf dem Stack liegt, und wird im Falle einer Exception aufgerufen. Sie verzweigt dann ihrerseits wieder hierher zurück, um die sich anschließenden Anweisungen auszuführen, die der Anwendungsprogrammierer vorgesehen hat: die im Except-End-Block aufgeführten Anweisungen. Das den Except-Block abschließende »end« fügt nur noch eine Zeile hinzu:

```
        CALL    @DoneExcept
Ende:   :       ; hier geht's normal weiter
```

Dieser Aufruf der Routine DoneExcept räumt alles wieder auf, was im Rahmen der Exception-Behandlung so angefallen ist – unter anderem den von der Try-Einleitung belegten Stack. Sie ist jedoch auch für die Zerstörung der Exception-Objekte zuständig. Diese Routine kommt nur zum Tragen, wenn während der Abarbeitung der zwischen Try und Except stehenden Anweisungen eine Ausnahme aufgetreten sein sollte, die durch HandleAnyException und/oder die vorgesehenen Anweisungen behandelt wurde. Und fertig ist die Ausnahmebehandlung! Der Ausnahmeverteiler des Betriebssystems betrachtet nun die Ausnahme als erledigt und übergibt die Verantwortung wieder dem Prozessor, der nun je nach Exception-Typ (trap, fault, abort) mit dem gleichen oder nächsten Befehl des unterbrochenen Programms fortfährt – oder im schlimmsten Fall das Programm endgültig beendet.

on ... do ...

Der eben beschriebene Mechanismus behandelt alle auftretenden Exceptions gleich. Delphi bietet aber auch die Möglichkeit, Exceptions einzeln und gezielt bearbeiten zu können. Hierzu erlaubt es im Except-Block eine Liste von On-Do-Blöcken, die die einzelnen Exceptions bearbeiten sollen und einen optionalen, für die nicht direkt behandelten Exceptions dienenden Else-Teil, etwa:

```
try
   überwachte Anweisungen
except
   on EMathError do Irgendetwas;
   on EInvalidOp do IrgendetwasAnderes;
   else do Sonstwas;
end;
```

Bis zum Label Handler bleibt der Code gleich, der durch den Compiler aus »except« generiert wird. Dann aber folgt im Except-Compilat:

```
Handler:jmp    @HandleOnException
; anstelle der im Ausnahmefall auszuführenden Anweisungen
; steht hier eine Liste mit Adressen, gefolgt von Anweisungen
        DD     NumberOfChecks
        DD     Adresse_EMathError_Object
        DD     Adresse_EMath
        DD     Adresse_EInvaildOp_Objects
        DD     Adresse_EInval
        DD     $0000_0000
        DD     Adresse_Rest
EMath:          ; Irgendetwas
        JMP    Done
EInval:         ; IrgendetwasAnderes
        JMP    Done
Rest:           ; Sonstwas
Done:   CALL   @DoneExcept
Ende:           ; hier geht's normal weiter
```

Nach Done: folgt dann wieder das obligatorische CALL @DoneExcept, das aus der Übersetzung des abschließenden »end« kommt.

Der hier involvierte Delphi-Handler ist ein anderer: HandleOnException. Er nutzt den Platz zwischen seinem eigenen Aufruf mittels JMP und dem Aufruf CALL @DoneExcept nicht als Reihe von auszuführenden Anweisungen, sondern als Liste, in der erstens die Anzahl zu prüfender Exception-Objekte (On-Blöcke bzw. Else) steht (hier 3: zwei Exceptionobjekte und ein »else«) sowie zweitens für jedes zu prüfende Exception-Objekt jeweils die Adresse seiner Datenstruktur und der Anweisungen, die dem Objekt (hinter dem Schlüsselwort »Do«) zugeordnet wurden. Der Spezialfall »Else« hat als Adresse für die Datenstruktur den Wert »0«, aber einen eigenen Anweisungsteil. Dieser Liste schließt sich dann für jede On-Do-Anweisung die dazugehörige auszuführende Anweisung an.



Delphi gestattet ja, in »On – Do«-Blöcken neben dem Exception-Objekt auch einen »Bezeichner« der Form *On E : Exception-Objekt Do*. Dieses zusätzliche »E :« (für alle, die keine Erfahrung mit Delphi haben, lies: »E vom Typ«) fordert der Compiler ausschließlich aus semantischen Gründen, um die Adresse des Exception-Objekts in der nach »Do« folgenden Anweisung (und nur dort!) verfügbar zu machen. Dies ist notwendig, wenn z.B. eine Routine aufgerufen wird, der diese Adresse übergeben werden soll. Das Compilat sieht jedoch in beiden Fällen, also mit oder ohne »E :«, identisch aus. Insbesondere werden nicht etwa lokale Variablen auf dem Stack eingerichtet oder sonstige Register oder Speicherstellen belegt.

verschachtelte Try-Except-Blöcke

Soweit der »einfache« Fall des Exception-Handlings auf Anwenderseite. Try-Except-Blöcke können aber, wie bereits angesprochen, auch verschachtelt werden. Das bedeutet, dass die Exception-Behandlung auf verschiedenen Ebenen durch verschiedene Handler erfolgen kann. An dieser Stelle werden wir nicht genauer darauf eingehen, da für unsere Assemblerbelange die einfachen, unverschachtelten Exception-Handler ausreichen.

raise

Aus diesem Grunde werden wir hier auch nicht die Compilate anderer wichtiger Mechanismen sezieren, die im Rahmen der Ausnahmebehandlung eine Rolle spielen, wie z.B. das anwenderspezifische Auslösen von Exceptions (also Software-Exceptions) mittels Raise. Falls Sie dies in einem Assemblermodul benötigen, können Sie ja Raise dort als external deklarieren und dann aufrufen!



Die gezeigten Disassemblate zeigen die mit Delphi 5.0 erstellten Compilate. Sie unterscheiden sich, wenn überhaupt, nur unwesentlich von Compilaten, die mit Delphi 3.0, Delphi 4.0 oder Delphi 6.0 erzeugt werden. Die Unterschiede beziehen sich lediglich darauf, dass bei der Erstellung des Stack-Rahmens bei Delphi 3.0 und Delphi 4.0 das Register EDX anstelle von EAX zur Adressierung von FS:$0000_0000 herangezogen wird, dass bei Delphi 3.0 beim regulären Abschluss des überwachten Anweisungsblocks der Link auf dem Stack direkt (*POP dword ptr FS:00000000h*) und nicht über den Umweg EDX zurückgeschrieben wird und dass ebenfalls bei Delphi 3.0 der Stack durch Addition der Konstanten $08 zu ESP abgeräumt wird anstelle der POPs ins Nirwana via ECX – was im Ergebnis das Gleiche ist. Wie Sie sehen: Keine grundsätzlichen Unterschiede.



Auch C++ muss aufgrund der Realisierung objektorientierter Ansätze einen größeren Aufwand treiben, als es das Betriebssystem von einem anwenderseitigen Exception-Handler eigentlich erwartet. Daher erfolgt der Aufbau des Rahmens für die Exception-Behandlung prinzipiell unter C++ genauso wie unter Delphi – im Detail jedoch etwas anders: C++ wäre nicht C++, hätte es nicht eigene Vorstellungen.



Ich betrachte hier Code, der vom C++-Builder erzeugt wurde. Mit Visual C++ läuft das aber, soweit ich feststellen konnte, analog. Daher sollten auch alle Anhänger von Microsofts C++-Compiler mit den hier gegebenen Informationen weiterkommen.

Man kennt auch unter C++ zwei verschiedene Arten, wie auf Exceptions reagiert werden kann. So gibt es einen

- Try-Catch-Block, bei dem im Catch-Teil Anweisungen stehen, die die Exception behandeln. Hier kann analog der »einfachen« Except-Anweisung unter Delphi keine Fallunterscheidung anhand der aufgetretenen Exception gemacht werden. Will man dies, so muss man die andere Form verwenden: einen
- Block, in dem auf ein einzelnes Try mehrere Catch-Anweisungen folgen können, bei denen jeweils in einer Catch-Anweisung eine bestimmte zu definierende Exception behandelt wird. Dies entspricht den On-Do-Blöcken in Delphi.

try

C++ kennt also ebenfalls die Anweisung »try«, die in Delphi den Aufbau des Stack-Rahmens für den Ausnahmeverteiler übernimmt.

```
try {
// überwachte Anweisungen
}
catch(...) {
// Anweisungen im Falle des Auftretens einer Exception
}
```



Und schon geht's los mit den Unterschieden: Die Einrichtung des Stacks der Routine erfolgt in C++ anders als in Delphi. In Delphi erzeugt der Compiler mittels des *begin*, mit dem jede Routine beginnt, einen Stack-Rahmen, der durch das die Routine abschließende *end* wieder abgebaut wird. Innerhalb der Routine verändert dann die Try-Anweisung die Stackgröße anhand ihrer Bedürfnisse. Daher muss die Except-Anweisung für den Fall nicht auftretender Exceptions Code generieren, der den durch Try veränderten Code wieder auf die Größe

schrumpfen lässt, die das abschließende *end* erwartet und die das *begin* vorgegeben hat. C++ geht einen anderen Weg: Die öffnende Klammer zu Beginn einer Routine erzeugt einen fertigen Stack-Rahmen, der durch Try nicht mehr verändert wird. Dies hat den Vorteil, dass die schließende Klammer am Ende der Routine den Stack aufräumen kann. Das bedeutet jedoch, dass bereits zum Zeitpunkt der Herstellung des Stack-Rahmens die Größe des benötigten Stacks bekannt sein muss.



Um keine Überraschungen zu erleben, sollten Sie daher mittels C++ eine Dummy-Routine kompilieren, die die Struktur aufweist, die später die Assemblerroutine haben soll. Übernehmen Sie die so erzeugte Hülle der Routine in Ihren Assemblerquelltext und füllen Sie sie mit den gewünschten Assemblerbefehlen.

Doch es gibt weitere Unterschiede. »Try« selbst übersetzt der Compiler hier in einen Teil Code, der unmittelbar nach dem Einrichten des Stack-Rahmens für die Routine angefügt wird, und in einen Teil, der unmittelbar vor den zu überwachenden Anweisungen steht. Dazwischen stehen die Anweisungen, die vor dem Try-Block angesiedelt sind:

```
       MOV     EAX, Adresse#1
       CALL    __InitExceptBlockLDTC(void *)
; hier stehen die **VOR** dem TRY befindlichen Anweisungen
       MOV     WORD PTR [EBP-0X14], 0x0008
; hier beginnen die zu überwachenden Anweisungen
```

C++ überlässt den Stackaufbau, genauer: die Belegung der bereits reservierten Stellen auf dem Stack, einer spezialisierten Routine namens InitExceptBlockLDTC, der als Argument in EAX die für unsere Belange nicht weiter wichtige Adresse eines Exception-Objekts übergeben wird. Das heißt, dass im Unterschied zu Delphi hier alle Anweisungen mit einem bereits für die Exception-Behandlung vorbereiteten Stack ablaufen. Die zu überwachenden Anweisungen liegen in einem Block, der mit der Belegung einer Variablen auf dem Stack ([ebp-0x14]) mit der Konstanten 0x0008 eingeleitet ...

catch (...) ... und mit dem Belegen der gleichen Variablen mit dem Wert 0x0000 abgeschlossen wird:

```
       MOV     WORD PTR [EBP-0X14], 0x0000
       JMP     @Ende
; hier folgen die im Ausnahmefall auszuführenden Anweisungen
       MOV     WORD PTR [EBP-0x14], 0x0010
       CALL    __CatchCleanup()
```

```
Ende:   MOV     EDX, [EBP-24]
        MOV     FS:[0x00000000], EDX
```

In diesem Beispiel wurde das C++-Pendant zu Delphis »except« eingesetzt. Es heißt »catch«. Auch hier wurde zunächst der Fall betrachtet, dass alle möglichen Exceptions im Rahmen eines einzigen Anweisungsblocks behandelt werden sollen, also mittels der Anweisung »catch (...)«

Ist im überwachten Teil keine Exception aufgetreten, so wird schlicht und ergreifend zu einer Stelle gesprungen, an der analog zu Delphi der auf dem Stack gerettete Link zum »übergeordneten« Try-Catch-Block in die Adresse FS:$0000_0000 zurückgeschrieben wird. Dieser war durch die Routine InitExceptBlockLDTC auf den Stack gerettet worden.

Im Ausnahmefall dagegen sorgt die in InitExceptBlockLDTC bekannt gegebene Adresse des C++-Handlers dafür, dass die spezifizierten Anweisungen ausgeführt werden. In diesem Fall wird auch die Routine __CatchCleanup aufgerufen, die Ähnliches zu bewerkstelligen hat wie DoneExcept unter Delphi. Auch in diesem Fall wird der Bezug zum »übergeordneten« Block wieder gelöst.



Da C++ grundsätzlich andere Wege bei der Verwaltung eines Stacks für Routinen geht als Delphi und die Einrichtung des Rahmens für die Fehlerbehandlung hier gleich zu Beginn bei der generellen Einrichtung des Stacks erfolgte, muss unter C++ nicht extra ein Stackabbau erfolgen. Dies erledigt der die Routine abschließende Code (»Epilog«), der den Stack zerstört, quasi automatisch mit.

InitExcept-BlockLDTC

Diese Routine möchte ich, genau wie __CatchCleanup, nicht detaillierter besprechen, weil beides nicht wesentlich für das Verständnis ist. Nur so viel: Es wird unter anderem ein der Delphi-Version vergleichbarer Stack-Rahmen mit Informationen gefüllt, so z.B. ein Link zum »übergeordneten« Try-Catch-Block, eine im Exceptionfall aufzurufende C++-Handleradresse und ein Zeiger auf die aktuelle Stackbasis.

catch ( )

Vergleichbar mit den »On-Do«-Blöcken in Delphi kann auch unter C++ eine Exception-Behandlung für einzelne Exceptions erfolgen. Anders als bei Delphi erfolgt das jedoch nicht im Rahmen eines »Catch«-Blocks, sondern durch die Aneinanderreihung verschiedener Catch-Anweisungen:

```
try {
// überwachte Anweisungen
}
catch(const EMathError&) {
  Irgendetwas
}
catch(const EInvalidOp&) {
  IrgendetwasAnderes
}
```

Der C++-Compiler macht daraus folgenden Code. Uns interessiert hierbei nur der auf Try folgende Teil, der sich folgendermaßen darstellt:

```
       MOV     WORD PTR [EBP-0x18], 0x0008
       JMP     @Ende
EMath: Irgendetwas
       JMP     @Clean
EInval: IrgendetwasAnderes
Clean: MOV     WORD PTR [EBP-0x18], 0x0010
       CALL    _CatchCleanup
 Ende: MOV     EDX, [EBP-0x28]
       MOV     FS:[0x0000], edx
; hier geht's normal weiter
```

Zunächst fällt auf, dass im Vergleich zu der »einfachen« Form von eben die auszuführenden Befehle in entsprechenden Blöcken aneinander gereiht werden, die alle (bis auf den letzten natürlich) mit einem unbedingten Sprung an die Stelle beendet werden, an der _CatchCleanup aufgerufen wird. Dies ist auch absolut richtig so. Verglichen mit der Delphi-Realisation vermissen wir jedoch die Liste mit den Adressen der Exception-Objekte und der Behandlungsroutinen.

__except

Auch unter C++ ist die strukturierte Ausnahmebehandlung von C nutzbar. Sie wird ebenfalls mittels »Try« eingeleitet, jedoch erfolgt die Ausnahmebehandlung in einem Block, der mit __except eingeleitet wird. __except erwartet als Argument den Rückgabewert eines Exceptionfilters. Gibt man hier den Wert »Exception_Execute_Handler« an, so wird ein Compilat erzeugt, dessen disassemblierte Form so aussieht:

```
       MOV     WORD PTR [EBP-0x14], 0x0000
       JMP     @Ende
; hier folgen die im Ausnahmefall auszuführenden Anweisungen
       MOV     WORD PTR [EBP-0x14], 0x0010
Ende:  MOV     EDX, [EBP-24]
       MOV     FS:[0x00000000], edx
; hier geht's normal weiter
```

Wie Sie sehen können, ist der einzige Unterschied zum Catch(...)-Fall der, dass die Routine __CatchCleanup (die unter anderem für die Zerstörung der Exception-Objekte zuständig ist) nicht aufgerufen wird! Tatsächlich werden try und catch (und auch die anderen, hier nicht besprochenen und relevanten Anweisungen wie throw) in die entsprechenden C-Anweisungen __try, __except (und RaiseException) »übersetzt«.

So weit die Hintergrundinformationen, die ich Ihnen im Rahmen dieses Buches geben kann. Wie Sie sicherlich gemerkt haben, bleibt noch jede Menge Raum für eigene Spaziergänge und Experimente. Ich möchte daher das Kapitel abschließen mit einem kleinen »Rezept«, wie die SEH in Assemblermodulen genutzt werden kann.

- Denken Sie daran, dass die SEH in unterschiedlichen Hochsprachen (Delphi, C++) und eventuell auch unterschiedlichen Dialekten (C, C++) und gar Versionen unterschiedlich realisiert wird, auch wenn der generelle Aufbau in der Regel der gleiche, zumindest aber sehr ähnlich ist. Das bedeutet, dass Sie nicht ein »generelles« Modul programmieren können, das universell einsetzbar ist. Die SEH ist ein Feature der Hochsprache, nicht des Assemblers und auch nicht des Betriebssystems!
- Prüfen Sie, ob Sie das Assemblermodul nicht in der Weise realisieren können, dass innerhalb der entsprechenden Routine mit den Try-Except-Blöcken nur Assembler-Routinen aufgerufen werden, die SEH selbst jedoch im Rahmen der Hochsprachenroutine erfolgt.
- Ist das nicht möglich oder erwünscht, erzeugen Sie mit der Hochsprache eine Dummy-Routine, die die einzelnen Elemente der SEH berücksichtigt, die Sie benötigen (»einfaches« except oder On-Do- bzw. Catch-Blöcke). Markieren Sie den jeweiligen Beginn des durch die SEH »verwalteten« Codes, indem Sie einen Befehl eingeben, der im Assemblat leicht identifiziert werden kann. Dies ist leicht mit dem integrierten Assembler möglich. Ich benutze gerne `XOR ESI, ESI` – es macht keine Fehler, falls ich »aus Versehen« das Programm starte statt debugge, und sieht so schön exotisch aus!
- Rufen Sie nun den integrierten Debugger auf, indem Sie den Cursor in der IDE auf den Beginn der Routine setzen und mittels F4 das Programm bis zum Cursor ausführen lassen. Wechseln Sie dann in die CPU-Anzeige.
- Hier sehen Sie nun die einzelnen Assembler-Befehle, die der Compiler erzeugt hat. Notieren Sie sich alle Befehle, die in der Routine stehen.

- Wechseln Sie in den Assembler-Quelltext und geben Sie diese Instruktionen ein. Ersetzen Sie dann Ihre »Marke« (bei mir, wie gesagt, `XOR ESI, ESI`) durch die Assemblerbefehle, die Sie programmieren möchten.
- Vergessen Sie nicht, die benötigten Unterprogramme im Assembler-Modul als EXTRN zu deklarieren.

Denken Sie immer daran, dass diese Konstruktion spezifisch für den aktuellen Assembler und die aktuelle Hochsprache in der aktuellen Version korrekt arbeitet. Das muss mit einem anderen Assembler und einer anderen Hochsprache oder anderen Version nicht notwendigerweise so sein! Dokumentieren Sie daher diese Routine gut und denken Sie daran, dass bei Re-Kompilierung mit einer neuen Version des Compilers ggf. ein neues »Interface« erforderlich wird.

4 Der Integrierte Assembler

4.1 Programmierung mit dem Inline-Assembler

Wie ich bereits bei der Besprechung des Stand-Alone-Assemblers festgestellt habe, gibt es eigentlich keinen Grund, diesen nicht ausgiebig zu nutzen. Dennoch möchte ich nicht leugnen, dass die Inline-Assembler auch einen gewissen Charme haben. So ist es wirklich sehr elegant, einfach im Rahmen eines Hochsprachenprogramms ein paar Assemblerbefehle einzustreuen und auf diese Weise die Fähigkeiten dieser mächtigen Sprache zu nutzen, ohne »umständlich« einen Editor starten, den ganzen Verwaltungsaufwand mit Segmentdeklaration und Modellangabe betreiben, den Assembler bemühen und Fehler ausmerzen zu müssen, bevor man das erzeugte Modul mittels des geeigneten Befehls dann in die Hochsprache integrieren kann.

Dieser Charme war jedoch bis vor kurzem sehr herb. Denn die Inline-Assembler unterstützten bislang »nur« den Befehlssatz von 80386ern, und auch den nicht vollständig. Das führte dazu, dass z.B. der so wichtige Befehl CPUID in Hochsprachen nur über den Umweg der Programmierung »von Hand« mit DB-Direktiven möglich war.

Zumindest mit Delphi 6.0 hat sich das geändert. Diese Hochsprache verfügt nun über einen Inline-Assembler, der die Befehle des Pentium III und Athlon beherrscht und somit auch die SIMD- und 3DNow!-Befehle. Kleines Wermuts-Tröpfchen: Den Pentium 4 mit seinen SSE2-Befehlen kennt Delphi 6.0 noch nicht. Aber das liegt einfach in der Natur der Sache und dem damit begründeten zeitlichen Unterschied bei der Entwicklung von Hardware und der dazugehörigen Software.

Dennoch möchte ich festhalten, dass der Inline-Assembler in keiner Weise einen Stand-Alone-Assembler ersetzen kann. Wenn Sie eine Weile mit beiden gearbeitet haben, werden Sie mir zustimmen.

Was den C++-Builder von Borland betrifft: eine gute und eine schlechte Nachricht. Die Schlechte: Die aktuelle Version 5.0 setzt auf TASM32, Version 5.3 auf. Und diese Version kennt gerade einmal die 80386-Befehle. Die gute Nachricht: Version 6.0 steht vor der Türe. Und es ist davon auszugehen, dass Borland/Inprise einerseits nicht hinter Microsoft anstehen und andererseits sich nicht selbst Konkurrenz (Delphi 6.0!) machen möchte. Daher ist davon auszugehen, dass C++-Builder 6.0 auch über einen Inline-Assembler verfügt, der dem von Delphi in nichts nachsteht.

Visual C++ 7.0 konnte ich leider nicht testen, da mir die Beta-Version während der Drucklegung des Buches nicht lauffähig zur Verfügung stand. Man hört, dass nun mit Visual C++ 7.0 ebenfalls die Prozessor-Befehle moderner Prozessoren unterstützt werden. Ist der verwendete Assembler ein Kriterium, scheint das zu stimmen. Denn das mitgelieferte ML (früher: MASM) in der Version 7.0 kennt alle Befehle, sogar die SSE2-Befehle des Pentium 4.



Eine kleine Anmerkung in eigener Sache! Nachdem mir leider weder von Borland noch von Microsoft die »neuen« Compiler zur Verfügung standen, kann ich zurzeit keine verlässlichen Aussagen zu deren Möglichkeiten und Grenzen machen, wie ich das für Delphi 6.0 tun kann. Bitte sehen Sie mir daher nach, wenn ich mich bei der folgenden Betrachtung des Inline-Assemblers auf den von Delphi 6.0 beschränke. Ich gehe davon aus, dass der von C++-Builder 6.0 analog dem von Delphi arbeitet (schließlich stammt er aus der gleichen Schmiede) und dass kein allzu großer Unterschied zu dem von Visual C++ 7.0 besteht. Ich gehe daher weiter davon aus, dass Sie als erfahrener C++-Programmierer, wenn überhaupt, keine allzu großen Probleme damit haben werden, das auf den jeweiligen C++-Compiler umzumünzen, was ich im Folgenden zu Delphis sagen werde. Ich habe lange überlegt, ob ich entsprechende Aussagen zu den alten Versionen machen sollte, entschied mich dann aber dagegen: Wenn nun schon eine überarbeitete Auflage sich am status quo orientiert (Pentium 4, 32-Bit-Umgebungen, Delphi 6.0), macht es wenig Sinn, »Altes« aufzuwärmen. Denn immerhin stehen die neuen Versionen ja vor der Türe, und das »Alte« beschreibt den

Zustand zum Zeitpunkt des 80386 – zumindest was den Inline-Assembler betrifft. Zu alt für meine Begriffe!

Schmalspur-Assembler

Es gibt ein paar Dinge zu beachten, wenn Sie inline assemblieren! Wenn Sie im vorangehenden Kapitel über den Stand-Alone-Assembler Blut geleckt haben, folgt gleich ein Dämpfer! Auch in der neuesten Version 6.0 von Delphi (hier als Platzhalter auch für die kommenden Versionen von C++ von Borland und Microsoft!) ist der Inline-Assembler kein Makro-Assembler! Sie können somit das gesamte Kapitel über Makros und ihre Möglichkeiten glatt vergessen! Des Weiteren dürfen Sie getrost alles ignorieren, was über Deklarationen gesagt wurde: Sei es über die Deklaration neuer Typen, sei es über die Allozierung von Daten! Der Inline-Assembler gestattet keinerlei Deklaration, er kennt »nur« die Typen und Daten, die in den Hochsprachen deklariert und alloziert wurden.



Ausnahme: Der Inline-Assembler kennt sehr wohl die Direktiven DB, DW, DD und DQ. ABER! Sie dienen nicht etwa dazu, Daten zu allozieren. Das, wie gesagt, ist nicht möglich. Vielmehr können sie dazu verwendet werden, einzelne Bytes, Words oder DoubleWords in das *Codesegment* einzustreuen. Wichtig ist dies, damit Sie auch Byte-Folgen *als Instruktionen* generieren können, die der Inline-Assembler (noch) nicht kennt, so z.B. die SSE2-Befehle. Da bereits der Inline-Assembler von Delphi 5.x oder C++-Builder 5.0 dies konnte, konnte auch unter diesen Hochsprachen-Versionen mit dem Befehl CPUID und anderen gearbeitet werden: via Programmierung über DB.



Unter Delphi 5.0 und C++-Builder 5.0 war nur die Direktive DB erlaubt! Und selbst die kannte aber Visual C++ nicht. In Microsofts Compiler musste an ihrer Stelle die Pseudo-Instruktion »_ _emit« verwendet werden, die ein Byte in den Code einstreute. Ich bin selbst gespannt, wie das unter Visual C++ 7.0 gelöst werden wird.



Wenn Sie nun aber glauben, mit Hilfe der Direktiven Daten im Codesegment ansiedeln zu können: Weit gefehlt! Weder der Compiler von Delphi noch das Betriebssystem lässt Ihnen das durchgehen. Delphi-Codesegmente sind in der Regel mit dem Attribut *execute-only* versehen, können also nur ausgeführt, nicht aber gelesen werden. Und beschrieben schon gleich gar nicht: Es gibt definitiv keine beschreibbaren Codesegmente!



Warum ist eigentlich der Inline-Assembler mit den Direktiven so störrisch? Sie könnten ja auch für das Datensegment zugelassen werden! Ganz einfach: Jede Hochsprache hat ihre eigene Art, mit Datensegmenten umzugehen. Manche erlauben und verwenden initialisierte Datensegmente, andere nicht. Manche arbeiten mit Datensegmenten, die nur über FAR-Zeiger angesprochen werden können, bei andern ist alles »flat«, sprich »near«. Das bedeutet, dass die Hochsprache – besser: der Compiler – die Kontrolle behalten muss. Und in diesem Fall stören die Eigenmächtigkeiten, die der Stand-Alone-Assembler dem Programmierer an die Hand gibt (ASSUME, .DATA, .DATA?, .FARDATA, .FARDATA?), gewaltig. Daher der radikale Cut: Inline gibt es nur Befehls-»Daten«, die per Direktive eingestreut werden können.



Das hat z.B. unter Delphi weitere Konsequenzen. Sie können beispielsweise nicht Prozeduren mittels PROC deklarieren oder Annahmen über Segmente mittels ASSUME machen! Falls Sie Assembler-Routinen programmieren wollen, geht das nur innerhalb eines Delphi-Gerüstes.

Das bedeutet: Sie programmieren eine ganz »normale« Rumpf-Routine mittels

```
function ASMfunc(var Param:Integer):Integer;
begin
end;

procedure ASMproc(var Param:Integer);
begin
end;
```

In diesen Rumpf können Sie nun Delphi- und/oder Assembler-Instruktionen einbauen. Die Assembler-Instruktionen müssen dann analog *begin – end* mittels *asm – end* geklammert werden:

```
function ASMfunc(var Param:Integer):Integer;
begin
   // hier könnten Delphi-Instruktionen stehen
   asm
   // hier stehen Assembler-Instruktionen
   end
   // hier könnten wiederum Delphi-Instruktionen stehen
end;
```

```
procedure ASMproc(var Param:Integer);
begin
   asm
      // hier stehen Assembler-Instruktionen
   end
end;
```

Das bedeutet: Die allgemeine Form der Nutzung von Assembler-Instruktionen in Delphi-Quelltexten erfolgt formal nach



```
asm
        Anweisung
        :
end;
```

Unter C++ stellt sich die Situation formal ebenso dar:



```
__asm
{
        Anweisung
        :
}
```

Beachten Sie bitte, dass dem Schlüsselwort »asm« zwei Unterstriche vorangestellt sind.

»Inline« »Assembler«

Vielleicht kennen Sie aus früheren Delphi-Versionen noch die Schlüsselworte *Inline* und *Assembler*. Diese sind unter Delphi 6.0 zwar noch definiert, sie zeigen jedoch keine Wirkung mehr. Daher brauchen Sie künftig nicht mehr das Schlüsselwort Assembler hinter einen Prozedur-Kopf zu schreiben, wenn Sie Assembler-Routinen deklarieren wollen. Das geht nun so:

Falls innerhalb einer Prozedur außer Assembler-Befehlen keine Delphi-spezifischen Instruktionen stehen, können Sie anstelle der Ein- und Ausleitung der Routine mittels *begin – end* mit eingeschlossenem *asm – end* ausschließlich *asm – end* verwenden:

```
function ASMfunc(var Param: Integer): Integer
asm
        :
        :
end;
```

```
procedure ASMproc(var Param: Integer);
asm
        :
        :
end;
```

Worin besteht der Unterschied? Im Code, den der Compiler als Prolog und Epilog der Routine erzeugt. Fangen wir langsam an.

Prolog und Epilog

Der Compiler erzeugt bei jedem *begin*, mit dem eine Routine eingeleitet wird, einen Prolog für die Routine. Darunter versteht man die Routine »einleitenden« Code. Jedes die Routine abschließende *end* erzeugt analog einen Epilog, also Code, der die Routine beendet. Je nachdem, ob nun eine Funktion oder Prozedur deklariert wird, ob dieser Parameter übergeben werden und/oder lokale Variablen deklariert, sehen Prolog und Epilog unterschiedlich aus:

```
procedure SimpleProc;
begin
  messagebeep(0);
end;

procedure ParamProc(var I, J: Integer);
begin
  I := 4;
  J := 5;
end;

procedure LocalVarProc;
var I, J : Integer
begin
  I := 4;
  J := 5;
end;

 procedure LocalVarAndParamProc(var I, J: Integer);
var K, L : Integer
begin
  K := I;
  L := J;
end
```

Listing 1: Delphi-6.0-Quelltext ohne Assembleranweisungen

Wenn Sie diese Zeilen einmal innerhalb eines Delphi-Programms mit dem integrierten Debugger und dem CPU-Fenster debuggen, so finden Sie ein Disassemblat, das etwa so aussieht:

```
; keine Parameter, keine lokalen Variablen
push    $00                     ; Parameter auf den Stack
call    -$xxxxxxxx              ; MessageBeep aufrufen
ret                             ; end

; zwei Parameter
push    ebp                     ; begin
mov     ebp, esp
add     esp, -$08
mov     [ebp-$08], edx
mov     [ebp-$04], eax
mov     eax, [ebp-$04]          ; I := 4
mov     [eax], $00000004
mov     eax, [ebp-$08]          ; J := 5
mov     [eax], $00000005
pop     ecx                     ; end
pop     ecx
pop     ebp
ret

; zwei lokale Variablen
push    ebp                     ; begin
mov     ebp, esp
add     esp, -$08
mov     [ebp-$04],$00000004     ; I := 4
mov     [ebp-$08],$00000005     ; J := 5
pop     ecx                     ; end
pop     ecx
pop     ebp
ret

; ein Parameter, eine lokale Variable
push    ebp                     ; begin
mov     ebp, esp
add     esp, -$10
mov     [ebp-$08], edx
mov     [ebp-$04], eax
mov     eax, [ebp-$04]          ; K := I
mov     eax, [eax]
mov     [ebp-$0C], eax
mov     eax, [ebp-$08]          ; L := J
```

```
mov     eax, [eax]
mov     [ebp-$10], eax
mov     esp, ebp                ; end
pop     ebp
ret
```

Listing 2: Disassemblat eines vom Delphi-6.0-Compiler gemäß Listing 1 erzeugten Compilates

Ich will das nun nicht Schritt für Schritt kommentieren. In diesem Disassemblat verstecken sich einige Optimierungen des Compilers. So werden die Adressen der Variablen nicht über den Stack übergeben, sondern in den Registern EAX und EDX. Auch scheint Delphi 6.0 nun Parameter und lokale Variable nicht mehr wie in früheren Versionen via [ESP + Offset] anzusprechen, sondern C++-artig mittels [EBP – Offset], was natürlich das Gleiche ist, vorausgesetzt, man passt den Offset an. Auch wird der Stack-Rahmen in manchen Situationen durch die (schnelleren) POP-CX-Befehle aufgelöst anstelle des *MOV ESP, EBP*. Wie gesagt, darauf möchte ich hier nicht weiter eingehen! Was Sie aus dem Listing dagegen ersehen sollen, ist, dass jeweils der durch *begin* bzw. *end* erzeugte Prolog bzw. Epilog unterschiedlich aussehen. Analoges passiert, wenn Sie mit Funktionen arbeiten. Vielleicht macht es Ihnen Spaß, ein wenig zu experimentieren.

Assembler-Routinen

Funktionen und Prozeduren, die nicht mit *begin – end,* sondern ausschließlich mit *asm – end* geklammert werden, nennt Borland Assembler-Funktionen und -Prozeduren. Realisieren wir nun einmal spaßeshalber ein ähnliches Listing mittels Assembler-Routinen, etwa in:

```
procedure SimpleProc;
asm
  push  $00
  call  windows.messagebeep
end;

procedure ParamProc(var I, J: Integer);
asm
  mov   I, $04
  mov   J, $05
end;

procedure LocalVarProc;
var I, J: Integer
asm
  mov   I, $04;
```

```
  mov   J, $05
end;

procedure LocalVarAndParamProc(var I, J: Integer);
var K, L: Integer
asm
  mov    eax, I
  mov K, eax
  mov    eax, J
  mov L, eax
end
```

Listing 3: Delphi-6.0-Quelltext mit Assembleranweisungen

Wie Sie schnell erkennen, haben diese Routinen die gleiche Funktionalität wie die aus Listing 1. Schauen wir uns nun das Disassemblat an, so sehen wir allerdings Unterschiede.

```
; keine Parameter, keine lokalen Variablen
push   $00                    ; Parameter auf den Stack
call   MessageBeep            ; MessageBeep aufrufen
ret                           ; end

; zwei Parameter
mov    eax, $00000004         ; mov   I, $04
mov    edx, $00000005         ; mov   J, $05
ret                           ; end

; zwei lokale Variablen
push   ebp                    ; asm
mov    ebp, esp
add    esp, -$08
mov    [ebp-$04],$00000004    ; mov   I, $04
mov    [ebp-$08],$00000005    ; mov   J, $05
pop    ecx                    ; end
pop    ecx
pop    ebp
ret

; zwei Parameter, zwei lokale Variablen
push   ebp                    ; asm
mov    ebp, esp
add    esp, -$08
mov    eax, eax               ; mov eax, I
mov    [ebp-$04], eax         ; mov K, eax
mov    eax, edx               ; mov eax, J
mov    [ebp-$08], eax         ; mov L, eax
pop    ecx                    ; end
```

```
pop     ecx
pop     ebp
ret
```

Listing 4: Disassemblat eines vom Delphi-6.0-Compiler gemäß Listing 3 erzeugten Compilates

Hier finden wir zweierlei:

- Durch die Verwendung von *asm* und *end* ohne *begin – end* werden andere Prologe und Epiloge erzeugt. Je nachdem, ob und wie Parameter übergeben werden (hier: über die Register EAX und EDX), wird ein Stack-Rahmen erzeugt (Fall 3 und 4) oder nicht (Fall 1 und 2). Und wenn er erzeugt wird, kann er unterschiedlich in der Größe ausfallen, je nachdem, ob lokale Variable definiert oder Parameter übergeben werden und wie diese Parameter übergeben werden. So ließ sich mein Delphi-6.0-Compiler trotz guter Zurede über Compilerschalter {$O-}, {$OPTIMIZATION OFF} und/oder IDE-Option nicht dazu überreden, die Parameter *nicht* via Register zu übergeben!
- Der gravierendste, wenn auch zunächst vielleicht nicht auffälligste Unterschied aber ist: In Object-Pascal-Teilen (also bei Verwendung von Delphi-Befehlen) stellt eine Referenz auf eine Variable den Inhalt dieser Variablen dar, im integrierten Assembler dagegen die Adresse der Speicherstelle (siehe fett dargestellte Zeilen in Listing 4)!

Variablen-Referenzen

Das halte ich, vorsichtig ausgedrückt, für sehr unglücklich, da es den Programmiergewohnheiten der Delphi-Programmierer entgegensteht. Wer den Inline-Assembler nutzt, will nicht unbedingt auf eine neue Sichtweise »umschalten« – sonst könnte er ja gleich den Stand-Alone-Assembler verwenden! Es steht auch in gewisser Weise den Mechanismen entgegen, die Borland mit dem IDEAL-Modus des TASM selbst der Deutlichkeit halber geschaffen hat. Hier gibt es ja die Zuordnung *MOV EAX, I* nicht – man muss entweder *MOV EAX, [I]* verwenden, um auf den Inhalt zugreifen zu können, oder *MOV EAX, OFFSET I*, wenn man die Adresse haben möchte.

Befremdlich ist dieser Dachverhalt auch aus zwei anderen Gründen! So ist im Inline-Assembler sehr wohl der Operator OFFSET erlaubt, man hätte sich also vollständig auf den IDEAL-Modus zurückziehen können. Das wäre konsequenter, wenn auch nicht unbedingt »richtiger« gewesen. Was aber noch schlimmer ist: Das Ganze ändert sich wiederum,

wenn die Routinen mit *begin – end* geklammert werden, ansonsten aber alles beim Alten bleibt:

```
procedure SimpleProc;
begin
asm
  push  $00
  call  windows.messagebeep
end;
end;

procedure ParamProc(var I, J: Integer);
begin
asm
  mov   I, $04
  mov   J, $05
end;
end;

procedure LocalVarProc;
var I, J: Integer
begin
asm
  mov   I, $04;
  mov   J, $05
end;
end;

procedure LocalVarAndParamProc(var I, J: Integer);
var K, L: Integer
begin
asm
  mov    eax, I
  mov K, eax
  mov    eax, J
  mov L, eax
end;
end;
```

Listing 5: Delphi-6.0-Quelltext mit Assembleranweisungen in Delphi-Prozeduren

Hier wird offensichtlich wieder im Delphi-Sinne die Variable mit ihrer Adresse, nicht mit ihrem Inhalt gleichgesetzt, wie folgendes Disassemblat zeigt:

```
; keine Parameter, keine lokalen Variablen
push    $00                     ; Parameter auf den Stack
call    MessageBeep             ; MessageBeep aufrufen
ret                             ; end

; zwei Parameter
push    ebp
mov     ebp, esp
add     esp, -$08
mov     [ebp-$08], edx
mov     [ebp-$04], eax
mov     [ebp-$04],$00000004     ; mov   I, $04
mov     [ebp-$05],$00000005     ; mov   J, $05
pop     ecx
pop     ecx
ret                             ; end

; zwei lokale Variablen
push    ebp                     ; asm
mov     ebp, esp
add     esp, -$08
mov     [ebp-$04],$00000004     ; mov   I, $04
mov     [ebp-$08],$00000005     ; mov   J, $05
pop     ecx                     ; end
pop     ecx
pop     ebp
ret

; zwei Parameter, zwei lokale Variablen
push    ebp                     ; begin
mov     ebp, esp
add     esp, -$10
mov     [ebp-$08], edx
mov     [ebp-$04], eax
mov     eax, [ebp-$04]          ; mov eax, I
mov     [ebp-$08], eax          ; mov K, eax
mov     eax, [ebp-$08]          ; mov eax, J
mov     [ebp-$10], eax          ; mov L, eax
mov     esp, ebp                ; end
pop     ebp
ret
```

Listing 6: Disassemblat eines vom Delphi-6.0-Compiler gemäß Listing 5 erzeugten Compilates

Das ist weder logisch noch konsequent noch überzeugend! Ich persönlich halte das für einen gravierenden Bug, sodass folgende Anmerkung hier nochmals und nachdrücklich angefügt wird:



Delphi macht Unterschiede in der Verwendung von Variablen, je nachdem, ob sie in Assembler-Routinen (= Routinen ohne Begin-End-Block) oder in Delphi-Routinen (= Routinen mit Begin-End-Block) eingesetzt werden. So referenzieren Variablen in Delphi-Routinen die Daten selbst. Die Angabe der Variablen in Anweisungen wie *MOV EAX, I* bewirkt hier selbst in Assembler-Blöcken (*asm – end*) die Verwendung des unter der angegebenen Adresse stehenden Datums gemäß *MOV EAX, [I]*.

In Assembler-Routinen dagegen betrachtet Delphi die Variable als Adresse. Die Angabe der Variable gemäß *MOV EAX, I* lädt hier die Adresse, unter der das Datum ansprechbar ist, nicht etwa das Datum selbst analog zu *MOV EAX, OFFSET I*.



Daher mein Tipp: Verwenden Sie Assembler-Routinen sehr sparsam, wenn Sie nicht routiniert genug sind. Der Wasserkopf eines eingerichteten und wieder entfernten Stack-Rahmens, der durch *begin – end* erzeugt wird, schadet in der Regel nicht, macht sich kaum in der Performance bemerkbar und bläht Ihren Code nicht auf, bewahrt Sie aber vor unliebsamen und schwer zu entdeckenden Fehlern.

Das sich Assembler-Routinen so anders verhalten, noch einige Hinweise:

- Der Compiler erstellt in solchen Routinen keinen Code zum Kopieren von Wert-Parametern (»call by value«) in lokale Variable. (Bitte den Unterschied zu Zeiger-Parametern, »call by reference«, beachten, die in der Parameterliste der Routine mit *var* gekennzeichnet sind. Bei ihnen wird die Adresse des Datums als Zeiger übergeben!) Wichtig ist dies somit bei Daten, die wertmäßig nicht über den Stack (oder bei Optimierungen über ein Register) übergeben werden können: Strings, Records etc. Der Compiler übergibt hier die Adresse des Originals und erstellt, wie gesagt, keine lokale Kopie, was er in Delphi-Routinen tut. Solche Parameter sind somit grundsätzlich als Var-Parameter zu betrachten und zu behandeln!
- Die Verwendung der Referenz auf das Symbol @Result in Assembler-Funktionen ist ein Fehler! Mit einer Ausnahme ist @Result in

solchen Funktionen nicht deklariert. Die Ausnahme sind Funktionen, die eine Referenz auf einen String, eine Variante oder eine Schnittstelle (»interface«) zurückgeben. Ist somit die Funktion vom Type String, Variante oder Interface deklariert, gibt es das Symbol @Result.

- Der Assembler erzeugt keinen Stack-Rahmen, falls die Routine nicht Parameter und/oder lokale Variable hat (siehe SimpleProc in den Listings oben).
- Wie Sie in den Listings oben nachvollziehen können, wird der Pro- und Epilog gemäß dem Vorhandensein von lokalen Variablen und/oder Parametern wie folgt gebildet:

```
; Prolog:
PUSH  ESP          ; nur, wenn locals≠0 und/oder params≠0
MOV   ESP, EBP     ; nur, wenn locals≠0 und/oder params≠0
SUB   ESP, locals  ; nur, wenn locals≠0

; Epilog
MOV   ESP, EBP     ; nur, wenn locals≠0
POP   EBP          ; nur, wenn locals≠0 und/oder params≠0
RET
```

Wie Sie in den Listings gesehen haben, kann die Reservierung von Platz auf dem Stack (*SUB ESP, locals*) je nach Situation auch durch »schnellere« Instruktionen (*PUSH ECX* mit anschließendem Überschreiben des Wertes auf dem Stack) realisiert werden. Gleiches gilt für die Zerstörung des Stacks (*MOV ESP, EBP* bzw. *POP ECX*). Kriterium, wann welcher Weg gewählt wird, ist die Länge der Instruktionen: Da *MOV ESP, EBP* ein ModR/M-Byte benötigt, ist die Länge der Instruktion zwei Bytes – genauso lang wie zwei *PUSH-ECX*-Befehle. Erst bei drei lokalen Variablen/Parametern macht also *MOV ESP, EBP* das Rennen.

Allzweckregister

Es gibt noch ein paar Punkte zu berücksichtigen. So verwendet Delphi einige der Allzweckregister für interne Zwecke, wie z.B. der Verwaltung des Stack (EPB und ESP).



Daher gelten unter Delphi die Register EDI, ESI, ESP, EBP und EBX als tabu! Das bedeutet, dass sie innerhalb von Assemblerteilen nur dann verwendet werden dürfen, wenn ihre Inhalte unmittelbar nach dem Prolog auf den Stack gesichert werden und unmittelbar vor dem Epilog wieder von dort restauriert werden.

Eine typische Inline-Assembler-Routine könnte dann wie folgt aussehen:

```
procedure ASMProc(var Features:LongInt);
asm
  push  ebx                   ; EBX wird durch CPUID verändert!
  mov   eax, $00000001        ; Funktion 1 des CPUID-Befehls
  cpuid
  mov   [Features], edx       ; feature flags zurückgeben
  pop   ebx                   ; EBX restaurieren
  ret
end;
```

Unterbliebe dies, führte dies zu ernsthaften Problemen!

Labels

Für Labels, die in Assemblerteilen eingesetzt werden, gelten die gleichen Bedingungen wie für Labels in Delphi-Teilen. Eine Ausnahme: lokale Labels. Sie haben alle mit einem @-Zeichen zu beginnen und gelten nur in dem Teil, der sie enthält und durch die Schlüsselworte asm und end begrenzt wird. Außerhalb dieses Blocks sind solche Labels nicht sichtbar.

Vordefinierte Symbole

Der Inline-Assembler kennt auch einige vordefinierte Symbole. So steht *@Code* für den Selektor des Code- und *@Data* für den des Datensegments. Innerhalb von Funktionen verweist *@Result* auf die Adresse der Variablen, die in Delphi mittels Result angesprochen werden kann:

```
function GetFour: LongInt;
begin
  Result := 2;
  asm
    mov   eax, @Result
    add   @Result, EAX
  end;
end;
```



@Result ist nur in *Funktionen* deklariert, die einen Pro- und Epilog besitzen, der mit *begin – end* erzeugt wurde! In allen anderen Fällen führt die Verwendung von @Result zu einem Fehler.

4.2 Inline-Assembler und die strukturierte Ausnahmebehandlung (SEH)

Noch ein kurzes Wort zum Thema »Exceptions und der integrierte Assembler«. Wie Sie leicht nachvollziehen können, ist das überhaupt kein Problem. Denn Sie können natürlich den integrierten Assembler auch innerhalb von Quelltextteilen verwenden, die mit der strukturierten Ausnahmebehandlung zu tun haben. Hier hat der integrierte Assembler sicherlich Vorteile gegenüber dem Stand-Alone-Bruder, wo ja doch ein paar Klimmzüge notwendig werden, wie wir in Kapitel »Assembler und die strukturierte Ausnahmebehandlung (SEH)« auf Seite 744 gesehen haben.

Teil 3: Anhang

5 Anhang

5.1 Definitionen und Erläuterungen

5.1.1 Befehlssemantik

Die meisten Befehle von CPU, FPU und SIMD-Einheit benötigen Daten als Input und/oder produzieren Daten als Output. Zur Datenübergabe verwenden sie Operanden, aus denen sie die Ausgangsdaten beziehen (*source operands*, »Quelloperanden«) und in die sie die Ergebnisse ablegen (*destination operands*, »Zieloperanden«). Diese Operanden folgen in der Intel-Semantik unmittelbar dem Befehl und werden von links nach rechts durchnummeriert:

Befehl [Operand #1[, Operand #2[, Operand #3]]]

Die eckigen Klammern sollen ausdrücken, dass jeder Befehl eine unterschiedliche Anzahl von Operanden haben kann. So gibt es Befehle, die überhaupt keinen Operanden benötigen, wie z.B. *WAIT* oder *NOP*. Andere Befehle besitzen einen Operanden (*NOT*) oder zwei (*ADD*). Einige wenige Operanden benötigen sogar drei Operanden (*SHLD*, *IMUL*).

Der unmittelbar auf den Befehl folgende Operand (Operand #1) besitzt bis auf wenige Ausnahmen Zwitterfunktion. So ist er in der Regel, wenn vorhanden, sowohl der erste (und vielleicht einzige) Quelloperand wie auch Zieloperand, wie das Beispiel *NOT* zeigt: `NOT EAX`. Hier ist das Register EAX sowohl Quelle des zu bearbeitenden Datums (source operand) als auch Ziel für die Ergebnisablage (destination operand), was durch die geschweiften Klammern angedeutet wird:

Befehl {Destination=Source}

Diese Zwitterfunktion besitzt Operand #1 auch bei Befehlen, die zwei Operanden erfordern. So addiert der Befehl `ADD EAX, EBX` zur Quelle #1 (= EAX) die Quelle #2 (= EBX) und legt das Ergebnis im Ziel (= EAX) ab:

Befehl {Destination=Source#1}, Source#2

Bitte beachten Sie die Reihenfolge: Operand #1 ist immer der erste Operand! Wichtig ist dieser Unterschied bei nicht-kommutativen Befehlen, also bei Befehlen, bei denen die Vertauschung der Reihenfolge der Operanden zu unterschiedlichen Ergebnissen führt, z.B. bei der Subtraktion: So zieht `SUB EAX, EBX` den in EBX stehenden Wert von dem in EAX stehenden ab (EAX := EAX – EBX), nicht etwa umgekehrt.

Bei den wenigen Befehlen mit drei Operanden gibt es zwei Gruppen: Bei der einen Gruppe fungiert Operand #1 wiederum als Zwitter, ist also der erste von drei Quelloperanden und gleichzeitig Zieloperand:

Befehl {Destination=Source#1}, Source#2, Source#3

während bei der zweiten Gruppe Operand #1 dezidierter Zieloperand ist. Der Befehl bezieht seinen Input dann aus Operand #2 und Operand #3:

Befehl Destination, Source#1, Source#2

Beispiel für Fall 1 ist `SHLD EAX, EBX, Const`, bei dem *Const* Bits nach links aus EAX herausgeschoben und rechts mit der gleichen Anzahl von Bits ergänzt werden, die ihrerseits links aus EBX herausgeschoben werden. In EAX (Operand #1 = Destination) stehen somit *(32 – Const)* Bits aus EAX (Operand #1 = Source #1) und *Const* (Operand #3 = Source #3) Bits aus EBX (Operand #2 = Source #2). Beispiel für Fall 2 ist die Drei-Operanden-Version von IMUL: `IMUL EAX, EBX, Const`. Hier wird im Zieloperanden (Operand #1) das Produkt des ersten Quelloperanden (Operand #2) und einer Konstanten (Operand #3 = Quelloperand #2) abgespeichert.



Vorsicht! Manche Befehle haben vermeintlich keinen Operanden (*AAA*), in Wirklichkeit aber ist dieser Operand impliziert. So wirkt *AAA* grundsätzlich nur auf den Inhalt von AX, müsste also theoretisch heißen `AAA AX`. Heißt er aber nicht! Selbst *zwei* implizierte Operatoren gibt es: So müsste die Korrektur nach einer Multiplikation mit BCD-Zahlen eigentlich `AAM AX, $0A` heißen. Sie heißt aber lediglich *AAM*, da sowohl das Register AX als auch die Konstante $0A impliziert werden.

Und um es noch ein wenig komplizierter zu machen: Nicht jeder Ein-Operand-Befehl hat auch nur einen Operanden. Analog zu *AAA* und *AAM* gibt es Zwei-Operanden-Befehle, die einfach einen Operanden implizieren. Und perfiderweise ist das sogar häufig der Zwitteroperand #1, wie bei `MUL EBX`. Dieser Befehl hieße korrekt `MUL EDX:EAX, EAX`,

EBX, da er das DoubleWord aus EAX (Source #1) mit dem aus EBX (Source #2) multipliziert und das resultierende QuadWord in der Registerkombination EDX (höherwertiges DoubleWord) und EAX (niedrigerwertiges DoubleWord) ablegt.

5.1.2 Adress- und Operandengrößen

Bei Zugriffen auf den Speicher spielen zwei Fragen eine wesentliche Rolle, für die der Prozessor eine Antwort braucht:

- Werden die in der Befehlssequenz angegebenen Adressen, über die der Zugriff erfolgen soll, mit 16 oder mit 32 Bits angegeben? Folgen also dem Opcode zwei oder vier Bytes, die als Adresse zu interpretieren sind?
- Haben die Operanden des Befehls eine Größe von 16 oder 32 Bits? Dies ist deshalb wichtig, da alle Befehle, die wahlweise mit 16-Bit- oder 32-Bit-Operanden arbeiten können – z.B. `MOV AX, Mem16` und `MOV EAX Mem32` –, mit dem gleichen Opcode (hier: $A1) codiert werden! Also: Was codiert nun z.B. $A1?

Um nicht bei jeder Befehlssequenz, die auf Speicher zugreift, explizit die Daten- und Adressgröße angeben zu müssen, haben die Prozessorhersteller Standardgrößen für Adressen und Daten definiert. Das bedeutet, dass in Fällen, in denen von dem einmal definierten Standard abgewichen wird, dies durch »Präfixe« (vgl. Seite 134 bzw. 769) signalisiert werden muss: Wenn nicht die Standard-Adressgröße verwendet wird, muss dem Opcode der *address size override prefix* (siehe Seite 135) vorangestellt werden, bei Verwendung von Nicht-Standard-Datengrößen (evtl. auch zusätzlich) der *operand size override prefix* (siehe Seite 135).

Was also heißt in diesem Zusammenhang »Standard-«? Wenn die Register, mit denen ein Prozessor Daten (und damit auch Adressen!) verwalten kann, nur 16 Bit breit sind, spricht man von 16-Bit-Prozessoren. Bei solchen Prozessoren können somit standardmäßig nur 16-Bit-Daten verarbeitet werden, Standard-Daten- und Adressgröße ist somit 16 Bit. In Prozessoren, die 32 Bit breite Register besitzen, können Daten mit 32 Bit verwaltet werden. Es ist somit, wenn auch nicht zwingend, so jedoch plausibel und logisch, anzunehmen, bei solchen Prozessoren seien 32-Bit-Daten und -Adressen Standard. Das ist auch so: Wozu braucht man einen Ferrari, wenn man nur in Zone-30-Gebieten mit »Verkehrsberuhigungsinseln« als Schikanen fährt?

Doch sind nicht nur die physikalischen Voraussetzungen des Prozessors zu berücksichtigen. Auch das Betriebssystem spricht ein Wörtchen mit: DOS und Windows 3.x waren Betriebssysteme, die auf 16-Bit-Prozessoren im 16-Bit-Modus (real mode) liefen. Aus Kompatibilitätsgründen sollten sie jedoch auch auf 32-Bit-Prozessoren laufen, was sie ja auch taten. Da sie jedoch für 16-Bit-Systeme ausgelegt waren, mussten sie auch auf 32-Bit-Prozessoren mit 16-Bit-Daten und -Adressen arbeiten. Das bedeutet: Auch die Umgebung, in der ein Programm läuft (und damit Assemblerbefehle abarbeitet), muss berücksichtigt werden (z.B. real mode).

Mit dem 80386 kamen dann die ersten 32-Bit-Prozessoren auf und mit OS/2 und Windows NT/95 die ersten 32-Bit-Betriebssysteme. Sie waren für 32-Bit-Prozessoren und den protected mode ausgelegt, liefen nicht mehr auf 16-Bit-Prozessoren und hätten es eigentlich nicht mehr nötig gehabt, auch 16 Bit als Standard ansehen zu müssen, wenn nicht ... ja wenn nicht die DOS- und Win3.x-Kompatibilität gewesen wäre, also auch in 32-Bit-Umgebungen 16-Bit- Programme laufen sollten. (Der 80286 spielte zu diesem Zeitpunkt bereits keine Rolle mehr, sodass man auf den 16-Bit-Protected-Mode keine Rücksicht mehr nehmen musste!)

Address Size

Daher ist in jedem Segment, das in den 32-Bit-Betriebssystemen verwendet wird, die Standardgröße für Adressen und Daten verzeichnet: In Codesegmenten gibt es das Flag D (default length; siehe Seite 409), das die aktuelle Breite von Adressen angibt (»*address size*«). Ist das Flag D gesetzt (32-Bit-Codesegment), so gelten in diesem Segment 32-Bit-Adressen als Standard und zur Nutzung von 16-Bit-Adressen muss dieser Standard durch den *address size override prefix* außer Kraft gesetzt werden. Dies erfolgt nur für den folgenden Befehl, da der Präfix ja Teil der Befehlssequenz ist, die einen Befehl codiert. Ist D dagegen gelöscht (16-Bit-Codesegment), so sind 16-Bit-Adressen Standard und der *address size override prefix* erhöht für den folgenden Befehl die Adressgröße auf 32 Bit.

Operand Size

Analoges gilt für Daten: Ist im aktuellen Datensegment das Flag B (big flag; siehe Seite 412), das die aktuelle Breite von Daten und somit »Operanden« für Befehle angibt (»*operand size*«), gesetzt (32-Bit-Datensegment), so sagt es dem Prozessor, dass er standardmäßig 32-Bit-Daten zu arbeiten hat, ein Speicherzugriff auf die angegebene Adresse also mit vier konsekutiven Bytes (DoubleWord) zu erfolgen hat. Ein *operand size override prefix* reduziert die Größe dann für den nächsten Befehl auf 16

Bit und einen Word-Zugriff auf die Speicherstelle mit der angegebenen Adresse. Bei gelöschtem big flag (16-Bit-Datensegment) dagegen wird standardmäßig mit Word-Zugriffen gearbeitet, der *operand size override prefix* lässt in diesem Fall den nächsten Zugriff mit einem DoubleWord erfolgen.

Stack Operand Size

Beachten Sie auch, dass das Stacksegment, als Sonderform eines Datensegments, ein eigenes B-Flag besitzt und damit die Struktur des Stacks beschreibt (»*stack operand size*«)! Ist es gelöscht, so ist der Stack 16-bittig aufgebaut und die PUSH- und POP-Befehle (inkl. PUSHF/POPF) arbeiten mit Words. Bei gesetztem B-Flag dagegen ist der Stack 32-bittig ausgelegt und die entsprechenden Befehle verwenden DoubleWords. Wichtig ist dies, da dies Auswirkungen auf die Zusammenarbeit Stack – Datensegment hat. Wie die einzelnen Befehle in den unterschiedlichen Kombinationen des *stack operand size* und *operand size* zusammenarbeiten, steht jeweils ausführlich in den Beschreibungen der betroffenen Befehle in Band 2, Die Assembler-Referenz, unter dem Stichwort »Operation«.



Auch wenn es redundant und daher überflüssig sein mag: Bitte beachten Sie, dass die Adressgröße nichts über die Datengröße aussagt und umgekehrt. *Data size* und *operand size* sind zwei Größen, die nichts, aber auch gar nichts miteinander zu tun haben, auch wenn sie evtl. bei der Erstellung einer Befehlssequenz (»assemblieren«) beide berücksichtigt werden müssen. So kann in 16-Bit-Umgebungen mit 16-Bit-Adressen durchaus auf 32-Bit-Daten zugegriffen werden und in 32-Bit-Umgebungen mit 32-Bit-Adressen auf 16-Bit-Daten! Die Frage der Adressgröße ist: »Wie viel Speicher kann der Prozessor adressieren?« und bei der Datengröße: »Handelt es sich bei dem Datum an dieser Adresse um diesen Datentyp?«.

Und auch die anderen relevanten Segmente haben diese Größenangabe: So wird ein Call aus 32-Bit-Segmenten über 16-Bit-Gates oder in ein 16-Bit-Codesegment (alter Gerätetreiber?) mit einer Befehlssequenz kodiert, die den *address size override prefix enthält*, während der Zugriff auf 32-Bit-Gates oder -Codesegmente ohne diesen Präfix auskommt. Analoges gilt für den Zugriff auf task state segments.

5.1.3 Mnemonics, Befehlssequenzen, Opcodes und Microcode

Was haben die Bytes $88, $89, $8A, $8B, $8C, $8E, $A0, $A1, $A2, $A3, $B0, $B8, $C6, $C7 mit den Words $0F20, $0F21, $0F22, $0F23 gemeinsam und was unterscheidet sie z.B. von den Words $DBE3 oder den Bytes $00, $01, $02, $03, $04 und $05?

Bevor Sie nun beginnen, irgendeine Systematik in den Codes zu suchen, folgt lieber gleich die Auflösung: Sie haben ein *mnemonic* gemeinsam, nämlich »MOV«, und unterscheiden sich dadurch von anderen Bytes/Words mit anderen Mnemonics, wie z.B. *FNINIT* oder *ADD*.

Mnemonics

Was sind Mnemonics? Es sind »Hilfsmittel«, sich die kryptischen Bytewürmer, aus denen die Maschinenbefehle zusammengesetzt sind, zu merken. Der Begriff *Mnemonik* ist gleichbedeutend mit »Mnemotechnik« und kommt einmal nicht aus dem Angelsächsischen, sondern aus dem Griechischen, heißt dort so viel wie »Gedächtniskunst« und wird verstanden als »Erleichterung des Sicheinprägens schwieriger Gedächtnisstoffe durch besondere Lernhilfen«. Diese Lernhilfen sind im Falle von Assembler und Maschinencode *Namen* für Instruktionen, unter denen man sich etwas vorstellen kann: Sie werden zugeben, dass Ihnen »`MOV EDI, [Quelle]`« erheblich mehr sagt als »`8B3DBCA0FCBF`«. (Übrigens: MOV steht hier für den Opcode $8B.)

Befehlssequenz

Die eigentlichen Prozessor-verständlichen Instruktionen setzen sich aus mehr oder weniger komplizierten Sequenzen von Bytes zusammen. So ist die formale Deklaration einer Befehlssequenz:

Intel-Codierung

[prefix(es)] opcode byte(s) [ModR/M [SIB] [displacement]] [immediate]

wobei nicht alle Teile realisiert sein müssen, was die eckigen Klammern andeuten sollen: Der Ein-Byte-Opcode »$37« wird durch das Mnemonic »*AAA*« repräsentiert, hat weder Präfixe notwendig noch Operanden, weshalb auch kein *immediate* nötig, kein ModR/M- oder SIB-Byte erforderlich ist und damit auch kein *displacement* zum Zuge kommt.

Opcode

Wichtigster Teil dieser Sequenzen sind die Opcode-Bytes (»der Opcode«), da sie die Aktion angeben, die durchgeführt werden soll: Addition, Datenspeicherung, Sprung etc. Ein Opcode kann entweder aus Ein- oder aus Zwei-Byte-Instruktionen bestehen, also maximal 2 Bytes umfassen. In einigen Fällen reichen die zwei Bytes nicht aus, um den Opcode vollständig zu definieren. In diesem Fall existiert auf jeden Fall ein ModR/M-Byte, dessen *reg field* dann zur zusätzlichen Opcode-

Codierung herangezogen wird. Einzelheiten hierzu stehen im Kapitel »Speicheradressierung« auf Seite 816. Bei einigen neueren Befehlen, vor allem aus dem SIMD-Befehlssatz, dient das Präfix als drittes Byte eines häufig so (nicht ganz korrekt) genannten »Drei-Byte-Opcodes«.

prefix

Das/die Präfix(e) dient/dienen, so vorhanden, der Definition einer »Nicht-Standardsituation«, unter der der Befehl abzulaufen hat. So gibt es Präfixe, die die standardmäßige Adressberechnung außer Kraft setzen (*address size override*) oder die Größe der zu verwendenden Operanden (*operand size override*) oder dafür sorgen, dass der folgende Befehl mehrfach wiederholt wird (*REP*). Weitere Präfixe haben wir in Kapitel 1 bereits besprochen. Es können bis zu vier Präfixe dem eigentlichen Befehl vorangehen, wobei allerdings nicht alle Kombinationen erlaubt sind. Vielmehr werden sie in vier Gruppen eingeteilt, aus denen nur jeweils ein Präfix ausgewählt werden kann (vgl. Tabelle 5.1).

Gruppe 1	**Gruppe 2**	**Gruppe 3**	**Gruppe 4**
Repetierung:	segment override:	$66 operand size override	$67 address size override
$F0 LOCK	$2E CS:		
$F2 REPNE	$3E DS:		
$F3 REP/REPE	$26 ES:		
	$64 FS:		
»3-Byte-Opcode«:	$65 GS:		
$F2 SSE	$36 SS:		
$F3 SSE/SSE2	branch hints:		
	$2E branch taken		
	$3E branch not taken		

Tabelle 5.1: Einteilung der Befehlspräfixe in Gruppen



Bitte beachten Sie hierbei, dass die Wiederholungspräfixe $F2 und $F3 nur in Verbindung mit Stringbefehlen erlaubt sind (ansonsten gelten sie als reserviert und werden, wie auch $66 – operand size override – z.B. bei einigen SSE- und SSE2-Befehlen zur Erweiterung des Opcodes verwendet) und die *segment override prefixes* in Verbindung mit Verzweigungsbefehlen (JMP, Jcc, CALL, ...) als reserviert gelten (weshalb auch die Präfixe $2E und $3E in Verbindung mit den bedingten Sprüngen Jcc als *branch hints* fungieren! Man kann gespannt sein, was bei zukünftigen Prozessoren die Präfixe $26, $64, $65 und $36 in Verbindung mit irgendwelchen Befehlen bewirken ...). Beachten Sie auch, dass die Verwendung von mehr als einem Präfix aus einer Gruppe zu unvorhergesehenen, prozessorspezifischen Ergebnissen führen kann und daher tunlichst unterbleiben sollte.

ModR/M
SIB
displacement

ModR/M, SIB und *displacement* spielen bei der Adressierung eine Rolle und kommen nur dann als Teil der Befehlssequenz vor, wenn eine Speicherstelle und/oder ein Register als Operand im Spiel sind. ModR/M und SIB sind jeweils Bytes, während displacements 1, 2 oder 4 Bytes lang sein können. (Hier entscheidet z.B. das address size override prefix mit, wie viele Bytes zum Tragen kommen. Oder anders ausgedrückt: Das address size override prefix gibt an, wie viele der dem Mod/RM- bzw. SIB-Byte folgenden Bytes in Nicht-Standard-Situationen als displacement zu interpretieren sind.) Zu Einzelheiten konsultieren Sie bitte Kapitel »Speicheradressierung« auf Seite 816 und »Befehlscodierung« in der Assembler-Referenz.

immediate

Mit *immediate* wird üblicherweise eine Konstante bezeichnet, die Teil der Befehlsfolge sein kann, wie z.B. in `MOV EAX, 0FEDh`, in der »0FEDh« dieses immediate darstellt. Immediates können 1, 2 oder 4 Bytes lang sein. Auch hier spielt ein prefix eine Rolle: das *operand size override prefix*, das angibt, wie viele der folgenden Bytes als Konstante zu interpretieren sind (falls nicht bereits im Befehl selbst codiert ist, welche Größe immediate hat. Häufig werden befehlsbedingt und unabhängig von *operand size* nur 8-Bit-Konstanten als Operand gestattet.).

Zurück zu den Mnemonics und Opcodes. Die Opcodes sind, wie gesagt, die wesentlichen Bytes, da sie die Art der Aktion definieren. So lässt der Opcode $88 den Prozessor ein Byte aus dem Speicher in ein Register transferieren, während $89 ihn dazu veranlasst, (je nach Standardeinstellung und Vorhandensein des address size override prefix) ein Word oder ein DoubleWord in ein Register zu laden. $8A bzw. $8B dagegen bewirken das Gegenteil: ein Byte, Word oder DoubleWord wird aus einem Register in den Speicher verfrachtet. Bei diesen Opcodes dürfen aber nur die Allzweckregister eingesetzt werden. Soll das Gleiche mit den Segmentregistern erfolgen, kommen die Opcodes $8C (ein Word aus dem Speicher in ein Segmentregister) und $8E (das Ganze zurück) zum Einsatz – Bytes und DoubleWords sind hier ausgeschlossen.

Um das Ganze nicht ganz so einfach zu machen, gibt es noch eine Ausnahme. Der Akkumulator EAX (bzw. seine Teile AX und AL) haben eine gewisse Sonderbedeutung unter den Registern, wie wir an mehreren Stellen bereits gesehen haben. Dieser Sonderbedeutung wird auf Befehlsebene dadurch Rechnung getragen, dass es zusätzliche Sonder-Opcodes gibt, falls der Akkumulator involviert ist, also z.B. eine Speicherstelle in den Akkumulator oder vice versa übertragen werden soll: $A0, wenn ein Byte in AL kommt, $A1, wenn es sich um ein Word oder

DoubleWord handelt, und $A2 und $A3, wenn das Ganze umgekehrt erfolgt. Sinn der Übung: Es handelt sich hierbei um sehr schnell auszuführende Ein-Byte-Befehle, die im günstigsten Fall einen Bruchteil eines Taktes zur Ausführung benötigen.

Verzichten wir auf die Darstellung der restlichen oben genannten Opcodes und halten wir fest, dass alle diese Opcodes für Befehle stehen, die irgendein Datum von einem zum anderen Platz bewegen. Sie bilden damit eine Gruppe von Befehlen, die unter einem gemeinsamen »Etikett« gehandelt werden, für das es genau ein leicht zu merkendes und überzeugendes Mnemonic gibt: *MOV*. In welche Opcodes dieses Mnemonic dann übersetzt wird, können wir getrost dem Assembler überlassen. Er wird in Abhängigkeit von Operanden und anderen Randbedingungen die korrekte Befehlssequenz zusammenbauen.

3DNow!-Codierung

AMD hat bei der Einführung der 3DNow!-Befehle eine etwas andere Codierung verwendet, bei der die Unterscheidung der Befehle über ein Suffix erfolgt. Alle 3DNow!-Befehle fangen mit dem Zwei-Byte-Code $0F0F an:

0Fh 0Fh ModR/M [SIB] [displacement] Suffix

Falls somit ein AMD-Prozessor die »magische« Opcode-Folge $0F0F »sieht«, weiß er, dass ein 3DNow!-Befehl folgt. Über das ModR/M- und ggf. das SIB-Byte weiß er dann ebenfalls, welche Register involviert sind und ob ein Speicheroperand eine Rolle spielt. Im letzteren Fall kommt anschließend das displacement ins Spiel. Welcher 3D-Now!-Befehl nun mit den bereits decodierten Operanden auszuführen ist, sagt ihm dann das folgende Suffix. Wie gesagt, das gilt nur für die 3DNow!-Befehle. Alle anderen, Intel-analogen Befehle der AMD-Prozessoren folgen natürlich der Intel-Codierung.

Intel-Prozessoren und die meisten Assembler/Debugger fangen mit dieser Befehlsfolge nichts an!



Microcode

Doch wie passen die »Microcodes« in diese Problematik? Sie haben spätestens in Vorwort und Einleitung dieses Buches davon gehört. Die Befehlssequenzen oder besser: die Opcodes, die der Prozessor einliest, sind nicht die »niedrigsten« Elemente, mit denen der Prozessor umgehen kann und muss. Vielmehr ist auch bereits ein so vermeintlich einfach aufgebauter Opcode wie $88, der ein Byte aus dem Speicher in ein Register überträgt, ein aus verschiedenen, grundlegenden Aktionen zusammengesetzter Befehl: Lege die übergebene Adresse auf den

Adressbus, veranlasse, dass die adressierte Speicherstelle ausgelesen und auf den Datenbus gelegt wird, hole das Datum von dort ab und lege es im angegebenen Register ab. Mit anderen Worten: Für den Prozessor sind die Opcodes vergleichbar mit einer »Hochsprache«, die er in die ausführbaren Aktionen zerlegt. Diese Aktionen bilden den Microcode.

5.1.4 Anwendungen, Programme, Module, Tasks, Prozesse und Threads

Ach wie schön einfach war doch alles unter DOS! Da hatte man seinen überschaubaren Speicher, sein Programm, seine Daten und sein Betriebssystem. Man gab letzterem via Kommandozeileninterpreter COMMAND.COM den Befehl, ein Programm zu starten, das dann seinerseits die Daten lud, die es benötigte. Und schon konnte man arbeiten.

Heute ist das nicht mehr so einfach! Man arbeitet ja in Multitasking-Umgebungen. Das bedeutet, dass mehrere »Programme«, also »tasks« (?), gleichzeitig laufen können. Ist also *task* nur ein anderes Wort für *Programm*? Und welche Bewandtnis hat es dann mit den »Prozessen«, die in den modernen Hochsprachen von heute eine wesentliche Rolle spielen? Ganz zu schweigen von den »threads«? (Für viele sind *threads* in Wirklichkeit *threats*!)

Auch wenn ich Gefahr laufe, Eulen nach Athen zu tragen und Ihnen als Hochsprachenprofis hier etwas zu Threads & Co sage, was Ihnen sowieso völlig klar ist, sollte kurz eine Definition dieser Begriffe erfolgen.

Programm Als »Programm« wird eine ausführbare Datei (.COM, .EXE) bezeichnet, die auf einem Datenträger (Festplatte, Floppy, CD-ROM etc.) darauf wartet, zwecks Ausführung durch das Betriebssystem geladen zu werden. Sie enthält den spezifischen ausführbaren Code sowie alle Informationen, um Datenstrukturen einzurichten, die während des Programmablaufs notwendig sind. Häufig bezeichnet man Programme auch als »Module«, weshalb die Betriebssystem-Funktion unter Windows, die ein Programm lädt, in früheren Versionen auch »LoadModule« hieß (heute wird üblicherweise »CreateProcess« verwendet).

Modul Das »eigentliche« Modul ist aber wesentlich breiter definiert. So enthalten Module generell ausführbaren Code, weshalb neben den ausführbaren Programmen auch dynamisch linkbare Bibliotheken (DLLs) und Teile des Betriebssystems Module sind.

Anwendung

Unter 32-Bit-Windows sind Betriebssystem und Programm sehr viel stärker ineinander verzahnt als unter DOS. So konnten geeignet programmierte Programme früher fast vollständig auf Betriebssystemfunktionen verzichten (Beispiel: direkte Text- und Graphikausgabe in den Bildschirmspeicher oder BIOS-Aufrufe anstelle der Nutzung der DOS-Funktionen), die einzige Aufgabe des Betriebssystems war häufig nur, das Programm zu laden und zu starten. Heute ist diese Geringschätzung des Betriebssystems unter anderem aufgrund der Schutzkonzepte und der Speicherverwaltung praktisch nicht mehr möglich. Programme sind daher heutzutage »Anwendungen« der Betriebssystemfunktionen (»des Betriebssystems«), weshalb sie häufig auch als solche bezeichnet werden (»32-Bit-Anwendung«).

Prozess

Während ein Programm eine statische Folge von Anweisungen ist, kann ein Prozess als Behälter angesehen werden, der eine Infrastruktur schafft, in der diese statische Anweisungsfolge, das Programm bzw. die Anwendung, ausgeführt werden kann. Hierzu stellt er z.B. dem Programm einen exklusiv für ihn reservierten, virtuellen Adressraum zur Verfügung. In diesen wird das Programm dynamisch abgebildet und es werden die Datenstrukturen eingerichtet, die das Programm definiert (Daten- und Stacksegmente, Heap). Der Prozess erstellt ferner eine Liste von Handles für Systemressourcen (Semaphoren, Kommunikationsanschlüsse wie parallele und serielle Schnittstellen, Pipes etc.) und Dateien, auf die alle Threads eines Prozesses zugreifen können. Schließlich richtet es einen »Sicherheitskontext« (»Zugriffstoken«) ein, der die erforderlichen Sicherheits- und Berechtigungskriterien definiert, die zum Prozess gehören. Alle diese Komponenten zusammen bilden den »Prozess«. Er ist somit die »Umgebung«, in der der Code des Programms ausgeführt wird.



Der Prozess ist der Herr über den gesamten Adressraum, den der Prozessor ansprechen kann und den das Betriebssystem unterstützt! Das bedeutet, dass unter 32-Bit-Betriebssystemen wie Windows 9x/ME/NT/2000/XP und Prozessoren ab dem 80386 der gesamte Adressraum von 4 GByte diesem Prozess virtuell zur Verfügung gestellt wird, sodass jeder Prozess einen »eigenen« 4 GByte-Adressraum besitzt, in den die verschiedenen Komponenten »eingeblendet« werden – inklusive Betriebssystem (Kernel, API, GUI etc.).



In Wirklichkeit wird jedoch der Adressraum in einen »variablen«, Prozess-spezifischen Teil und einen konstanten, »globalen« Teil aufgeteilt. Das Betriebssystem und seine Komponenten sowie global verfügbare Module werden dann im konstanten Teil abgelegt, was verhindert, dass sie bei jedem neuen Prozess neu geladen werden müssen.

Thread Nun ist ein Prozess selbst immer noch nicht lauffähig. Vielmehr muss man sich ihn als Speicherabbild aller Komponenten vorstellen, die dazu benötigt werden, ein Programm tatsächlich »laufen« zu lassen. Die Ausführung des Codes in der Umgebung erfolgt durch einen »*thread*« (engl.: »Faden«). Ohne einen Thread hat ein Prozess keine Existenzberechtigung und wird daher vom Betriebssystem ohne jegliche Rücksicht aus dem Speicher entfernt (was auch der Grund dafür ist, dass ein Prozess automatisch beendet wird, sobald der Haupt-Thread, der bei der Prozesserzeugung mittels *CreateProcess* angelegt wird, beendet wurde und keine weiteren threads existieren). Zu jedem Thread gehören die Inhalte der CPU-Register, die den Status des Prozessors definieren, und verschiedene Thread-eigene Datenstrukturen (je ein Stack für die Privilegstufen 0 und 3 des Prozessors, häufig auch »Kernel-« und »Benutzermodus« genannt, und ein thread-local storage TLS).



Vielleicht hilft Ihnen folgende Analogie, um den Unterschied zwischen Prozess und Thread besser zu verstehen. Wenn Sie z.B. in Delphi ein Objekt definieren (»type TMyObject = object(..) ...; end;«), so teilen Sie dem Compiler lediglich mit, aus welchen Komponenten (Daten, Methoden, Eigenschaften) es besteht, und realisieren diese. Dies entspricht dem Erzeugen eines Prozesses. Doch mit diesem Objekt können Sie noch nichts anfangen, Sie müssen erst eine Instanz erzeugen und diese dann aktivieren. Das bedeutet, wenn Sie mittels *NewObject := TMyObject.Create;* eine Instanz des Objektes erstellt und in den Speicher abgelegt haben (was der Erzeugung eines Threads entspricht), ist es (in der Regel) noch lange nicht aktiv, führt also außer seiner Initialisierung noch keinen Code aus. Erst dann, wenn Sie bestimmte Methoden aufrufen, aktivieren Sie das Objekt (z.B. durch die Methode »Execute« eines Objektes, das Ihnen in einem Fenster die Auswahl einer Datei zum Öffnen ermöglicht). Dies ist gleichbedeutend mit dem Starten des Haupt-Threads eines Prozesses. (Weshalb die Windows-Funktion »CreateProcess« auch besser *CreateProcessAndExecuteMainThread* heißen sollte. Aber das wäre wohl zu lang!)



Übrigens: Ich weiß, dass alle Vergleiche hinken; aber dieser hinkt wenigstens in die richtige Richtung! Und: Ich weiß, dass Objekte in Delphi etwa das Gleiche sind, was Steintafeln, Hammer und Meißel in Druckereien mit Hochleistungscomputern und Lichtsatzmaschinen sind: Schnee aus dem Präkambrium. Beschwerden in diese Richtung sind somit überflüssig ;-)



Genauso, wie Sie verschiedene Instanzen von TMyObject erstellen und (mehr oder weniger) unabhängig voneinander ausführen können, können Sie auch mehrere Threads in einem Prozess erstellen, die dann ebenfalls (mehr oder weniger) unabhängig voneinander ablaufen. Diese Threads führen dann aber alle den gleichen Code in der gleichen Umgebung (= Prozess) aus, haben jedoch ihre eigenen Thread-spezifischen Datenstrukturen (Daten des Objektes).

Ich hoffe, threads sind nun keine *threats* (engl.: »Bedrohungen, Gefahren«) mehr.

Task

Stellen Sie sich nun einen laufenden Prozess vor. Er besteht, wie gesagt, aus einem oder mehreren Threads, aus dem Code, den Daten, dem/den Stack(s) und/oder Heap(s), den Bibliotheken und den Betriebssystemkomponenten. Vergleichen Sie diesen Prozess mit einem beliebigen anderen Prozess, der zu einem anderen Zeitpunkt läuft. Auch dieser besteht aus Code, Daten, Stack(s), Heap(s), Bibliotheken und Betriebssystemkomponenten. Streichen Sie nun von Ihrer imaginären Liste alle die Komponenten, die beiden Prozessen gemeinsam (= »konstant«) sind. Das sind in der Regel die Betriebssystemkomponenten und Bibliotheken, die »global« verfügbar sind. Was übrig bleibt, ist der »variable« Teil, der bei jedem »Taskwechsel« geändert werden muss. Ihn nennt man auch »task«.

Tasks müssen nicht notwendigerweise Anwendungen oder Programme sein, die eine bestimmte Funktion haben und daher im Rahmen des Multitasking regelmäßig solange aufgerufen werden, bis sie beendet werden (Textverarbeitung, Tabellenkalkulation, Druckaufträge). So kann beispielsweise ein Task auch lediglich aus einer Ansammlung von Routinen bestehen, die von jemand anderem aufgerufen werden können. So könnte ein Task Interrupt- und Exception-Handler zur Verfügung stellen, die nur in bestimmten Situationen notwendig sind und dann via task switch über ein task gate durch einen Interrupt gestartet werden.

5.1.5 »Unschärfen« und Ungenauigkeiten in diesem Buch

Selektor

Streng genommen ist der Selektor ein 16-Bit-Word gemäß Abbildung 2.15 auf Seite 429, das aus einem 13-Bit-Wert beginnend mit Bit 3 und endend mit Bit 15, dem »Selektor-Index«, besteht sowie dem Feld RPL (Bits 0 und 1) und dem Flag TI (Bit 2). Der Selektor-Index stellt den Zeiger in die entsprechende Deskriptoren-Tabelle dar (GDT, wenn TI, *table index*, = 0 und aktuelle LDT, wenn TI = 1). Er muss mit $2^3 = 8$ multipliziert werden, damit der tatsächliche Zeiger in die Deskriptortabelle berechnet wird. Praktisch erfolgt dies, indem der vollständige Selektor mit der Maske $FFF8 UND-verknüpft wird. Dadurch werden die Bits 0 bis 2 gelöscht. Das Ergebnis entspricht der Multiplikation des 13-Bit-Selektorindex mit 8. Dieses Vorgehen hat den Vorteil, dass der Selektor immer nur Werte annehmen kann, die ganzzahlige Vielfache von 8 sind. Und das ist auch korrekt so, da ein Deskriptor ja zweimal 32 Bit = 8 Bytes umfasst, Deskriptoren in ihren Tabellen somit immer an Offsets beginnen, die durch 8 restlos teilbar sind. Die damit nicht erforderlichen Bits 0 bis 2 können somit für andere Zwecke benutzt werden: Angabe der zu verwendenden Deskriptoren-Tabelle (TI) und Schutzkonzeptunterstützung (RPL).

In diesem Buch wird jedoch der Begriff Selektor etwas »unschärfer« benutzt. Zum einen wird er in seiner tatsächlichen Definition verwendet, nämlich als 16-Bit-Wert mit 13-Bit-Index, RPL-Feld und Flag. Dies erfolgt immer dann, wenn tatsächlich ein vollständiger Selektor eine Rolle spielt, also wenn z.B. ein Selektor im Rahmen eines Far-Calls als Ziel übergeben wird. In diesem Falle sind alle Selektorfelder von Bedeutung: der Index zur »Berechnung« des Zeigers in die Deskriptoren-Tabelle, TI zu Festlegung der zu verwendenden Deskriptoren-Tabelle und RPL zwecks Privilegienprüfung.

Doch wird auch etwas lax von »Selektor« gesprochen, wenn nur der Selektor-Index bzw. der mit 8 multiplizierte Index als Zeiger in die Deskriptoren-Tabelle gemeint ist. Nämlich dann, wenn tatsächlich nur der Zeiger eine Rolle spielt, z.B. bei der Überprüfung, ob er außerhalb des Segmentlimits liegt.

Der Grund für diese »Unschärfe« liegt einfach darin, dass es für den Uneingeweihten besonders am Anfang recht schwierig ist, den Mechanismus Selektor – Deskriptor – Segment überhaupt zu verstehen und eine Unterscheidung Selektor – Selektorindex eher verwirren als erhellen würde. Daher wurde der Selektor sprachlich auf das »reduziert«,

was ihn hauptsächlich ausmacht: den Index. Die zusätzlichen Informationen werden im Kontext gesehen und an entsprechender Stelle gewürdigt.

Exceptions

In diesem Buch wird davon geredet, dass die CPU »Exceptions auslöst«. Dies ist eigentlich falsch! *Exception* bedeutet auf Hardware-Ebene und somit im Assembler »Ausnahmesituation«. Und eine CPU (oder auch FPU!) kann keine »Ausnahmesituation auslösen«, sondern höchstens eine feststellen und darauf reagieren.

Dennoch wurde die Formulierung »Exception auslösen« ausgiebig benutzt. Der Hintergrund ist einfacher Natur: Wenn Sie Delphi oder C++-Builder kennen, kennen Sie die dort implementierten »Exceptions«. Dies sind in der Sprache implementierte Mechanismen, die bei der Behandlung von Ausnahmesituationen helfen sollen. Als Exception wird hier also der *Mechanismus* bezeichnet, der die Ausnahmesituationen behandelt, nicht die *Ursache*! Und diese Mechanismen werden sehr wohl von der CPU ausgelöst, sobald die dazugehörigen Ausnahmensituationen vorliegen.

Sehen Sie mir daher bitte nach, dass ich auf Kosten der Exaktheit zum Zwecke des »Brückenbauens« zwischen Assembler und Hochsprache die Formulierungen aus der Hochsprache weitgehend übernommen habe.

Prozeduren

Wer von Hochsprachen kommt, kennt ihn – den Unterschied zwischen Routinen und Prozeduren. Der Begriff *Routine* ist der Oberbegriff für Prozeduren und Funktionen. Wer also in Hochsprachen von Routinen spricht, meint entweder eine Funktion oder eine Prozedur.

Der Unterschied zwischen Funktion und Prozedur ist dabei (hoffentlich!) klar: Funktionen haben ein »Funktionsergebnis«, geben also irgendwie ein Ergebnis, ein Datum, dem rufenden Programmteil zurück. Prozeduren tun das eben nicht.

Nun gibt es unter Assembler nur Prozeduren, was bei genauerer Betrachtung noch nicht einmal falsch ist: Der Assembler ruft beide mit CALL, kann beiden Parameter über den Stack übergeben, beiden lokale Variablen zuweisen und schließt beide mit RET ab. Ob nun irgendwann im Verlauf der »Prozedur« irgendein Register oder eine Speicherstelle mit irgendeinem Wert belegt worden ist, der dann in der rufenden Routine weiterverwendet wird, ist ihm vollkommen egal – das ist Sache des Programmierers und hat mit dem »Assemblieren«, also dem Zusam-

mensetzen von Bytes zu Befehlssequenzen und diese dann zu Code- und Datenmodulen, sehr, sehr wenig zu tun!

Ich fände es nun einfach nicht passend, in dem Teil des Buches eine aus Assemblersicht unnatürliche Unterscheidung zwischen Routine und Funktion auf der einen und Prozedur auf der anderen Seite zu machen, und z.B. im Kontext der Direktive PROC, die die Deklaration einer *PROCedure* gestattet, von *Routinen* zu sprechen. Daher habe ich im Teil »Der Stand-Alone-Assembler« ab Seite 557 nur von Prozeduren gesprochen und Routinen gemeint. Allerdings dürfte diese Ungenauigkeit nicht zu großen Problemen bei Ihnen führen.

5.2 Datenformate

Wissen Sie, worin sich eine »packed byte integer« von einer »packed byte integer« unterscheidet? Nein, wir betrachten nicht ihren Wert, sondern ihre Definition! Kann man anstelle von »single-precision floating-point doublewords« auch »single-precision floating-point values« bei Berechnungen einsetzen? Und, wenn ja, bei welchen? Ist ein »double-extended precision floating-point value« nun »double extended« *precise* oder »extended double« *precise*? Im ersten Fall müsste die Zahl doppelt so groß wie eine *extended real* sein. Im zweiten Fall wäre es »lediglich« eine »erweiterte« doppelt-präzise Real und somit etwas größer als besagte – warum dann aber die seltsame Namensverdrehung?

Die Nomenklatur der unterschiedlichen Datenformate wird in unterschiedlichen Publikationen sehr unterschiedlich gehandhabt. Da dies ein Buch sein soll, das die wichtigsten derzeitigen Prozessoren und ihre Möglichkeiten beschreibt, sollte auch eine Sprachregelung gefunden werden, die auf alle anwendbar ist. Daher war es nötig, eine neue Nomenklatur aufzubauen.

Zum einen wollte ich nicht (nur) die Intel-Namen übernehmen. Nicht so sehr, weil ich etwas gegen Intel hätte oder AMD nicht vergraulen wollte. Sondern eher, weil auch Intels Namensgebung, wenn auch höchst akkurat und aussagekräftig, alles andere als »leichtgängig« ist: Wenn man für das Schreiben eines einzigen Datums, das »128 bit packed double-precision floating-point value«, fast eine ganze Zeile benötigt, so kann man nicht davon sprechen, leicht verständlich zu schreiben. Dies zeigt sich auch an solchen Ungetümen wie »*quadword unsigned integer*«. Und in Tabellen ist so etwas schlichtweg nicht nutzbar! Immerhin sind diese Ausdrücke exakter als AMDs »single-preci-

sion floating-point doubleword« – ist das nun eine Fließkommazahl (floating-point), eine Integer (doubleword) oder ein Hybrid aus beiden?

Auch hat (nicht nur) Intel im Laufe seiner Existenz unterschiedliche Namen für die gleichen Daten kreiert – ich denke nur an die guten alten *TempReals*, die heute keiner mehr kennen will und die nun (was unter eben geschildertem Vorbehalt auch durchaus OK ist!) »double-extended precision floating-point values« heißen.

Langer Rede kurzer Sinn: Ich habe mit der Vergangenheit (auch meiner eigenen aus vier vorangehenden Auflagen) teilweise gebrochen, versucht, ein wenig Struktur in die Nomenklatur zu bekommen, und verwende daher in diesem Buch folgende Datenbezeichnungen. Dabei habe ich einerseits auf Konsequenz geachtet (durchgängig weiß man nun, dass die 64-Bit-MMX-Register verwendet werden, wenn »Short-Packed«-Daten eine Rolle spielen – unabhängig davon, ob Integers oder Reals betroffen sind. Die 128-Bit-XMM-Register dagegen beheimaten »Packed«-Daten, ebenfalls Integers oder Reals), andererseits auch auf Klarheit (egal in welcher Form die Elemente eingesetzt werden: Bytes bleiben Bytes und floating-point values sind keine DoubleWords!). Es ist sogar noch Raum für künftige Erweiterungen vorhanden: Wenn ein Prozessorhersteller irgendwann einmal 256-Bit-Register einsetzt, so können wir die »LongPacked«- und bei 1024-Bit-Registern die »ExtendedPacked«-Datenstrukturen definieren. (Der Erweiterung nach oben scheinen keine Grenzen gesetzt: von »Mega« über »Giga« bis »Extreme«, »Monster« und »Killer«-Strukturen ist alles machbar. Und auch zwei Schritte nach hinten könnte man machen: So könnten 32-Bit-Datenstrukturen durchaus als »SmallPacked«-Daten und 16-Bit-Strukturen als »TinyPacked« bezeichnet werden. Das würde aber, vor allem unter dem Gesichtspunkt »packed«, wohl wenig Sinn machen: Es kämen nur »SmallPackedWords«, »SmallPackedBytes« und »TinyPackedBytes« in Betracht. Ob die aber so furchtbar viel an Performance bringen ...)

Die in diesem Buch verwendete Nomenklatur ist einfach, sie stützt sich weitgehend auf Datentypen, die aus Delphi heraus bekannt sein sollten. Ich verwende Delphi selbst sehr gerne, da das zugrunde liegende Pascal von Niklas Wirth ja als Lehrsprache entwickelt wurde und daher recht logisch und konsequent aufgebaut ist:

- Als Elemente von Datenstrukturen kommen die »normalen« Daten zum Einsatz, die sich in der Zwischenzeit bei vielen Programmie-

rern eingebürgert haben. Sie residieren in den üblichen Registern, also entweder in den FPU- oder in den Allzweckregistern der CPU. Wir unterscheiden zwei große Gruppen:

  - die Fließkommazahlen (= Reals) SingleReal, DoubleReal und ExtendedReal, die auch die FPU als Fließkomma-Experte kennt, und
  - die Ganzen Zahlen (= Integers) der CPU, die entweder vorzeichenlos sind und dann Bytes, Words, DoubleWords (oder DWords), QuadWords (oder QWords) und DoubleQuadWords (oder DQWords oder auch logischer OctelWords oder OWords) heißen, oder die ein Vorzeichen besitzen und dann analog als ShortInt, SmallInt, LongInt und QuadInt bezeichnet werden. Da noch nicht eingeführt, gibt es die dem DoubleQuadWord analoge DoubleQuadInt bzw. OctelInt noch nicht, aber sie ist jederzeit definierbar.

- Verlässt man die eingetretenen Pfade der »Elementardaten«, kommen sofort Datenstrukturen aus »gepackten« Elementardaten zum Einsatz. Diese Datenstrukturen werden definiert über die Größe, die sie einnehmen dürfen. Meistens hat dies hardwarebedingte Gründe: Die MMX-Register arbeiten mit 64-Bit-Daten, die XMM-Register mit 128-Bit-Daten. Daher werden zwei Datenstrukturen definiert, die auf diesen Registern aufsetzen:
  - die »ShortPacked«-Datenstrukturen der MMX-Register mit 64 Bit Breite und
  - die »Packed«-Datenstrukturen der XMM-Register mit 128 Bit Breite.
- Allerdings können einige Befehle auch auf »Elementardaten« in diesen Registern angewendet werden. So z.B. auf die »unterste« SingleReal in einer ShortPacked-Struktur. Diese aus der Datenstruktur »herausgerissenen« Daten werden wir als »Scalar« bezeichnen. Wir werden sie zu den »erweiterten« Elementardaten rechnen.

Mit diesen Definitionen nun können wir alle Daten eindeutig, klar und konsequent benennen, die wir brauchen. So ist eine ShortPackedSingleReal eine Datenstruktur aus Fließkommazahlen mit vier Byte Breite (siehe FPU-Format!), die insgesamt 64 Bit umfasst (»ShortPacked«) und damit in den MMX-Registern haust. Automatisch wissen wir damit auch, dass sie nur in AMDs 3DNOW!-Befehlen zum Einsatz kommen kann, da Intel für gepackte Fließkommazahlen die XMM-Register verwendet. Andersherum schließen wir messerscharf auf einen Intel-Pen-

tium-4-Prozessor, wenn wir von PackedQuadWords hören. Denn die »Packed«-Strukturen setzen auf XMM-Registern auf, die AMD (noch?) nicht kennt. Da QuadWords mit 64 Bit Breite verwendet werden, wissen wir sofort, dass diese 128-Bit-Struktur zwei davon aufnehmen kann. Schließlich sagt uns die ScalarDoubleReal eindeutig, dass eine Fließkommazahl mit doppelter Genauigkeit bearbeitet wird, jedoch nicht mittels der FPU-Befehle in den FPU-Registern, sondern mit Hilfe der SIMD-Erweiterungen in Intels XMM-Registern. Und auch bis heute noch vollkommen unbekannte Datenstrukturen können wir mit dieser Systematik enträtseln: Die ExtendedPackedDoubleQuadInts der 1024-Bit-XMMXXL-Register von AMTEL können acht dieser in der Bildtelephonie mittels Armbanduhr äußerst wichtigen DoubleQuadInts aufnehmen und damit die Voraussetzung zur dreidimensionalen Konferenzschaltung mittels Hyperraum-Kanälen schaffen.

5.2.1 »Little-Endian«- und »Big-Endian«-Format

Wie schreibt man Zahlen auf? Nein – lachen Sie jetzt nicht, die Frage ist nicht trivial! Genauer: In welcher Reihenfolge werden die Ziffern einer Zahl notiert? Doch ganz offensichtlich im Gegensatz zu der uns gewohnten Lese- und Schreibrichtung. Denn Wörter lesen wir von links nach rechts, Zahlen aber von rechts nach links!

Also – wenn ein Datum aus mehreren Bytes besteht: Schreiben wir diese Bytes analog unserer Wort-Schreibweise von links nach rechts, sodass der niedrigstwertige Anteil des Datums, das least significant byte oder erste Byte, links steht? Oder folgen wir der Zahlendarstellung, bei der es rechts steht (»erste Ziffer«)?

Solange wir mit »echten« Daten, also QuadWords oder Words umgehen, könnte man ja sagen: analog der Zahlendarstellung von rechts nach links! Aber für den Prozessor sind ja auch Befehlssequenzen Daten, vor allem, wenn er sie aus dem Speicher lesen muss. Und Befehlssequenzen könnte man ja eher als »Sätze« denn als »Daten« auffassen und somit die Darstellung links→rechts bevorzugen. Gibt es also sogar verschiedene Arten der Interpretation?

Ja, denn für den Prozessor ist eine Befehlssequenz ein »Satz«, Daten wie Adressen, Werte oder Konstanten dagegen »Zahlen«!

Und hier die Auflösung: Es gibt beide Methoden. Der Prozessorhersteller Intel hat sich für die Codierrichtung rechts→links entschieden, also ganz analog zu »unserer« Zahlendarstellung. Da bei dieser Darstel-

lungsart das »dicke Ende« zuerst kommt und das »dünne« danach, nennt man sie auch »Little-Endian«-Format oder nach dem Unternehmen, das dies festgeschrieben hat, »Intel-Notation«. Die Firma Motorola ging mit ihren Prozessoren genau den entgegengesetzten Weg: Hier folgt tatsächlich das dicke Ende dem Anfang! Folglich heißt diese Art der Darstellung »Big-Endian«-Format oder »Motorola-Notation«.



Bitte beachten Sie daher, dass bei der Programmierung von IA-32- oder IA-64-Prozessoren (»Intel Architecture«) das Little-Endian-Format gilt. Wichtig ist dies vor allem, wenn nicht, wie in diesem Buch, bei Speicherabbildungen die niedrigste Adresse rechts unten steht und nach links und oben anwächst. So stellen Debugger die niedrigste Adresse links oben dar und lassen Adressen nach rechts unten anwachsen. In diesem Fall sieht das DoubleWord $12345678 so aus: `$78 $56 $34 $12` und die Befehlsfolge $E912345678 (`JMP +12345678h`): `$E9 $78 $56 $34 $12`!



So muss auch codiert werden, wenn Opcodes mit Hilfe der Data-Anweisungen »von Hand« in irgendwelche Quelltexte eingestreut werden. So kennen z.B. nicht alle integrierten Assembler den Befehl CPUID, der aber eine gewisse Bedeutung hat! Will man ihn nutzen, so hat der dazugehörige Opcode $0FA2 via Bytes »normal« codiert zu werden:

```
DB      00Fh    ; erstes Byte des Opcodes
DB      0A2h    ; zweites Byte des Opcodes
```

Als Word codiert dagegen muss die Reihenfolge vertauscht werden:

```
DW      0A20Fh  ; das erste Byte steht rechts!
```

5.2.2 Binäre Zahlendarstellung und Hexadezimalsystem

Wer nicht gewohnt ist, mit Assembler zu arbeiten, hat erfahrungsgemäß am Anfang nicht unerhebliche Schwierigkeiten, sich an die etwas andere Art der Programmierung zu gewöhnen. Zugegeben: Es dauert ein wenig. Aber mit der Zeit geht sie einem »in Fleisch und Blut über«. Ein Teil dieser Anfangsschwierigkeiten sind in der Art und Weise begründet, wie der Prozessor Daten »sieht«: binär. Daher möchte ich an dieser Stelle darauf eingehen, wie man Zahlen in binärer Weise darstellt.

Wenn man die Zahl 4711 schreibt, so implizieren wir etwas! Erinnern wir und an den Mathematikunterricht: Die Schreibweise »4711« ist eine

verkürzte Form der Summe $4 \cdot 10^3 + 7 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0$. Die Ziffern, aus denen »4711« besteht, sind also nichts anderes als Faktoren, mit denen die jeweils der Position der Ziffer in der Zahl entsprechende Potenz der Basis »10« multipliziert werden muss (wobei die Positionszählung bei »0« und nicht bei »1« beginnt). Die verwendete Basis bestimmt hierbei, wie viele unterschiedliche Ziffern als Faktoren erlaubt sind. Im Falle von 10 als Basis sind das 10 unterschiedliche Ziffern, die Ziffern »0« bis »9«.

Nimmt man eine andere Basis, ändert sich grundsätzlich nichts an diesem Sachverhalt: So stehen lediglich zwei Ziffern, »0« und »1«, zur Verfügung, wenn man binär arbeiten möchte. Und diese Ziffern werden als Faktoren benutzt, mit denen die einzelnen Glieder abfallender Potenzen der Basis »2« in der Summe multipliziert werden. So ist 10010110 lediglich die verkürzte Form der Summe $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$.

Will man nun wissen, was 10010110 in »unserem« Dezimalsystem bedeutet, muss man ein wenig rechnen: Zunächst müssen die Potenzen von 2 als Wert zur Basis 10 berechnet werden, die dann mit den jeweiligen Faktoren multipliziert und schließlich aufsummiert werden: $1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 150$. Auch der umgekehrte Weg ist klar, wenn auch ein bisschen mehr Rechenarbeit erforderlich wird, da zunächst die höchste Potenz von 2 gefunden werden muss, die im Dezimalsystem kleiner als die oder gleich der zu konvertierende(n) Zahl ist. So ist bei der Zahl 98 die größte 2er-Potenz, die dies erfüllt, $64 = 2^6$. Zieht man nun diese Potenz von der zu konvertierenden Zahl ab, so bleiben 34 übrig. Mit diesem Rest wird ein neuer Zyklus gestartet: Die größte in 34 passende 2er-Potenz ist $32 = 2^5$. Diese Potenz vom Zwischenergebnis, 34, abgezogen hinterlässt als neues Zwischenergebnis 2. Und dies ist gerade selbst eine größte 2er-Potenz (2^1), sodass an dieser Stelle kein neuer Zyklus notwendig wird. Die Dezimalzahl »98« kann also binär dargestellt werden als $2^6 + 2^5 + 2^1$. Als Summe von abfallenden 2er-Potenzen ausgedrückt heißt das: $1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Lässt man nun die Basen weg, so erhält man für »98« die binäre Darstellung 1100010.



Wie im Dezimalsystem auch, werden führende Nullen vor der ersten (= signifikanten) »1« in der Regel unterdrückt, da sie keinerlei weitergehende Information besitzen. Dennoch ist mein Tipp: Unterdrücken Sie nicht *alle* führenden Nullen und *gruppieren* Sie Binärzahlen. Ich selbst gruppiere in meinen Quelltext-Kommentaren byteweise, was be-

deutet, dass immer 8 binäre Zeichen eine Gruppe bilden. Und was das Thema führende Nullen betrifft: Es erhöht die Lesbarkeit des Quelltextes und seiner Kommentare erheblich, wenn man immer so viele binäre Ziffern angibt, wie das darzustellende Datum nach Definition aufweist. Konkret heißt das, Bytes mit 8 Ziffern darzustellen, Words mit 16, DoubleWords mit 32 usw. Die Differenz der für ein gegebenes Datum maximal möglichen und der zur Darstellung mindestens notwendigen Ziffern ist dann die Anzahl der führenden Nullen.



Im Quelltext für Assembler selbst, also bei der Definition oder beim Einsatz von binär dargestellten Konstanten als Operanden für Befehle, können nur die führenden Nullen Verwendung finden, denn der Assembler ignoriert sie beim Assemblieren schlichtweg. Gruppieren können und dürfen Sie hier nicht, da er jedes nicht als Ziffer geltende Zeichen in Daten heftigst moniert – wie soll er auch wissen, dass es sich um ein von Ihnen als Lesehilfe eingestreutes Gruppierungszeichen handelt? Es könnte ja auch ein Tippfehler sein! Gruppieren Sie daher nur in den Kommentaren des Quelltextes. Doch tun Sie's hier wirklich! Es erhöht die Lesbarkeit, vor allem bei großen Zahlen oder Bit-Feldern.

Ein Beispiel: Die Zahl 98 von oben soll als Byte (und damit mit 8 Ziffern) dargestellt werden, sie benötigt, wie wir gesehen haben, jedoch nur 7. Daher käme hier eine führende Null in Betracht: 01100010. Soll dagegen die Zahl 150 von weiter oben als Word dargestellt werden, so müssen 8 führende Nullen (16 Bits DoubleWord minus 8 Bits mindestens zur Darstellung) verwendet werden. Mit byteweiser Gruppierung sieht 150 binär dann so aus: 00000000_10010111. Dieses Verfahren hat, so man sich konsequent daran hält, den Vorteil, dass man sofort sehen kann, wenn Operand und Datum nicht zueinander passen: Der Befehl MOV AL, 1001001110010111b wird wohl vom Assembler nicht akzeptiert werden, da nur Bytes zugelassen sind (MOV *AL*), aber eine Word-Konstante folgt (16 binäre Stellen).

Hexadezimalsystem

Die binäre Codierung ist aber, will man mit ihr programmieren, recht unhandlich. Schnell artet Programmierung in pure Tipparbeit aus: Bereits eine so »kleine« Zahl wie 1,000 hat schon, binär codiert, recht viele signifikante Ziffern (auch ohne führende Nullen!): 1111101000. Wollte man den Maximalwert eines lächerlichen Words mit 65,535 darstellen, so sähe das bereits so aus: 1111111111111111. Und ein typisches DoubleWord (4,105,764,723) müsste so codiert werden: 11110100101110001111111101110011. Klar, dass das schnell zu Tippfehlern und mehr führen würde.

Daher suchte man eine neue Basis, die zwei Bedingungen erfüllen musste: Sie muss zur binären Darstellung kompatibel sein und deutlich zur Verbesserung der »Lesbarkeit« beitragen. Bedingung 1 ist recht einfach erfüllbar, wenn man als Basis eine Potenz von 2 verwendet. Denn in diesem Fall sind beide Darstellungen schnell und unkompliziert ineinander überführbar, ohne dass groß Rechenarbeit notwendig würde. Daher kamen nur die Basen »4«, »8«, »16« etc. in Frage. Die Basis 4 brachte keine wesentlichen Verbesserungen, was die Erleichterung des Handlings betrifft. Die Basis 8 war da schon besser, weil sie in etwa vergleichbar mit der Basis unseres Dezimalsystems ist. Und aus diesem Grunde haben auch »Oktalzahlen« eine gewisse Zeit lang eine bestimmte Bedeutung gehabt. Doch es gab noch eine andere Basis, die noch bessere Eigenschaften aufwies: die Basis »16«!

Mit dem sog. Hexadezimalsystem sehen nun die eben dargestellten Zahlen fast unscheinbar aus: 3E8, FFFF und F4B8FF73. Dem Rechner ist diese Darstellungsart egal, da eine Hexadezimalzahl eben aufgrund der Verwendung einer Potenz von 2 als Basis nur eine andere bildliche Darstellung einer binär codierten Zahl ist. (So ist das auch bei uns mit unserem Dezimalsystem: Ob wir nun von TDM sprechen und in Einheiten von »Tausend DM« rechnen oder mit Kilogramm, Tonnen oder Milligramm hantieren – in jedem Fall ist eine andere Potenz von 10 Basis und damit eine Konvertierung einfach durch Streichen/Anhängen überflüssiger/zusätzlicher Nullen zu erhalten.)



Um mögliche Verwechslungen mit Systemen, die auf anderen Basen fußen – wie z.B. dem Dezimalsystem –, auszuschließen, wird einer Hexadezimalzahl üblicherweise in Pascal-Dialekten ein »$«-Zeichen, in C-Dialekten ein »0x« vorangestellt oder unter Assembler ein kleines »h« angehängt, einer Dezimalzahl ein »d«, einer Oktalzahl ein »o« und einer Binärzahl ein »b«: 920d = $398 = 0x398 = 398h = 1630o = 1110011000b.



Auch in Assembler-Quelltexten muss diese Datenkennzeichnung erfolgen. Als benutzerfreundliche Schnittstelle zwischen dem (dezimalorientierten) Menschen und dem (binär arbeitenden) Prozessor interpretieren die Assembler nicht gekennzeichnete Daten standardmäßig als Dezimalzahl. Wenn Sie also mit Hexadezimal- oder Binärdaten programmieren, vergessen Sie nicht die Postfixe: Binärzahlen folgt grundsätzlich ein »b« (0100b), Hexadezimalzahlen ein »h« (64h), Oktalzahlen ein »o« (37o) und – optional – Dezimalzahlen ein »d« (4711d). Das »$«-Zeichen aus Pascal-Dialekten ist in eigenständigen (stand-alone) Assemblern genauso verpönt wie das »0x« aus C++-Dialekten und kann

daher nur in den in Delphi bzw. C++ integrierten Assemblern verwendet werden.

Hexadezimal-Ziffern

Die Verwendung von Hexadezimalzahlen führt nun dazu, dass nicht mehr wie im Dezimalsystem 10 Ziffern zur Codierung der Zahl verwendet werden, sondern 16: Die »normalen« Ziffern »0« bis »9« und die *Ziffern* (nicht Buchstaben!) »A« bis »F«. Das aber wiederum bedeutet, dass die im Dezimalsystem zwei Ziffern benötigende Zahl »14« im Hexadezimalsystem mit einer Ziffer dargestellt werden kann und wird: »E«. (Schade eigentlich, dass man bei Einführung des Hexadezimalsystems so wenig kreativ war und nicht »echte« neue Zeichen erschaffen hat, sondern auf die ersten sechs Buchstaben ausgewichen ist!)

Und wie im Falle der Binärzahlen wird zur Darstellung einer Dezimalzahl im Hexadezimalsystem etwas Rechenarbeit benötigt, da die (in Potenzen von 10 vorliegende Dezimal-) Zahl in Potenzen von 16 zerlegt werden muss: $100_d = 96_d + 4_d = 6_h \cdot 16^1 + 4_h \cdot 16^0$. Unter Verwendung der hexadezimalen Ziffern haben wir dadurch die Dezimalzahl 100_d in die Hexadezimalzahl $64 überführt (Sie sehen an dieser Stelle die Notwendigkeit zum und die Bedeutung des »$«, »0x« bzw. »h«!).

Wieso ist nun das Hexadezimalsystem eine »Vereinfachung« in der Darstellung binärer Werte? Ganz einfach: Es verwendet als Basis, wie gesagt, eine Potenz von 2: $2^4 = 16$. Das aber bedeutet, dass jeweils vier binäre Ziffern zu einer Hexadezimalziffer zusammengefasst werden können. Das Vorgehen der »Umwandlung« einer binär dargestellten Zahl in eine hexadezimale Darstellung soll in Abbildung 5.1 gezeigt werden:

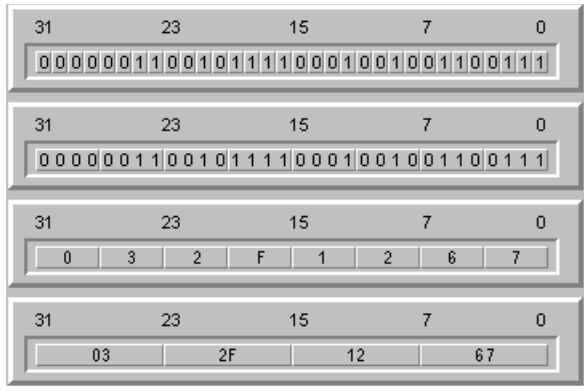


Abbildung 5.1: Zusammenhang zwischen binärer und hexadezimaler Darstellung

Nehmen wir das Beispiel der Zahl 53.416.551 aus Kapitel 1. Sie ist in Abbildung 5.1 oben binär dargestellt. Wenn wir die binären Ziffern nun in einem ersten Schritt gemäß der eben angesprochenen Art und Weise in Vierergruppen einteilen, erhalten wir in der Abbildung die zweite Darstellung von oben. Da nun je vier binäre Ziffern eine Hexadezimalziffer bilden, schreiben wir diese entsprechend um, was in der dritten Darstellung von oben erfolgt ist. Die unterste Darstellung trägt nun lediglich der Tatsache Rechnung, dass vier binäre Ziffern ein Nibble, also ein »halbes« Byte bilden, das es für den Prozessor eigentlich nicht gibt. Daher werden jeweils zwei Hexadezimalziffern (= Nibbles) zusammengruppiert. Und siehe da: Ohne Zauberei und doppelten Boden ist aus dem binären Wurm mit 32 Ziffern in Zeile 1 eine durchaus handhabbare Zahl mit acht Ziffern in Zeile 4 entstanden. Ohne Rechenarbeit!



Wie Sie sofort sehen werden, unterstützt auch diese Darstellung die Interpretation der Registerinhalte, die wir in Kapitel 1 angesprochen haben. Wie soll es auch anders sein, sind doch binäre und hexadezimale Darstellung nur zwei verschiedene Seiten der gleichen Medaille. Wenn Sie in Abbildung 5.1, unten, lediglich das Word-»Register« betrachten (Bits 15 bis 0), so steht dort der hexadezimale Wert $1267. Dies ist $1 \cdot 16^3 + 2 \cdot 16^2 + 6 \cdot 16^1 + 7 \cdot 16^0 = 4096 + 2 \cdot 256 + 6 \cdot 16 + 7 = 4711_d$. Und auch die byteweise Betrachtung liefert das gleiche Resultat: $\$67 = 103_d$!



Es macht auch im Hexadezimalsystem Sinn, führende Nullen und Gruppierung konsequent zu nutzen, vor allem, wenn eine ungerade Anzahl von Hexadezimalziffern zur Darstellung ausreichen (Denken Sie an die »half bytes«! Hexadezimalzahlen wie $3FE oder $A sollten nicht *so* stehen bleiben, sondern besser als $03FE bzw. $0A dargestellt werden). Denn der Vorteil, den diese kostenlose und wenig zeitaufwändige Überprüfungsmethode bringt, überwiegt bei weitem den etwas höheren Tippaufwand: Sie können damit eine wesentliche Quelle schwer aufzufindender Fehler sichtbar machen. Auch bei Hexadezimalzahlen gruppiere ich, hier jedoch wortweise, und fülle anhand der Breite der einzusetzenden Daten mit führenden Nullen auf: $0F, wenn das Byte »15« gemeint ist, oder $0000_4711 beim DoubleWord 18.193.



Und auch an dieser Stelle: Im Quelltext für Assembler können bei Konstanten als Operanden nur die führenden Nullen Verwendung finden. Der Assembler ignoriert sie beim Assemblieren. Gruppieren können und dürfen Sie hier ebenso wenig wie bei der Darstellung binärer Zahlen. Dies bleibt für Kommentare im Quelltext vorbehalten!



Und noch ein Tipp: Der Assembler vermutet hinter jeder Zeichenfolge, die mit einer Ziffer beginnt, ein Datum. Daher beginnt kein Befehl und keine Anweisung mit einem der Zeichen »0« bis »9«, genauso wenig, wie er diese Zeichen am Anfang einer Definitionen jeglicher Art (Variablennamen, Makronamen, Segmentnamen usw.) akzeptiert. Umgekehrt geht er also grundsätzlich davon aus, dass jede Zeichenfolge, die nicht mit einer Ziffer beginnt, kein Datum, sondern der Beginn einer Definition, Anweisung oder eines Befehls ist. In diesem Zusammenhang interpretiert der Assembler (als benutzerfreundliches Instrument) die Hexadezimalziffern »A« bis »F« nicht als Ziffer, sondern als Buchstabe! Daher erhalten Sie Fehlermeldungen, so der Assembler z.B. auf die Hexadezimalzahl F4E3h stoßen sollte. Dies können und müssen Sie umgehen, indem Sie eine führende Null einführen: 0F4E3h wird problemlos akzeptiert. Ich selbst habe mir daher angewöhnt, in Quelltexten neben der oben genannten, datumsorientierten Anzahl führender Nullen immer noch eine zusätzliche zu verwenden, selbst wenn sie nicht erforderlich ist. Das Byte $0F sieht in meinen Quelltexten immer so aus: 00Fh. (In Kommentaren dagegen lasse ich diese zusätzliche Null weg: 0Fh bzw. $0F.)

5.2.3 Elementardaten

Bevor wir auf die Darstellung der Elementardaten zu sprechen kommen können, müssen noch einige Einzelheiten zu ihrer Codierung angesprochen werden!

Codierung von Fließkommazahlen

Fließkommazahlen werden aufgrund ihres Exponenten anders codiert als die weiter unten gezeigten Integers: Sie besitzen zwar wie vorzeichenbehaftete Integers im MSB, dem *most significant bit* (höchstwertigen Bit), ein Vorzeichen; diesem schließt sich aber nicht wie im Falle der Integers direkt die Mantisse, sondern der Exponent an. Erst auf den Exponenten dann folgt die Mantisse, hier dargestellt anhand einer Single-Real.

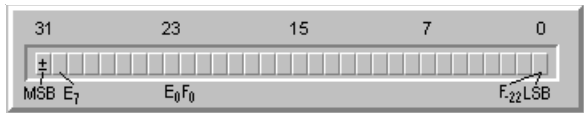


Abbildung 5.2: Bedeutung der Bits bei der Codierung von Fließkommazahlen

Zu beachten ist dabei, dass die Mantisse einen Nachkommateil besitzt, weshalb die Indizierung der Nachkommastellen mit negativen Zahlen erfolgt. Es bedeuten LSB: *least significant bit*, E_X Exponentenziffer X und F_{-X} Ziffer X des Nachkommaanteils der Mantisse (fraction).

Mantisse

Die Darstellung von Mantisse und Exponent ist etwas gewöhnungsbedürftig. Beginnen wir mit der Mantisse. Zwar ist ein gewaltiger Vorteil von Realzahlen, dass ihre Darstellung, was die Mantisse betrifft, geradlinig und »normal« verläuft: Negative Realzahlen unterscheiden sich von positiven mit gleichem absolutem Wert lediglich darin, dass das Vorzeichenbit gesetzt ist. Dies folgt unserer »natürlichen« Art der Darstellung und ist, wie wir noch sehen werden, bei den Integers ganz anders. Das aber war es auch schon mit dem »Normalen«.

J-Bit

So haben zwei der drei definierten Realzahlen keinen »Integer-Anteil«, wie Intel das nennt, also keine Ziffer vor dem Dezimalzeichen (im Intel-Jargon »J-Bit« genannt). Das hat durchaus nachvollziehbare Gründe: Da erstens binär und zweitens mit Exponenten gearbeitet wird (werden kann, muss), bewegt sich die Mantisse grundsätzlich »nur« zwischen 1.0000... und 1.9999... (Denn ein 2.xxxxx kann ja durch Erhöhen und ein 0.yyyyy durch Erniedrigen des Exponenten um 1 auf das Mantissen-Intervall [1.0; 2.0[abgebildet werden.) Damit ist aber die führende (Vorkomma-)»1« redundant – und kann weggelassen oder, was den Sachverhalt genauer trifft, bei Betrachtungen »impliziert« werden. Das hat durchaus Auswirkungen: Man spart so ein Bit, was entweder die Genauigkeit um eine Stelle erhöht oder – je nach Sichtweise – die zur Codierung der Zahl notwendigen Stellen um eine reduziert. Und das kann durchaus beachtliche Auswirkungen haben, wenn z.B. dieses zusätzliche Bit dem Exponenten zugeschlagen wird: Der Wertebereich der darzustellenden Zahl wird dadurch schlichtweg um eine Zweierpotenz erhöht!

Dennoch verbietet einem niemand, dieses redundante MSB der Mantisse trotzdem zu implementieren, was Intel bei der dritten definierten Realzahl auch getan hat. Da deren Exponent bereits astronomische Wertebereiche erlaubt und auch eine sehr hohe Genauigkeit realisiert ist, hat man sich den Luxus erlaubt, das J-Bit explizit zu realisieren. Auf diese Weise werden »außergewöhnliche« Darstellungen möglich, wie sie Intel einmal vorhatte, jedoch aufgrund der Standardisierung wieder aufgeben musste. Wir werden das noch sehen.

Schiefer Exponent

Bleibt noch der Exponent. Er wird üblicherweise als »Schiefer Exponent« oder »biased exponent« codiert. Der Name rührt daher, dass er

um einen Faktor, die »Schiefe« oder »biasing constant« – kurz »bias«, erhöht wird, der gerade ausreicht, um den kleinsten möglichen Wert auf die positive Seite des Zahlenstrahls *hinter die* Null zu bringen. Denn sowohl die »0« beim Exponenten als auch der maximal darstellbare Wert haben eine Sonderbedeutung, auf die wir noch zu sprechen kommen! Damit liegen auch negative Exponenten auf der positiven Seite des Zahlenstrahls, weshalb die Darstellung als »schief« (bezogen auf den Nullpunkt) bezeichnet wird.

Also: Nehmen wir nun an, wir wollten einen binären Exponenten im Bereich von, sagen wir, +127 bis -126 darstellen. Dann wäre unser »Schiefefaktor« 127, damit aus -126 der auf 0 folgende Wert 1 wird. Der »schiefe« Exponent könnte dann, wenn man ihm ein Byte zur Codierung spendiert, Werte zwischen 254 ($FF - 1) und 1 annehmen, wobei 254 »+127« (254 – Faktor) bedeutet, 127 »0« (127 – Faktor) und 1 »-126« (1 - Faktor). Die Werte 0 und 255 haben, wie gesagt, eine Sonderbedeutung.

Oder umgekehrt: Wenn wir erfahren, dass der mögliche Wertebereich des Exponenten einer DoubleReal durch 11 Bits vorgegeben wird (also maximal $7FF betragen kann) und der »bias« $3FF ist, so wissen wir künftig, dass

- erstens $7FF und $000 für Sonderfälle reserviert sind;
- zweitens ein Exponent »0« dargestellt wird durch $3FF (Exponent + Faktor); und
- drittens positive Exponenten im Bereich $3FF (= $7FE - $3FF) und $000 (= $3FF - $3FF) liegen und negative zwischen $000 (= $3FF - $3FF) und -$3FE (= $001 - $3FF). Der positive Exponent einer auf diese Weise codierten Zahl reicht also von $2^{\$3FF} = 2^{1023} \cong 8{,}9885 \cdot 10^{307}$ bis $2^{\$000} = 2^0 = 1$, der negative von $2^{\$000} = 2^0 = 1$ bis $2^{-\$3FE} = 2^{-1022} \cong 2{,}2251 \cdot 10^{-308}$. Zusammen mit einem Maximalwert der Mantisse von 2 (= 1.9999999999...) lässt sich damit die maximal darstellbare Zahl als $2 \cdot 8{,}9885 \cdot 10^{307} = 1{,}7077 \cdot 10^{308}$ und mit einem Minimalwert von 1 für die Mantisse der minimal darstellbare Wert als $2{,}2251 \cdot 10^{-308} = 2{,}2251 \cdot 10^{-308}$ ermitteln. Fertig ist der Wertebereich!

Normale

Doch betrachten wir die Realzahl ein wenig genauer und interessieren uns für die Bitstellungen. Im folgenden Schaubild, das aus Gründen der Übersichtlichkeit nicht den eben für eine DoubleReal dargestellten, sondern den deutlich geringeren Wertebereich einer SingleReal benutzt, soll eine *erhaben* dargestellte Position dafür stehen, dass entweder eine »0« oder eine »1« erlaubt sind, während *eben* dargestellte Posi-

tionen eine bestimmte Vorgabe darstellen. Eine »normale« Zahl lässt sich also so darstellen:

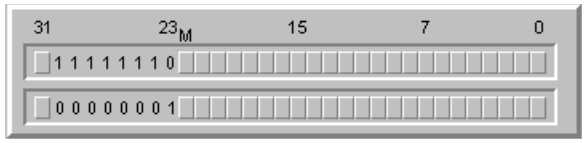


Abbildung 5.3: Speicherabbild des Wertebereiches valider finiter Zahlen am Beispiel einer SingleReal

Der Exponent kann, wie eben gesagt, Werte von \$FE (= 11111110_b; obere Zeile) bis \$01 (= 00000001_b; untere Zeile) annehmen. Das Vorzeichen (Bit 31) ist frei wählbar, ebenso jedes der 23 Mantissenbits der SingleReal. Wie man sieht, sind damit die meisten Bitkombinationen zur Darstellung der »gültigen« Daten reserviert. Lediglich die beiden Sonderfälle für den Exponenten, wenn alle Bits gesetzt (\$FF) oder gelöscht (\$00) sind – was dem Maximal- bzw. Minimalwert des »schiefen« Exponenten entspricht –, werden damit nicht abgedeckt. Diese »reservierten« Bitstellungen werden verwendet, um »Sonderfälle« zu codieren.

Infinite und Null

Die bekanntesten und leicht nachvollziehbaren Sonderfälle sind hierbei die, denen eine vollständig gelöschte Mantisse gemein ist. Im Falle eines Exponenten von \$FF wird damit eine Infinite (Unendlichkeit), im Falle von \$00 der Wert Null codiert. Bitte beachten Sie, dass in beiden Fällen das Vorzeichen frei wählbar ist, was bedeutet, dass es sowohl ± 0 als auch ± ∞ gibt:

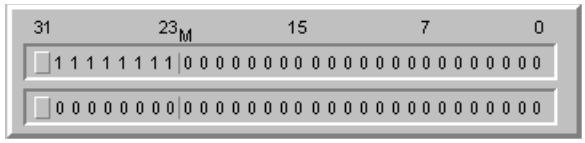


Abbildung 5.4: Speicherabbilder einer »Infiniten« und des Wertes Null am Beispiel einer SingleReal

Diese vier »Sonderfälle« sind in ihrer Bedeutung wohl einsichtig und rechtfertigen daher bereits allein die Reservierung von zwei Exponentenwerten (Maximalwert \$FF und Minimalwert \$00).

M-Bit

Doch es gibt noch weitere gute Gründe für diese »Code«-Bereiche des Exponenten: So sollte man auch darstellen können, dass eine Zahl nicht definiert und damit eine Berechnung nicht erlaubt ist. Der Code für eine solche »undefinierte« Zahl ist: Vorzeichen und alle Exponentenbits sind

gesetzt und das so genannte M-Bit (das ist das erste dem imaginären Trennzeichen folgende Bit) ebenfalls. Alle anderen Bits müssen gelöscht sein!

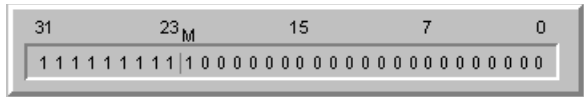


Abbildung 5.5: Speicherabbild einer »Indefiniten« am Beispiel einer SingleReal

Indefinite Diese »Indefinite« (bitte nicht verwechseln mit »Infinite«!) ist ein Spezialfall von Codezahlen, die alle eines gemeinsam haben: Der Exponent enthält den Wert $FF und das M-Bit ist gesetzt:



Abbildung 5.6: Speicherabbild einer »quiet NaN« am Beispiel einer SingleReal

qNaNs Solche »Zahlen« nennt man »quiet NaNs« oder »qNaNs«. NaNs sind *N*ot *a N*umber, also »keine Zahl«! Und als »quiet« werden sie bezeichnet, weil sie sich bei bestimmten Fließkomma-Operationen ganz still verhalten und keinen Laut in Form einer *exception* von sich geben. Das kann unter manchen Umständen sehr sinnvoll sein. So kann man qNaNs z.B. für Diagnostikzwecke missbrauchen: Bestimmte Diagnosetools könnten mit Integerwerten arbeiten, die einfach in qNaNs verpackt sind (die Mantisse ist ja frei nutzbar!) und daher keinen Schaden anrichten können. Bitte beachten Sie, dass das Vorzeichen hierbei keine Rolle spielt, es also »positive« und »negative« qNaNs (aber nur eine einzige Indefinite!) gibt.

sNaNs Lässt man dagegen das M-Bit »0« sein, so hat man ebenfalls eine Menge von Zahlen, die nach der Definition *not a number* sind. Man nennt sie »*signalling*« NaNs oder »sNaNs«. Wie die qNaNs mit der Indefiniten haben auch die sNaNs einen Spezialfall: die Infiniten, die wir schon kennen. Bei ihnen sind alle Bits der Mantisse »0«. Diese NaN-Gattung macht ihrem Unmut über nicht erlaubte Operationen deutlich durch *exceptions* Luft.



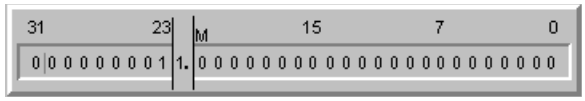
Abbildung 5.7: Speicherabbild einer »signalling NaN« am Beispiel einer SingleReal

MMX

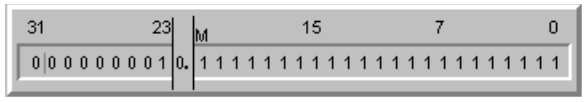
Solche NaNs spielen auch unter MMX eine bedeutende Rolle. Indem die Bits 79 bis 64 der FPU-Register unter MMX auf 1 gesetzt sind, sind alle MMX-Daten definitionsgemäß negative NaNs. Sie signalisieren der FPU damit, dass mit ihnen nicht im üblichen (FPU-)Sinn zu verfahren ist. Doch wie erkennt die FPU, ob nun ein gültiges MMX-Datum vorliegt oder eine »echte« FPU-NaN? Durch den Eintrag im Tag-Feld! Wie Sie bereits wissen, trägt jeder MMX-Befehl außer *EMMS* den Wert 00_b (= »gültig«) in die zu den entsprechenden Registern gehörigen Bits des Tag-Words ein. Somit wird ein MMX-Datum als »gültige NaN« gehandelt. Die FPU-Befehle tun das nicht: Je nach Ergebnis der Operation wird das korrespondierende Tag-Feld des Registers auf den Wert 10_b (= »special«) gesetzt, sobald eine NaN aufgetreten ist oder als Operator verwendet werden soll. »Echte« NaNs sind somit »special NaNs«, wenn Sie so wollen, und »gültige NaNs« sind MMX-Daten.

Null-Exponent

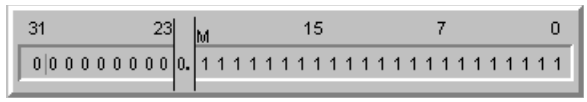
Auch auf der anderen Extremseite des Exponenten gibt es noch Bitstellungen, die nicht besprochen worden sind. Um sie zu erklären, müssen wir ein wenig ausholen – und fügen für einen kurzen Moment in den unteren Teil der Abbildung 5.3 auf Seite 791 das bei den bislang betrachteten SingleReals nur implizit vorhandene Vorkomma-Bit explizit zwischen die Bits 22 und 23 ein:



Bitte beachten Sie, dass die kleinste (normal) darstellbare Zahl codiert wurde: Der Exponent hat den kleinsten »normalen« schiefen Wert und die Mantisse mit 1.0 steht ebenfalls am »unteren Anschlag«. Nun ziehen wir den formal kleinsten darstellbaren Mantissenwert (= 00000000000000000001_b) ab:



Dies ist aber nur möglich, wenn der Exponent um eins erniedrigt wird, da der Übertrag real irgendwo bleiben muss – der Vorkommateil ist ja nur virtuell vorhanden! Somit erhalten wir den untersten Extremwert für den Exponenten, also den, den wir als einen Sonderfall bezeichnet haben.



Wenn wir nun das virtuelle Vorkommabit dorthin schicken, wo wir es hergeholt haben – ins Nirwana –, so wird sichtbar, dass wir durch eine einfache mathematische Operation aus normalen Zahlen »Zahlen« erhalten haben, die nach unserer Definition von oben »Sonderfälle« sind und allgemein so dargestellt werden:

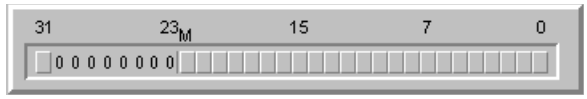


Abbildung 5.8: Speicherabbildung einer Denormalen am Beispiel einer SingleReal



Bitte beachten Sie, dass dieses bei der zur Darstellung gewählten SingleReal virtuelle Vorkommabit bei ExtendedReals alles andere als virtuell, sondern sehr real ist. Es heißt hier J-Bit und ist Teil der Mantisse!

Denormale

Zurück zu den Sonderfällen! Was sind das für Zahlen? Antwort: Ganz normale Zahlen, mit denen man, wie wir gesehen haben, rechnen und arbeiten kann. Denn was wir gemacht haben, ist ja gar nicht so weltfremd und lässt sich am Beispiel $1.23456789 \cdot 10^0$ nachvollziehen. Hier könnten wir anstelle des unter wissenschaftlicher Darstellung (mit Mantisse und Exponenten) üblichen Vorgehens, in der bei Divisionen durch 10 der Exponent erniedrigt und die Mantisse beibehalten wird, die Mantisse des Ausdrucks durch 10 dividieren und den Exponenten unangetastet lassen: $0.123456789 \cdot 10^0$. Das ist zwar unüblich, aber legitim, wie jeder sofort einsehen wird, und drückt einen der eben von uns angesprochenen Spezialfälle aus, in dem eine »normale« Zahl vorliegt, bei der lediglich der Vorkommateil 0 ist. Sie ist ganz offensichtlich gültig und man kann mit ihr rechnen! Aber sie entspricht nicht der »normalen« Darstellung. Daher nennt man sie »denormal«. Denormale zeichnen sich also dadurch aus, dass das Vorkommabit Null ist – was beim Fehlen eines solchen Vorkommabits durch den Exponenten simuliert wird: Hat der Exponent den »schiefen« Wert Null, heißt das, das Vorkommabit ist ebenfalls Null, ansonsten Eins.



Denormale sind somit Zahlen, deren Wert kleiner als der kleinste »normal« darstellbare Wert einer Realzahl ist. Intel nennt sie »winzig« (»tiny«). Ihre Extremwerte sind die weiter oben schon angesprochenen

Werte, bei denen Exponent und Mantisse vollständig Null sind: die beiden Nulldarstellungen. Nun könnte man sagen, dass solche Winzlinge ja durchaus abgerundet werden könnten. Denn in gewisser Weise signalisieren sie ja durch ihre bloße Existenz, dass der gültige Wertebereich unterschritten wurde, also das aufgetreten ist, was als *underflow* bezeichnet wird und eine *exception* auszulösen hat – und es unter gewissen Umständen auch tut (vgl. das Kapitel »FPU-Exceptions« auf Seite 529)! Die Gleichsetzung eines solchen Wertes mit »Null« scheint also mehr als gerechtfertigt. Dies wird noch unterstützt durch die Tatsache, dass der Zugewinn an Wertebereich, den die Denormalisierung mit sich bringt, ja durch einen Verlust an Genauigkeit erkauft werden muss. Denn für jede Erniedrigung des Exponenten über den minimal darstellbaren Wert hinaus muss die Mantisse um eine führende Null vor der ersten signifikanten Ziffer ergänzt werden. Also: besser abrunden!?!

Frei nach Radio Erewan: Im Prinzip ja – aber wer sagt, dass nicht die nächste Operation den Zwerg wieder aus seiner Winzigkeit erlösen könnte? Und $0.000000004 \cdot 1{,}234{,}567{,}890$ ist eben nicht das Gleiche wie $0 \cdot 1{,}234{,}567{,}890$! Daher bleibt es im Ermessen des Programmierers, ob er im Rahmen einer *underflow exception*, so er die überhaupt zulässt, mit den Winzlingen weiter rechnen möchte oder nicht. Er muss nicht, kann aber, wenn es Sinn macht. (Wie Sie bei der Besprechung der betroffenen Instruktionen sehen, gibt es ja durchaus die Möglichkeit, Denormale als »Null« zu betrachten. Die Flags DAZ – *denormals are zero* – und FZ – *flush to zero* – im MXCS-Register des XMM-Registersatzes steuern dies bei den SIMD-Reals.)

Übrigens: Zumindest uns »biologischen Systemen« sind diese Denormalen gar nicht so fremd! Vielmehr rechnen wir permanent mit ihnen. Ja mehr noch: Die Art der Zahlendarstellung der »Siliziumsysteme« mit ihrer Exponentendarstellung ist uns »unangenehmer« als die mit Denormalen. Oder verlangen Sie vielleicht beim Fleischer $1.25 \cdot 10^{-1}$ Kilogramm Wurst? Nein: wohl eher ein Viertel Pfund. Und was ein Viertel ist, weiß jeder Abc-Schütze: 0,25! Eine Tausendstel Sekunde (oder eben 0,001 und nicht etwa $1.0 \cdot 10^{-3}$ Sekunden) sind heute riesige Unterschiede, die über Platz 1 bis 3 auf dem Siegertreppchen aller Sportarten entscheiden. Und wie wenig ein Mikrogramm eines Wirkstoffes in Medikamenten ist, macht man sich mit 0,000001 Gramm erheblich besser klar als mit $1.0 \cdot 10^{-6}$ Gramm – oder? Auch der damit verbundene Verlust an Genauigkeit ist uns mehr oder weniger bewusst, wenn wir z.B. unscharf von »im Promillebereich« sprechen und Bruchteile von Prozenten meinen.

Tabelle 5.2 stellt die eben besprochenen »Zahlen« nochmals zusammen.

Bedeutung		S	Exponent	Mantisse	
				J	M
negative quiet NaN	»größte« qNaN	1	1 1 ..... 1 1	1	1 1 1..... 1 1 1
	»kleinste« qNaN	1	1 1 ..... 1 1	1	1 0 0 ..... 0 0 1
»indefinite«	»Undefinierte«	1	1 1 ..... 1 1	1	1 0 0 ..... 0 0 0
negative signalling NaN	»größte« sNaN	1	1 1 ..... 1 1	1	0 1 1 ..... 1 1 1
	»kleinste« sNaN	1	1 1 ..... 1 1	1	0 0 0 ..... 0 0 1
»negative infinite«	»minus Unendlich«	1	1 1 ..... 1 1	1	0 0 0 ..... 0 0 0
negative »normals«	größte Zahl	1	1 1 ..... 1 1	1	1 1 1 ..... 1 1 1
	kleinste Zahl	1	0 0 ..... 0 0	1	0 0 0 ..... 0 0 0
negative »denormals«	größte Denormale	1	0 0 ..... 0 0	0	1 1 1 ..... 1 1 1
	kleinste Denormale	1	0 0 ..... 0 0	0	0 0 0 ..... 0 0 1
zero	»minus Null«	1	0 0 ..... 0 0	0	0 0 0 ..... 0 0 0
	»plus Null«	0	0 0 ..... 0 0	0	0 0 0 ..... 0 0 0
positive »denormals«	kleinste Denormale	0	0 0 ..... 0 0	0	0 0 0 ..... 0 0 1
	größte Denormale	0	0 0 ..... 0 0	0	1 1 1 ..... 1 1 1
positive »normals«	kleinste Zahl	0	0 0 ..... 0 0	1	0 0 0 ..... 0 0 0
	größte Zahl	0	1 1 ..... 1 1	1	1 1 1 ..... 1 1 1
»positive infinite«	»plus Unendlich«	0	1 1 ..... 1 1	1	0 0 0 ..... 0 0 0
positive signalling NaNs	»kleinste« sNaN	0	1 1 ..... 1 1	1	0 0 0 ..... 0 0 1
	»größte« sNaN	0	1 1 ..... 1 1	1	0 1 1 ..... 1 1 1
positive quiet NaNs	»kleinste« qNaN	0	1 1 ..... 1 1	1	1 0 0 ..... 0 0 0
	»größte« qNaN	0	1 1 ..... 1 1	1	1 1 1 ..... 1 1 1

Hinweis: Die grau unterlegte Spalte zeigt das sog. J-Bit, also das »Vorkomma-Bit« der Mantisse, das nicht bei allen Realzahlen explizit vorhanden ist. SingleReals und DoubleReals besitzen nur ein impliziertes J-Bit, das genau dann formal »0« ist, wenn der Exponent den »schiefen« Wert »0« hat. Andernfalls ist es formal »1«. Bei ExtendedReals dagegen kann das J-Bit explizit verändert werden und nimmt dann die angegebenen Werte an.
In der Tabelle bedeuten: »S«: Vorzeichen; »J«: J-Bit – binäre Ziffer vor dem imaginären Mantissen-Trennzeichen; »M«: M-Bit – erstes, auf das imaginäre Trennzeichen folgendes Bit.
ExtendedReals haben einen Exponenten mit 15 Bits und eine Mantisse (inkl. J-Bit) mit 64 Bits; DoubleReals besitzen einen 11-Bit-Exponenten und (exkl. J-Bit) eine 52-Bit-Mantisse; SingleReals weisen in ihrem Exponenten 8 Bits und in ihrer Mantisse (exkl. J-Bit) 23 Bit auf.

Tabelle 5.2: Bitstellungen und deren Bedeutung bei Fließkommazahlen

Unendlichkeit

Ein zu den Denormalen analoges Problem sind die Unendlichkeiten! Auch das sind, zumindest nach Definition durch IEEE und gemäß der Umsetzung durch die Prozessorhersteller, »echte« Zahlen, mit denen man rechnen kann. (Mein Mathematiklehrer in der Schule würde mich für diese Feststellung noch nachträglich durch das Abitur fallen lassen!) Sie stellen, wie Intel das sagt, die maximal darstellbare Zahl dar, die man mit anderen Zahlen vergleichen kann und die sogar ein Vorzeichen besitzen. (Ich muss gestehen, ein bisschen komisch fühle ich mich schon,

wenn ich schreibe, dass ein »Unendlich« eine Zahl ist!) Das Rechnen mit Unendlichkeiten ist damit genauso erlaubt wie das mit Denormalen – und es ist (brrrr ...) »korrekt«! Allerdings signalisiert es wie die Denormalen die Unterschreitung, eine Ausnahmesituation: die Überschreitung des gültigen Wertebereiches. Was auch zu einer *exception* führen kann und zu der mit ihr verbundenen, schwerwiegenden Frage im *exception handler*: Mache ich weiter oder nicht – und, wenn ja, wie?



Wie Sie gesehen haben, können vorzeichenbehaftete Unendlichkeiten codiert werden, da das Vorzeichenbit existiert und frei verfügbar ist. Somit unterscheiden sich -∞ und +∞ voneinander. Das Zahlenmodell, in dem dies der Fall ist, nennt man *affines* Modell. Dem steht das *projektive* Zahlenmodell gegenüber, bei dem der Zahlenstrahl »im Unendlichen« zum Kreis geschlossen wird, die beiden Unendlichkeiten also auf einem Punkt zu liegen kommen und somit nicht mehr voneinander zu unterscheiden sind. Alle FPUs und NPXe ab dem 80387 benutzen nur noch das affine Modell. Der 80287 und sein Vorgänger kannten auch noch das projektive Modell und konnten über das IC-Flag (*infinity control*) des *control registers* dazu gebracht werden, das entsprechende Modell zu verwenden. Das projektive Modell fiel, wie so manches, der Standardisierung durch IEEE zum Opfer.

Pseudo-Zahlen

Es wurde bereits mehrfach darauf hingewiesen, dass das J-Bit nur in ExtendedReals tatsächlich physisch vorhanden ist. Was also, wenn es vorhanden ist und man es verändern kann? Dann haben wir die absolut identische Situation wie eben – nur dass das J-Bit zusätzlich einen komplementären Wert annehmen kann. Dem wird Rechnung getragen, indem man in diesem Fall den Zahlen den Präfix »Pseudo-« voranstellt. So heißen bei gelöschtem J-Bit die sNaNs mit gelöschtem M-Bit *pseudo signalling NaNs* (»psNaN«) und die mit gesetztem M-Bit *pseudo quiet NaNs* (»pqNaN«). Damit gibt es auch eine Pseudo-Indefinite, Pseudo-Unendlichkeiten (»*pseudo infinites*«) sowie Pseudo-Denormale und die Pseudo-Nullen mit gesetztem J-Bit. Die im Falle eines gelöschten J-Bits zu erhaltenden » Pseudo«-Analoga zu den »normalen« Zahlen werden »unnormal« genannt.



Diese Pseudo-Varianten sind nach IEEE *nicht* definiert und sollten daher auch nicht verwendet werden. Sie werden hier auch lediglich der Vollständigkeit halber erwähnt. Ihre Existenz verdanken sie überhaupt nur der Tatsache, dass die für Standardisierungsfragen zuständigen Gremien wie üblich der Realität hinterherhinkten, sodass im Computer-Pleistozän um den 80287 herum (der eine oder andere Dinosaurier

wie ich wird sich noch an diesen Co-Prozessor erinnern!) die Chiphersteller schneller Hardware-Fakten geschaffen haben, als es Funktionären beim Erstellen des IEEE Std. 754 lieb war. Dies führte dazu, dass die Pseudo-Zahlen im 80287 realisiert waren, im 80387 oder den FPUs ab dem 80486 dann wieder nicht mehr. Die Pentiums unterstützten sie zwar auch nicht mehr, »kannten« sie aber wieder in der Art und Weise, dass sie sie mehr oder weniger duldeten oder besser gesagt ihre Existenz ignorierten und nicht beklagten. Auf Einzelheiten möchte ich hier nicht weiter eingehen. Mein Rat: Lassen Sie die Finger von diesen »Pseudo-Zahlen«, Sie würden sich nur das Leben aufgrund der Inkompatibilitäten einzelner Prozessoren unnötig erschweren.

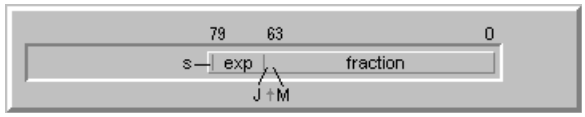
Bedeutung		S	Exponent	Mantisse	
				J	M
negative pseudo quiet NaN	»größte« Pseudo-qNaN	1	1 1 ..... 1 1	0	1 1 1..... 1 1 1
	»kleinste« Pseudo-qNaN	1	1 1 ..... 1 1	0	1 0 0 ..... 0 0 1
»pseudo indefinite«	»Pseudo-Undefinierte«	1	1 1 ..... 1 1	0	1 0 0 ..... 0 0 0
neg. pseudo signalling NaN	»größte« Pseudo-sNaN	1	1 1 ..... 1 1	0	0 1 1 ..... 1 1 1
	»kleinste« Pseudo-sNaN	1	1 1 ..... 1 1	0	0 0 0 ..... 0 0 1
»negative pseudo infinite«	»minus Pseudounendlich«	1	1 1 ..... 1 1	0	0 0 0 ..... 0 0 0
negative »unnormals«	größte unnormale Zahl	1	1 1 ..... 1 1	0	1 1 1 ..... 1 1 1
	kleinste unnormale Zahl	1	0 0 ..... 0 0	0	0 0 0 ..... 0 0 0
neg. »pseudo denormals«	größte Pseudodenormale	1	0 0 ..... 0 0	1	1 1 1 ..... 1 1 1
	kleinste Pseudodenormale	1	0 0 ..... 0 0	1	0 0 0 ..... 0 0 1
pseudo zero	»minus Pseudo-Null«	1	0 0 ..... 0 0	1	0 0 0 ..... 0 0 0
	»plus Pseudo-Null«	0	0 0 ..... 0 0	1	0 0 0 ..... 0 0 0
pos. »pseudo denormals«	kleinste P.-Denormale	0	0 0 ..... 0 0	1	0 0 0 ..... 0 0 1
	größte Pseudodenormale	0	0 0 ..... 0 0	1	1 1 1 ..... 1 1 1
positive »unnormals«	kleinste unnormale Zahl	0	0 0 ..... 0 0	0	0 0 0 ..... 0 0 1
	größte unnormale Zahl	0	1 1 ..... 1 1	0	1 1 1 ..... 1 1 1
»positive pseudo infinite«	»plus Pseudounendlich«	0	1 1 ..... 1 1	0	0 0 0 ..... 0 0 0
pos. pseudo signalling NaNs	»kleinste« Pseudo-sNaN	0	1 1 ..... 1 1	0	0 0 0 ..... 0 0 1
	»größte« Pseudo-sNaN	0	1 1 ..... 1 1	0	0 1 1 ..... 1 1 1
positive pseudo quiet NaNs	»kleinste« Pseudo-qNaN	0	1 1 ..... 1 1	0	1 0 0 ..... 0 0 0
	»größte« Pseudo-qNaN	0	1 1 ..... 1 1	0	1 1 1 ..... 1 1 1

Hinweis: Diese Bitstellungen sind nur bei »Zahlen« möglich, bei denen das sog. J-Bit (grau unterlegt), also das »Vorkomma-Bit« der Mantisse, explizit vorhanden ist und daher den Wert »0« annehmen kann. Dies ist nur bei den ExtendedReals der Fall. Sie unterscheiden sich von den in Tabelle 5.2 dargestellten »echten« Zahlen durch ein jeweils »umgedrehtes« (= negiertes) J-Bit.
In der Tabelle bedeuten: »S«: Vorzeichen; »J«: J-Bit – binäre Ziffer vor dem imaginären Dezimalzeichen; »M«: M-Bit – erstes, auf das imaginäre Dezimalzeichen folgendes Bit.
Die Exponenten der ExtendedReals haben 15 Bits und die Mantissen (inkl. J-Bit) 64 Bits Umfang.

Tabelle 5.3: Nach IEEE Std. 754 nicht unterstützte »Pseudo«-Fließkommazahlen

ExtendedReal

Die FPU-Realzahl *ExtendedReal* besitzt neben einem Vorzeichen und einer Mantisse mit 64 signifikanten binären Stellen einen 15-Bit-Exponenten. Sie ist auch die einzige Realzahl, bei der das J-Bit als Bit 63 explizit vorhanden ist und (zumindest theoretisch) verändert werden kann!



s: Vorzeichen (sign); exp: Exponent; fraction: Nachkommaziffern; J: Vorkommastelle (J bit); M: erste Nachkommastelle (M bit). Der graue Pfeil gibt die Position des imaginären Trennzeichens zwischen Vor- und Nachkommateil der Mantisse an.

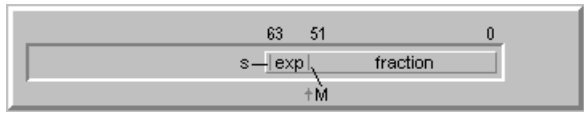
Abbildung 5.9: Speicherabbild einer ExtendedReal

	binär		**dezimal**	
Wertebereich:	$\pm 1.9999 \cdot 2^{\$3FFF}$ bis $\pm 1.0000 \cdot 2^{-\$3FFE}$		$\approx \pm 1.18973 \cdot 10^{4932}$ bis $\approx \pm 3.3621 \cdot 10^{-4932}$	
Mantisse				
Genauigkeit:	64 »echte« Stellen (explizites J-Bit)		19 Stellen	
Exponent	positiv	negativ	positiv	negativ
mit Bias $3FFF:	$7FFE - $3FFF	$3FFF - $0001		
unbiased:	$3FFF - $0000	$0000 - -$3FFE	$2^{16,383}$ bis 2^{0}	2^{0} bis $2^{-16,382}$
max/min:			$5.9486 \cdot 10^{4931}$ bis 1	1 bis $3{,}362 \cdot 10^{-4932}$

Tabelle 5.4: Charakteristika einer ExtendedReal

Die ExtendedReal ist das interne Format, mit dem alle Operationen in den FPU-Registern ablaufen. Durch die entsprechenden Ladebefehle werden SingleReals und DoubleReals vor dem Laden in das FPU-Register in das ExtendedReal-Format überführt bzw. vor dem Ablegen in den Speicher aus dem ExtendedReal-Format konvertiert. Der sehr große Wertebereich und die im Vergleich zu einer DoubleReal nur unwesentlich höhere Genauigkeit dienen bei den internen Berechnungen dazu, Ungenauigkeiten, wie sie bei der Bearbeitung von DoubleReals auftreten können, zu vermeiden. Die Kombination einer DoubleReal im Speicher und die interne Verarbeitung im Rahmen von ExtendedReals sichern eine hohe Sicherheit bei Berechnungen bei gleichzeitiger Kompatibilität mit anderen Datenformaten (LongInt bzw. QuadWord – Stichwort: »alignment«).

DoubleReal Die FPU-Realzahl *DoubleReal* besitzt neben einem Vorzeichen und einer Mantisse mit 53 signifikanten binären Stellen einen 11-Bit-Exponenten. Das J-Bit (Ziffer vor dem imaginären Dezimalpunkt) als 53ste Ziffer ist nicht vorhanden und wird nur impliziert.



s: Vorzeichen (sign); exp: Exponent: fraction: Nachkommaziffern; M: erste Nachkommastelle (M bit). Der graue Pfeil gibt die Position des imaginären Trennzeichens zwischen implizitem Vor- und realem Nachkommateil der Mantisse an.

Abbildung 5.10: Speicherabbild einer DoubleReal

	binär		dezimal	
Wertebereich:	$\pm 1.99999 \cdot 2^{\$3FF}$ bis $\pm 1.0000 \cdot 2^{-\$3FE}$		$\approx \pm 1.79769 \cdot 10^{308}$ bis $\approx \pm 2.22507 \cdot 10^{-308}$	
Mantisse				
Genauigkeit:	53 Stellen (implizites J-Bit)		15 Stellen	
Exponent	positiv	negativ	positiv	negativ
mit Bias $3FFF:	$7FE bis $3FF	$3FF bis $001		
unbiased:	$3FF bis $000	$000 bis -$3FE	$2^{1,023}$ bis 2^{0}	2^{0} bis $2^{-1,022}$
max/min:			$8.9885 \cdot 10^{307}$ bis 1	1 bis $2.2251 \cdot 10^{-308}$

Tabelle 5.5: Charakteristika einer DoubleReal

Die DoubleReal existiert lediglich im Speicher! Sie ist das »Standardformat« für Realzahlen, da sie ein für die meisten Fälle mehr als ausreichendes, ausgewogenes Verhältnis von Wertebereich und Genauigkeit aufweist. Sie ist damit im Normalfall sowohl einer ExtendedReal wie auch einer SingleReal vorzuziehen. Beim Laden in ein FPU-Register wird sie in das interne Format, eine ExtendedReal umgewandelt. Umgekehrt erfolgt eine Konvertierung aus dem internen Format in das DoubleReal-Format, wenn sie gespeichert wird.

SingleReal Die FPU-Realzahl *SingleReal* besitzt neben einem Vorzeichen und einer Mantisse mit 24 signifikanten binären Stellen einen 8-Bit-Exponenten. Das J-Bit (Ziffer vor dem imaginären Dezimalpunkt) als 24ste Ziffer ist nicht vorhanden und wird nur impliziert.



s: Vorzeichen (sign); exp: Exponent; fraction: Nachkommaziffern; M: erste Nachkommastelle (M bit). Der graue Pfeil gibt die Position des imaginären Trennzeichens zwischen implizitem Vor- und realem Nachkommateil der Mantisse an.

Abbildung 5.11: Speicherabbild einer SingleReal

	binär		**dezimal**	
Wertebereich:	$\pm 1.99999 \cdot 2^{\$7F}$ bis $\pm 1.0000 \cdot 2^{-\$7E}$		$\approx \pm 3.40282 \cdot 10^{38}$ bis $\approx \pm 1.1755 \cdot 10^{-38}$	
Mantisse				
Genauigkeit:	24 Stellen (implizites J-Bit)		7 Stellen	
Exponent	positiv	negativ	positiv	negativ
mit Bias $3FFF:	$FE bis $7F	$7F bis $001		
unbiased:	$7F bis $00	$00 bis -$7E	$2^{127} - 2^{0}$	$2^{0} - 2^{-126}$
max/min:			$1.7014 \cdot 10^{38} - 1$	$1 - 1.1755 \cdot 10^{-38}$

Tabelle 5.6: Charakteristika einer SingleReal

Die SingleReal existiert lediglich im Speicher! Sie stellt die kleinstmögliche Darstellung einer Realzahl dar und ist aufgrund des nicht sehr großen Wertebereiches und der nicht besonders großen Genauigkeit nicht für den »Normalfall« geeignet. Beim Laden in ein FPU-Register wird sie in das interne Format, eine ExtendedReal umgewandelt. Umgekehrt erfolgt eine Konvertierung aus dem internen Format in das SingleReal-Format, wenn sie gespeichert wird.

Codierung von Integers

Verglichen mit der Art, wie Fließkommazahlen codiert werden, ist die Codierung von Integers banal! Soweit Integers überhaupt über ein Vorzeichen verfügen, stellt es das *most significant bit* oder »MSB« der Zahl dar, also das Bit mit der höchsten Positionsnummer.

vorzeichenbehaftete Zahlen

Gemäß der Intel-Konvention ist dies die an der linkesten Position der Zahl befindliche Stelle: Die Nummerierung beginnt beim »least significant bit«, dem LSB an Position 0 am rechten Ende, und schreitet nach links in Richtung MSB voran, hier am Beispiel einer LongInt gezeigt, wobei I_x die x-te Ziffer der Integer bezeichnet:

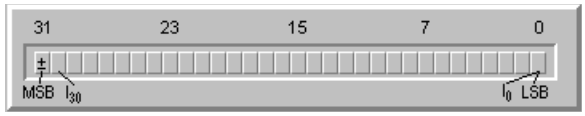


Abbildung 5.12: Bedeutung der Bits bei der Codierung von Integers

Das bedeutet, dass bei einer Integer mit dem Wert 1 alle Bits bis auf das LSB gelöscht sind, während die Zahl 2,147,483,647 bis auf das MSB alle Bits gesetzt hat. Etwas gewöhnungsbedürftig ist hierbei, dass negative Werte nicht wie in Realzahlen simpel durch Setzen des Vorzeichenbits aus positiven gebildet werden, so wie wir das wie »im wirklichen Leben« durch das einfache Voransetzen des Vorzeichens tun: Wenn in Word-Darstellung 2 = 0010_b ist, ist -2 *nicht* 1010_b! Vielmehr ist es das *2er-Komplement*, das entsteht, wenn man den absoluten Wert von 0 abzieht. Die korrekte Darstellung von -2 ist damit, wiederum in Word-Darstellung, 1110_b, da sich beide Werte zu Null addieren: $1110_b + 0010_b = 0000_b$, wenn man den Überlauf vernachlässigt. Handelt es sich daher bei der Integer um die Zahl -2,147,483,647, so ist neben dem Vorzeichenbit kein weiteres gesetzt, während sich -1 durch 32 gesetzte Bits auszeichnet.

Zweierkomplement

Tipp: Zur Bildung des 2er-Komplements brauchen Sie nur alle gesetzten Bits einer Zahl zu löschen und alle gelöschten Bits zu setzen (Bits »negieren«). Zu dem Resultat addieren Sie dann noch »1«, wie am Beispiel der Zweierkomplementbildung von »2«im Word-Format gezeigt wird:

$$00000000_00000010 \xrightarrow{\text{(Negierung)}} 11111111_11111101 \xrightarrow{(+1)} 11111111_11111110$$

sign extension

Und noch ein Tipp: Häufig ist man genötigt, eine Zahl »zu erweitern«, also ihren Wertebereich auszudehnen (die CPU verfügt ja sogar über entsprechende Konvertierungsbefehle). Dies entsteht mehr oder weniger offensichtlich durch das, was man in Hochsprachen »Typkonvertierung« nennt: So kann aus einem Byte ein Word entstehen oder aus einem Word ein DoubleWord. Was hat also in der binären Darstellung zu erfolgen, wenn das Byte »2« zum Word »2« ausgedehnt werden soll? Das scheint einfach: die binäre Darstellung der »2« im Byteformat ist 00000010_b, die im Wordformat $00000000_00000010_b$. Man muss also lediglich die neu hinzugekommenen Bits auf »0« setzen. Was aber bei der Zahl »-2«? Auch hier ein Vergleich der binären Darstellungen: 11111110_b und $11111111_11111110_b$. Im Falle negativer Zahlen werden also die neu hinzugekommenen Bits alle auf »1« gesetzt. Allgemein: Zur »Erweiterung« der Darstellung einer Zahl werden die neu hinzugekommenen Bits auf den Wert gesetzt, den das Vorzeichenbit der originalen Zahl besitzt. Dieses Verfahren nennt man »Vorzeichenerweiterung« (*sign extension*), da das Vorzeichen (formal) auf alle links von ihm anzusiedelnden Bits erweitert wird.

vorzeichenlose Zahlen

Besitzen Integers kein Vorzeichen, so wird das MSB ebenfalls zur Codierung verwendet. Verglichen mit einer vorzeichenbehafteten Zahl hat somit eine vorzeichenlose zwar keinen doppelt so großen Wertebereich, aber die maximal darstellbar Zahl ist doppelt so hoch: Da in der obigen Darstellung alle 32 Bits der Integer zur Verfügung stehen, kann mit einem DoubleWord ein maximaler Wert von 2^{32}-1 = 4,294,967,295 dargestellt werden. Der minimale Wert ist jedoch »nur« 0.

Ganz analog sind natürlich auch die anderen Integers definiert: Bytes, Words und QuadWords als vorzeichenlose Integer und ShortInts, SmallInts und QuadInts als vorzeichenbehaftete Pendants mit 8, 16 und 64 Bits und den entsprechenden Wertebereichen.

Indefinite

Auch bei Integers kann, wie bei den Realzahlen, natürlich eventuell ein Anlass bestehen, darstellen zu müssen, dass die resultierende »Zahl« nicht existiert oder undefiniert ist. Solch eine indefinite Integer wird dargestellt, indem man das Vorzeichenbit setzt und alle anderen Bits löscht. Damit ist aber dieser Code für eine Indefinite identisch mit dem größten negativen Wert, der dargestellt werden kann. Und das wiederum bedeutet, dass es keine »echte« und eindeutige Darstellung einer Indefiniten bei den Integers gibt: Ob $10000000_00000000_00000000_00000000_b$ nun -2,147,483,647 bedeutet oder »undefiniert«, liegt allein im Ermessen und der Interpretation des Programmierers.

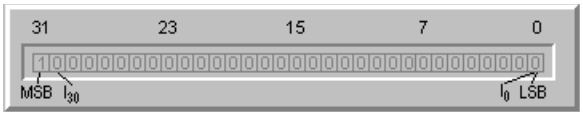


Abbildung 5.13: Speicherabbild einer »Indefiniten Integer« am Beispiel einer LongInt

BCDs

An dieser Stelle könnte man die Besprechung der Codierung von Integers als erledigt betrachten, gäbe es nicht noch eine Gruppe von Zahlen, die man als *binary coded decimals*, also als binär codierte Dezimalzahlen oder kurz »BCDs« bezeichnet. BCDs sind nach Intels Definition »vorzeichenlose 4-Bit-Integers mit Werten zwischen 0 und 9«! Und das bedarf ein wenig der Erläuterung.

Sie erinnern sich an die Problematik der Umrechnung von dezimaler in binärer Darstellung, die im Kapitel »Binäre Zahlendarstellung und Hexadezimalsystem« weiter oben auf Seite 782 angesprochen wurde. Die-

ses Problem der Hin- und Herkonvertierung mit Suche nach der jeweils höchsten Potenz von 2 bzw. 16 hat einen schlauen Kopf zu einer grandiosen Idee gebracht: Warum kann man Dezimalzahlen nicht mit hexadezimalen Ziffern darstellen? Also: $100_d = 1_d0_d0_d \rightarrow 1_h0_h0_h = \$100$! Denn die Dezimalziffern sind ja Teilmenge der Hexadezimalziffern, also in ihnen enthalten. Auf diese Weise würde die lästige Konvertierung entfallen! Dies war die Geburtsstunde der BCDs.

BCDs sehen aus wie Dezimalzahlen (weil keine der »fremden« Hexadezimalziffern A bis F zum Einsatz kommen kann), sind aber in Wirklichkeit Hexadezimalzahlen und somit binär codiert! Dies ist auch der Grund, warum in der Herleitung eben zwischen $1_d0_d0_d$ und $1_h0_h0_h$ ein Pfeil und kein Gleichheitszeichen steht. Denn wertmäßig liegen zwischen beiden Zahlen Welten (100 gegenüber 256)!

Und doch: Warum tut man so etwas? Bringen BCDs tatsächlich Vorteile? Denn bei etwas längerem Nachdenken findet man eigentlich nur Nachteile bei ihrer Verwendung:

- Reinste Verschwendung von Ressourcen! Da nur zehn von 16 möglichen Ziffern für jede Stelle der Zahl verwendet werden, vergeudet man mehr als ein Drittel der Kapazitäten pro Ziffer. Dies addiert sich schnell zu recht großen Werten: 100_d binär codiert ($100) hat den Wert 256 und damit eine »Ausnutzung« von 39%, bei $10{,}000_d$ ($1 0000) sind es nur noch 15% und bei $1{,}000{,}000_d$ ($100 0000) lächerliche 6%.
- Neue Arithmetik. Wenn man mit BCDs rechnen will, so muss man neue Befehle implementieren, die dem Sachverhalt Rechnung tragen. Denn z.B. bei der Addition von 1_h zu 9_h hat ja nun als Ergebnis 1_h0_h zu folgen. Also: $9_d + 1_d \rightarrow 16_d$!
- Reduzierte Performance. Nicht nur die »neuen« Rechenbefehle und eventuell noch andere notwendige »Korrekturen« sorgen dafür, dass das Arbeiten mit BCDs sicherlich nicht mit der gleichen Geschwindigkeit erfolgen kann wie das mit »normalen« Zahlen.

gepackte BCDs

Was das Problem Verschwendung betrifft: Im Beispiel oben wurde von dem ausgegangen, was Intel »gepackte« BCDs nennt. Das sind zwei dezimale Ziffern pro Byte. Denn auch das »unteilbare« Byte besteht ja aus zwei hexadezimalen Ziffern, häufig »nibbles« genannt. Und diese hexadezimalen Ziffern können ja jeder eine binär codierte Dezimalziffer aufnehmen: $12 eben die Ziffern »1« und »2«. Da zwei dieser Ziffern in einem Byte stecken, heißt die BCD 12 auch *gepackt*. Und wer gepackt sagt,

muss auch »entpackt« sagen können. Konsequenterweise heißt das, dass nur noch eine Ziffer pro Byte verwendet wird. Das ist auch so und daraus resultiert eine noch größere Verschwendung von Ressourcen. Denn nun können nur noch 10 Zahlen pro Byte binär codiert werden: die Zahlen »0« bis »9«. Und das macht 10/256 = 4% pro Stelle aus. Auf unsere Zahlen von oben angewendet heißt das: 100_d binär codiert benötigt 3 Byte mit einem Maximalwert von 999, was zu lächerlichen 0.006% Ausnutzung führt. Was soll das?

Und in der Tat: BCDs führen ein Schattendasein! Ich persönlich kenne sie nur noch aus der Zeit der ersten Steinzeitrechner mit 4 MHz Taktfrequenz, als es einen ziemlichen (zeitlichen) Aufwand bedeutet hat, Hexadezimalzahlen zum Zwecke der Darstellung in Dezimalzahlen umzuwandeln. Für »einfache« Aufgaben wie Taschenrechner mit begrenztem Wertebereich war das Rechnen und Anzeigen mit und von BCDs erheblich einfacher und »effizienter« als die Realisierung von Hin- und Rückrechenroutinen zwischen Dezimalen und Hexadezimalen. So ist auch hier der Grund für die ungepackten BCDs zu suchen; denn durch »einfache« OR-Verknüpfung einer ungepackten (d.h. einstelligen) BCD mit dem Wert $30 ergeben sich die ASCII-Werte $30 bis $39. Und genau diese Werte sind die ASCII-Werte der Zeichen »0« bis »9« ...

Aus solchen Gründen gibt es speziell für BCDs geschaffene Befehle. Sie lassen sich aufteilen in zwei Gruppen:

- Befehle, die BCDs in FPU-Registern betreffen; dies sind glücklicherweise nur zwei: Das Laden einer BCD (genauer gesagt: einer gepackten BCD – dazu später mehr) in ein FPU-Register und Speichern einer BCD aus einem FPU-Register. Diese beiden Befehle überführen eine BCD in eine »normale« ExtendedReal bzw. umgekehrt, sodass in der FPU mit den ganz normalen Rechenbefehlen gearbeitet werden kann.
- Befehle, die BCDs in Allzweckregistern betreffen; da die Lade- und Speicherbefehle in diesem Fall nicht, wie bei der FPU, eine Konvertierung übernehmen, müssen tatsächlich eigens geschaffene Rechenbefehle herhalten, zumindest aber Befehle, die bei der Interpretation von Rechenoperationen mit BCDs eingesetzt werden können.

Diese Sonderfälle arithmetischer Operationen wurden auf den Seiten 46 (CPU-BCDs) bzw. 245 (FPU-BCDs) behandelt.

OctelWord und OctelInt

DoubleQuadWord (= OctelWord) und DoubleQuadInt (= OctelInt) sind die größten vorzeichenlosen bzw. vorzeichenbehafteten Integers, mit denen ein Prozessor (zur Zeit!) arbeiten kann. Sie besitzen 128 binäre Stellen.

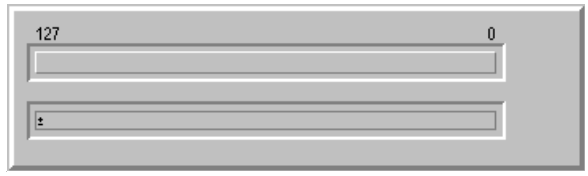


Abbildung 5.14: Speicherabbild eines vorzeichenlosen DoubleQuadWord (oben) und einer vorzeichenbehafteten DoubleQuadInt (unten)

Aufgrund der Größe beider Zahlen sind sie nur in XMM-Registern unter SSE2-Erweiterung nutzbar. Ihr Wertebereich und die Genauigkeit sind damit:

	binär		dezimal	
	max	**min**	**max**	**min**
DoubleQuadWord				
Wertebereich:	2^{128}-1	0	$\cong 3.402823669 \cdot 10^{38}$	0
Genauigkeit:	128 Stellen		38 Stellen	
DoubleQuadInt				
Wertebereich:	$+2^{127}$-1	-2^{127}-1	$\cong +1.701411835 \cdot 10^{38}$	$\cong -1.701411835 \cdot 10^{38}$
Genauigkeit:	127 Stellen		38 Stellen	

Tabelle 5.7: Charakteristika eines DoubleQuadWords und einer DoubleQuadInt

QuadWords und DoubleQuadInts sind etwas Besonderes! Sie sind weder »echte« Integers in dem Sinne, dass sie einen den ScalarReals vergleichbaren »Sonderstatus« als isolierte Teile einer gepackten Struktur sind, noch sind sie Teile einer gepackten Datenstruktur selbst. Das heißt, dass sie nicht mit den üblichen arithmetischen Befehlen unter SSE/SSE2 manipuliert werden können: Ein Addieren, Subtrahieren oder Multiplizieren ist genauso wenig möglich wie ein Vergleich zweier QuadWords oder die Konvertierung in Realzahlen und zurück.

Vielmehr muss man sie zum einen als Platzhalter für die XMM-Datenstruktur auffassen, zum anderen als »Bitfeld«. So wird ein PackedDoubleWord oder ein PackedByte nicht mit einem jeweils speziellen Lade-/Speicherbefehl in/aus ein/einem XMM-Register geschrieben, sondern mittels MOVQ, *move quadword*. Somit ist das QuadWord ein »abstrak-

ter« Begriff, ein Wort, das Ähnliches ausdrücken will (»Datum mit 16 Bytes Umfang«) wie die in diesem Buch benutzte Datenstruktur PackedInteger und daher in dieser Bedeutung ein Synonym zu diesem Begriff ist. Andererseits aber umfasst der SSE2-Befehlssatz Instruktionen, die nur mit QuadWords, nicht aber mit den skalaren oder gepackten Elementardaten durchgeführt werden können. Bei diesen Befehlen handelt es sich um die logischen Befehle wie AND oder OR. Andere bitorientierte Befehle (die Shift-Befehle) lassen sich wahlweise mit den gepackten Daten und den QuadWords durchführen.

QuadWord und QuadInt

Für die QuadWords/QuadInts in den MMX-Registern gilt grundsätzlich das Gleiche wie für die DoubleQuadWords/DoubleQuadInts in den XMM-Registern: Sie dienen einerseits als »abstrakter« Platzhalter für eine 8-Byte-Datenstruktur im Sinne von »ShortPackedInteger« vor allem für die Lade-/Speicherbefehle, zum anderen sind auch hier spezielle Instruktionen nur (die bitorientierte logischen Instruktionen) oder auch (die bitorientierten Shift-Befehle) mit diesen Datentypen möglich.



Abbildung 5.15: Speicherabbild eines vorzeichenlosen DoubleQuadWord (oben) und einer vorzeichenbehafteten QuadInt (unten)

Aufgrund der Größe beider Zahlen sind sie nur in MMX-Registern unter MMX-Erweiterung nutzbar. Ihr Wertebereich und die Genauigkeit sind damit:

	binär		dezimal	
	max	**min**	**max**	**min**
QuadWord				
Wertebereich:	2^{64}-1	0	$\cong 1.844674407 \cdot 10^{19}$	0
Genauigkeit:	64 Stellen		19 Stellen	
QuadInt				
Wertebereich:	$+2^{63}$-1	-2^{63}-1	$\cong +9.223372037 \cdot 10^{18}$	$\cong -9.223372037 \cdot 10^{18}$
Genauigkeit:	63 Stellen		18 Stellen	

Tabelle 5.8: Charakteristika eines QuadWords und einer QuadInt

Eine gewisse Bedeutung haben QuadWords und LongInts noch in der Hinsicht, dass sie mit 64 Bit oder 8 Byte Umfang genauso breit sind wie die DoubleReals der FPU und damit deren Integer-Pendant sind.

DoubleWord und LongInt

DoubleWords und LongInts haben jeweils eine Größe von 4 Byte. Sie sind damit die »klassischen« Standarddaten, um die sich bei den heutigen Prozessoren das meiste dreht. So haben sie genau die Größe der Allzweckregister der CPU seit dem 80386 und sind auch unter dem Stichwort »alignment« die Datenstrukturen, die eine besondere Bedeutung beim Datenaustausch zwischen Prozessor und Peripherie besitzen.

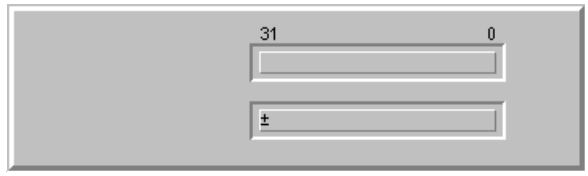


Abbildung 5.16: Speicherabbild eines vorzeichenlosen DoubleWord (oben) und einer vorzeichenbehafteten LongInt (unten)

	binär		dezimal	
	max	**min**	**max**	**min**
QuadWord				
Wertebereich:	2^{32}-1	0	= 4,294,967,295	0
Genauigkeit:	32 Stellen		9 Stellen	
LongInt				
Wertebereich:	$+2^{31}$-1	-2^{31}-1	= +2,147,483,647	= -2,147,483,647
Genauigkeit:	31 Stellen		9 Stellen	

Tabelle 5.9: Charakteristika eines DoubleWords und einer LongInt

DoubleWords/LongInts haben eine Breite von 32 Bit bzw. 4 Byte. Sie sind damit die Integer-Pendants zu den SingleReals der FPU.

Word und SmallInt

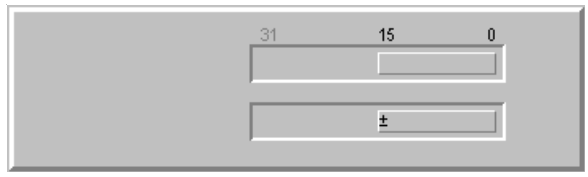


Abbildung 5.17: Speicherabbild eines vorzeichenlosen Word (oben) und einer vorzeichenbehafteten SmallInt (unten)

Hierbei handelt es sich um »Standard-Daten«, wie sie seit dem 8086 als Standard gelten.

	binär		dezimal	
	max	**min**	**max**	**min**
Word				
Wertebereich:	2^{16}-1	0	= 65,535	0
Genauigkeit:	16 Stellen		4 Stellen	
SmallInt				
Wertebereich:	**$+2^{15}$-1**	**-2^{15}-1**	**= +32,767**	**= -32,767**
Genauigkeit:	15 Stellen		4 Stellen	

Tabelle 5.10: Charakteristika eines Words und einer SmallInt

Byte und ShortInt

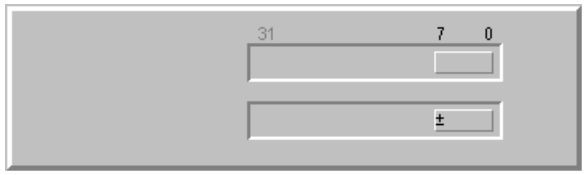


Abbildung 5.18: Speicherabbild eines vorzeichenlosen Byte (oben) und einer vorzeichenbehafteten ShortInt (unten)

Hierbei handelt es sich um »Standard-Daten«, wie sie seit dem 8086 als Standard gelten.

	binär		dezimal	
	max	**min**	**max**	**min**
Byte				
Wertebereich:	2^{8}-1	0	= 256	0
Genauigkeit:	8 Stellen		2 Stellen	
ShortInt				
Wertebereich:	$+2^{7}$-1	-2^{7}-1	= +127	= -127
Genauigkeit:	7 Stellen		2 Stellen	

Tabelle 5.11: Charakteristika eines Bytes und einer ShortInt

Binary Coded Decimals

BCDs können an zwei unterschiedlichen Stellen bearbeitet werden: Durch die CPU in Allzweckregistern und durch die FPU in den FPU-Registern. Dementsprechend werden im Folgenden sog. CPU-BCDs und FPU-BCDs unterschieden.

CPU-BCDs

CPU-BCDs, besser: durch die CPU bearbeitete BCDs, können wiederum eingeteilt werden in die »ungepackten« und die »gepackten« BCDs. Zu den Unterschieden vgl. Seite 804.



Abbildung 5.19: Speicherabbild einer ungepackten CPU-BCD

Definitionsgemäß umfasst eine ungepackte BCD vier Bits, sodass mit der Einschränkung, nur dezimale Ziffern darzustellen, der Wertebereich der BCD sehr überschaubar ist: Es sind die Ziffern »0« bis »9«.

Da die ungepackten BCDs nur vier der acht Bits eines Bytes belegen, kann durch die Nutzung der höheren vier Bits durch eine weitere ungepackte BCD das Byte vollständig genutzt werden. Eine BCD, die zwei BCD-Ziffern enthält, nennt man »gepackt«. Gepackte BCDs haben einen Wertebereich von 0 bis 99. CPU-BCDs sind definitionsgemäß vorzeichenlos.



Abbildung 5.20: Speicherabbild einer gepackten CPU-BCD

FPU-BCDs Die FPU-BCDs unterscheiden sich in mehrerlei Hinsicht von den CPU-BCDs. So besitzen die FPU-BCDs ein Vorzeichen. Ferner sind FPU-BCDs, wie DoubleReals oder SingleReals auch, nur im Speicher als Datenstruktur existent, weshalb der Begriff »FPU-BCD« etwas unscharf ist: Innerhalb der FPU und ihren Registern sind nur ExtendedReals erlaubt. Das bedeutet, dass die FPU-BCDs durch die entsprechenden Ladebefehle beim Laden aus dem Speicher (FBLD) in das ExtendedReal-Format konvertiert werden und umgekehrt beim Ablegen in den Speicher (FBSTP) aus diesem Format erzeugt werden.

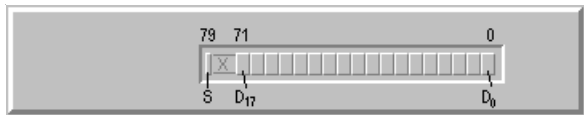


Abbildung 5.21: Speicherabbild einer FPU-BCD

Die FPU-BCDs sind, wie die gepackten CPU-BCDs auch, »gepackt« und als aus den BCD-codierten Ziffern zusammengesetzte Zahlen zu interpretieren. Das heißt Byte 0 des FPU-Registers enthält die BCD-Ziffern 1 und 0 der binär codierten Zahl, Byte 8 die Ziffern 17 und 16. Das

MSB (Byte #9) enthält lediglich das Vorzeichen an Bitposition 79, dem *most significant bit* dieses Bytes. Alle anderen sieben Bits sind so genannte »*don't care bits*«, also Bits, bei denen es keine Rolle spielt, ob sie gesetzt oder gelöscht sind, da sie in den Instruktionen nicht berücksichtigt werden. Eine FPU-BCD besteht somit aus Vorzeichen und 18 binär codierten Dezimalziffern (D_0 bis D_{17}).

	binär		dezimal	
	max	min	max	min
CPU-BCD				
Wertebereich:	-	-	9	-9
Genauigkeit:	-		1 Stelle	
Packed CPU-BCD				
Wertebereich:	-	-	99	-99
Genauigkeit:	-		2 Stellen	
FPU-BCD				
Wertebereich:	-	-	999.999.999.999.999.999	-999.999.999.999.999.999
Genauigkeit:	-		18 Stellen	

Tabelle 5.12: Charakteristika verschiedener BCDs

Da BCDs durchaus im Rahmen der FPU zu sehen sind und bei Exceptions der FPU mit Realzahlen Indefinite eine besondere Rolle spielen (als Signal, falls die betreffenden Exceptions maskiert sind), gibt es auch eine BCD-Indefinite:



Abbildung 5.22: Speicherabbild einer »Indefiniten BCD«

5.2.4 Gepackte Daten

Als »gepackte« Daten werden Datenstrukturen bezeichnet, die sich aus mehreren identischen Elementardaten zusammensetzen und Ziel der Instruktionen der SIMD-Befehlssätze sind. Mit diesen Datenstrukturen können die gleichen Operationen auf die Mitglieder dieser Strukturen gleichzeitig ausgeführt werden (*single instruction multiple data*).

Dieses Buch unterscheidet vier gepackte Daten: die ShortPackedReals und ShortPackedIntegers der MMX-Register sowie die PackedReals

und die PackedIntegers der XMM-Register. Zur Definition vgl. Seite 844, Tabelle 5.23, bzw. Tabelle 5.27 auf Seite 850.

ShortPacked-Integers

Aufgrund der Größe der MMX-Register mit 64 Bit können auch nur diese 64 Bit für die Datenstrukturen verwendet werden. Damit lassen sich je nach dem zugrunde liegenden Elementardatum folgende gepackten Daten darstellen:

ShortPacked-Byte
ShortPacked-ShortInt



Abbildung 5.23: Speicherabbild eines ShortPackedByte (oben) und einer ShortPackedShortInt

ShortPacked-Word
ShortPacked-SmallInt



Abbildung 5.24: Speicherabbild eines ShortPackedWord (oben) und einer ShortPackedSmallInt

ShortPacked-DoubleWord
ShortPacked-LongInt



Abbildung 5.25: Speicherabbild eines ShortPackedDoubleWord (oben) und einer ShortPackedLongInt

Die Bits 64 bis 79 der für MMX zweckentfremdeten FPU-Register gelten als reserviert und sind alle gesetzt.

ShortPacked-Reals

Da auch die ShortPackedReals die MMX-Register benutzen, stehen ihnen nur insgesamt 64 Bit zu Verfügung, die gerade ausreichen, zwei SingleReals in einer Datenstruktur zusammenzufassen. Auch in diesem Fall sind die Bits 64 bis 79 der für MMX/3DNow! zweckentfremdeten FPU-Register reserviert und alle gesetzt.

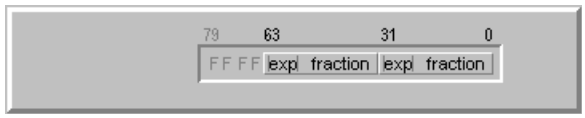


Abbildung 5.26: Speicherabbild einer ShortPackedSingle

ShortPackedReals sind Datenstrukturen, die unter 3DNow! bzw. 3DNow!-X zum Einsatz kommen. Sie benutzen damit die erweiterten Befehlssätze der AMD-Prozessoren und sind bei Intel-basierten Systemen nicht einsetzbar.

PackedIntegers

PackedIntegers sind die »großen Geschwister« der ShortPackedIntegers. Sie sind in den XMM-Registern der Intel-Prozessoren beheimatet und können daher (zurzeit?) auch nur durch Intel-Prozessoren bearbeitet werden.

PackedByte
PackedShortInt

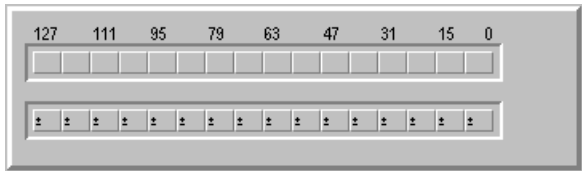


Abbildung 5.27: Speicherabbild eines PackedByte (oben) und einer PackedShortInt

PackedWord
PackedSmallInt

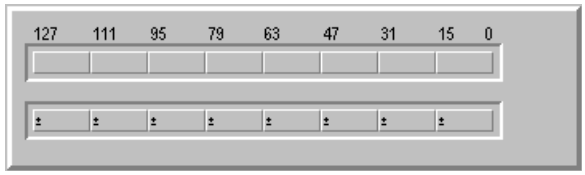


Abbildung 5.28: Speicherabbild eines PackedWord (oben) und einer PackedSmallInt

PackedDouble-Word
PackedLongInt

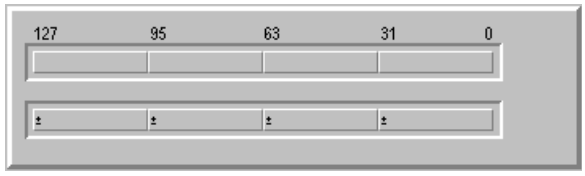


Abbildung 5.29: Speicherabbild eines PackedDoubleWord (oben) und einer PackedLongInt

PackedQuad-Word PackedQuadInt

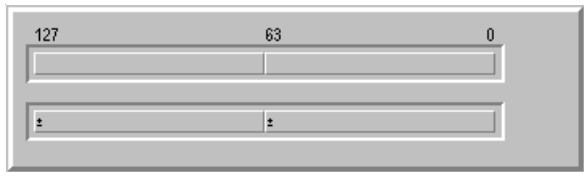


Abbildung 5.30: Speicherabbild eines PackedQuadWord (oben) und einer PackedQuadInt

Da die XMM-Register speziell für diese Datenstrukturen geschaffen wurden, können sie – anders als die FPU-Register unter MMX – auch vollständig mit den Datenstrukturen gefüllt werden – reservierte Bits gibt es nicht. Die MMX-Register sind 128 Bits breit, sodass sie doppelt so viele Elementardaten in den Strukturen aufnehmen können wie die MMX-Register. Und durch diese Breite ist auch eine weitere gepackte Datenstruktur möglich geworden: die PackedQuadInt.

PackedReals

Auch die XMM-Register können Realzahlen aufnehmen. Aufgrund der im Vergleich zu den MMX-Registern doppelten nutzbaren Breite können das zwei Datenstrukturen sein:

PackedSingle

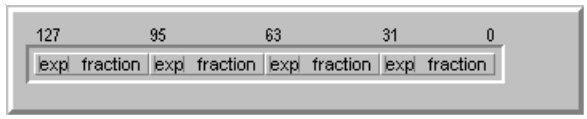


Abbildung 5.31: Speicherabbild einer PackedSingle

PackedDouble

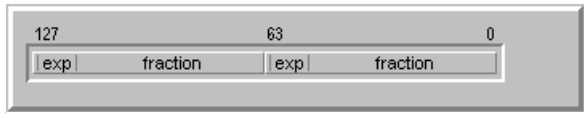


Abbildung 5.32: Speicherabbild einer PackedDouble

Da XMM-Register (zurzeit?) nur in Intel-Prozessoren realisiert werden, können PackedReals genauso wie PackedIntegers nur auf Intel-basierten Systemen genutzt werden.

5.2.5 Erweiterte Elementardaten

So genannte »skalare« Daten gibt es nur bei den Intel-Instruktionen. Intel versteht unter diesen Daten die Elementardaten einer Datenstruktur, die an der niedrigstwertigen Stelle der Struktur stehen. Sie können somit als »Erweiterung« der Elementardaten aufgefasst werden.

Skalare Daten erweitern die Möglichkeiten der MMX-Register um die Fähigkeit, mit einzelnen Elementardaten und nicht mit den Datenstrukturen im Ganzen arbeiten zu können. Dennoch machen sie die FPU nicht überflüssig: Die Genauigkeit, der Wertebereich und die arithmetischen und logischen Möglichkeiten der FPU sind deutlich größer. Skalare Daten sollten daher lediglich verstanden werden als Möglichkeit, einzelne Daten einer Datenstruktur gezielt ansprechen und modifizieren zu können.



Es gibt derzeit nur skalare Realzahlen! Für Integers sind keine skalaren Berechnungen vorgesehen! Weder ist die bei Einführung der MMX-Register definierte QuadInt bzw. das QuadWord ein skalares Datum (da sie das gesamte Register ausfüllen und daher kein skalarer Teil einer Struktur sein können), noch führt die Erweiterung der MMX-Befehle auf XMM-Register zur Bearbeitung von skalaren Daten! Denn z.B. lädt MOVQ entweder das Elementardatum QuadWord in ein MMX-Register oder ein gepacktes, nicht aber skalares (!) QuadWord in ein XMM-Register.

ScalarSingle

Als ScalarSingle bezeichnet man die niedrigstwertige SingleReal einer PackedSingle-Struktur:

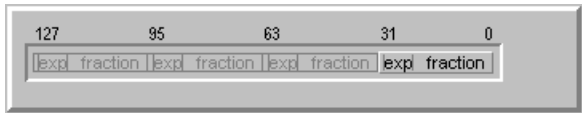


Abbildung 5.33: Speicherabbild einer ScalarSingle

ScalarDouble

Analog ist eine ScalarDouble die niedrigstwertige DoubleReal einer PackedDouble-Struktur:

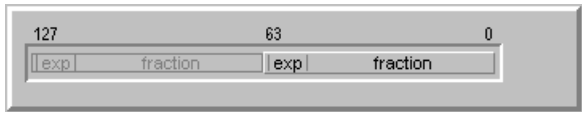


Abbildung 5.34: Speicherabbild einer ScalarDouble

5.2.6 Gegenüberstellung der verschiedenen Datenbezeichnungen

Buch	Assembler	Delphi	C++
vorzeichenlose Integer:			
Byte	BYTE	Byte	unsigned char
Word	WORD	Word	unsigned short
DoubleWord	DWORD	LongWord	unsigned long
vorzeichenbehaftete Integer:			
ShortInt	SBYTE	ShortInt	(signed) char
SmallInt	SWORD	SmallInt	(signed) short
LongInt	SDWORD	LongInt	(signed) long
-	-	Int64 [2])	-
Fließkommazahlen:			
SingleReal	REAL4	Single	float
DoubleReal	REAL8	Double	double
ExtendedReal	REAL10	Extended	long double
MMX- und XMM-Daten			
ScalarSingle	REAL4	Single	float
ScalarDouble	REAL8	Double	double
QuadWord	QWORD	-	-
OctelWord	OWORD	-	-
QuadInt	QWORD	-	-
OctelInt	OWORD	-	-
Andere Daten			
	TBYTE	-	-
-	FWORD	-	-
	PWORD	-	-

Tabelle 5.13: Gegenüberstellung der in diesem Buch und den in Assembler, Delphi und C++ benutzten Bezeichnungen für Daten

5.3 Speicheradressierung

Wenn ein Befehl Datenquellen benötigt, so gibt es prinzipiell zwei Möglichkeiten, worum es sich bei eingesetzten Operanden handeln könnte:

- Ein Register. Hierbei ist unerheblich, ob es sich um die Allzweckregister der CPU, Status- oder Kontrollregister, die Rechenregister der FPU oder die XMM-Register handelt oder um modellspezifische Register. In jedem Fall sind diese Register auf dem Chip angesiedelt, der mit ihnen umzugehen hat. Diese Operanden nennt man häufig auch Registeroperanden.

- Eine Stelle im Speicher, an der das Datum steht, das für die Operation benötigt wird oder an das ein Datum geschrieben werden muss. Diese Operanden nennt man Speicheroperanden.

Der Zugriff auf Registeroperanden ist trivial: Jeder Befehl, der auf ein Register zugreift, akzeptiert Mnemonics, die solche Register bezeichnen: EAX, BP, ST(3), MM7, XMM4, CR0. Die Adressierung solcher Registeroperanden ist somit kein Problem: Sie werden einfach durch das sie beschreibende »reservierte« Symbol benannt.

Bei Speicherstellen ist das ein wenig anders. Eine Speicherstelle hat kein Mnemonic, das zur Angabe verwendet werden könnte. Sie kann lediglich durch ein einziges, unverwechselbares Merkmal identifiziert werden: ihre einzigartige Adresse im virtuellen Speicherraum (vgl. Kapitel »Speicherverwaltung« auf Seite 394). Daher sind bei Bezügen auf Speicheroperanden *immer* Adressen im Spiel: Der Befehl bekommt als Operanden die Adresse *des ersten Bytes* im Speicher übermittelt, die das Datum aufnimmt.



Das ist wichtig! Da die kleinste Einheit, aus der der Speicher aufgebaut ist, ein Byte ist, beziehen sich alle Adressen auf Bytes, selbst wenn z.B. eine gepackte Struktur aus zwei DoubleReals (16 Bytes) »unter der Adresse abgelegt« ist. In Wahrheit ist unter dieser Adresse das erste Byte einer Sequenz aus 16 abgelegt. Dieses Byte stellt das LSB, das *least significant byte* oder niedrigstwertige Byte des Datums dar, dem 15 weitere folgen, deren Adressen in der Regel nirgendwo abgelegt wurden. Analog zeigt ein Label tatsächlich nicht auf einen Befehl, sondern auf das erste Byte in einer Sequenz von bis zu 15 Bytes, die die Codesequenz für den Befehl darstellen. Das kann durchaus ein Präfix-Byte sein, das nur indirekt etwas mit dem Opcode zu tun hat, vor dem es steht.



Das bedeutet, dass die bloße Angabe einer Adresse noch nichts darüber aussagt, welcher Datentyp sich dahinter verbirgt und wie viele weitere Bytes mit ihr in Verbindung zu bringen sind. Der Assembler benötigt daher noch zusätzliche Informationen, um die Codesequenz für den Befehl korrekt zu erzeugen. Wenn nämlich aus einer bestimmten Adresse ein Word »ausgelesen« werden soll, heißt das, dass der Assembler den Opcode verwenden muss, der genau das tut. So gibt es für die meisten Befehle, die mit Datentransfer aus oder zum Speicher zu tun haben, eigene Opcodes für Byte- oder Multi-Byte-Daten. So z.B. für den Befehl MOV die Opcodes $8A und $8B. Verwendet man den Opcode

$8A mit der Adresse $12345678, so liest die CPU aus der Adresse $12345678 genau ein Byte aus. Mit dem Opcode $8B würde sie in 32-Bit-Umgebungen (vgl. »Adress- und Operandengrößen« auf Seite 765) das an der Adresse stehende Byte sowie drei weitere an den folgenden (Byte-)Adressen stehende (insgesamt somit ein DoubleWord) auslesen. Sollen es nur zwei Bytes sein, da »ein Word angesprochen wird«, müsste dem Opcode $8B in 32-Bit-Umgebungen der operand size prefix $66 vorangehen, um zu signalisieren, dass nicht die Standardgröße für den Operanden Verwendung finden soll.

Direkte Adressierung

Kennt man also die Adresse eines Datums – und der Assembler kennt sie immer dann, wenn sie irgendwo deklariert wurde! –, so kann man sie als Konstante in der Codesequenz codieren: *Opcode – Adresse*. Diese Art der Adressierung nennt man *Direkte Adressierung*, sie zeichnet sich dadurch aus, dass die Adresse bereits zur Zeit der Assemblierung direkt angegeben und in eine Codesequenz einbaut werden kann.

Dass diese Adressen nicht als Hexadezimalzahlenungetüm auftreten, sondern benutzerfreundlich im Kleide eines Symbols (Variable), heißt allerdings nicht, dass sie nicht genau das sind: hexadezimale Zeiger auf ein Datum, das mittels direkter Adressierung angegeben werden kann. Die formale Verwendung im Rahmen einer direkten Adressierung ist somit

```
MOV   EAX, [DWordVar]
```

und liest sich: »Kopiere das Datum, auf das Symbol *DWordVar* zeigt, in Register EAX«. Warum das wichtig ist? Wir werden es in ein paar Absätzen sehen!

Indirekte Adressierung

Manchmal jedoch ist die Adresse abhängig von verschiedenen Randbedingungen, die erst zur Laufzeit eines Programms bekannt sind. Nehmen Sie z.B. ein einfaches Verschlüsselungsprogramm, das einen String anhand einer Verschlüsselungstabelle verschlüsselt. Zugegeben: Heute sind Verschlüsselungsprogramme wesentlich aufwändiger mit symmetrischer oder asymmetrischer Verschlüsselung und 128-Bit-Schlüsseln, als es in diesem Beispiel der Fall ist. Hier soll für jedes mögliche Byte ein »verschlüsseltes« Byte in einer Tabelle existieren, das durch einfaches Tabellenauslesen gewonnen wird (vgl. den Befehl XLAT auf Seite 91). Doch das Beispiel soll ja nicht die Realität widerspiegeln, sondern einen Sachverhalt veranschaulichen. Und dazu taugt es allemal!

In diesem Fall muss also die Adresse berechnet werden. Da sie zum Zeitpunkt der Assemblierung nicht feststehen kann, kann sie auch

nicht über direkte Adressierung angegeben werden. Diese Art der Adressenberechnung nennt man *Indirekte Adressierung*. Sie zeichnet sich dadurch aus, dass die Adresse aus einem Register ausgelesen wird.

Hierzu gibt es mehrere Möglichkeiten. Einfachste Methode: Die Adresse steht von Geisterhand (oder ganz normal vom Programmierer an irgendeiner Stelle im Programm) dort hingezaubert in einem Register:

```
MOV EAX, [EBX]
```

Dann interpretiert die CPU den Wert, der in EBX steht, als Adresse, an der das Datum steht.



Bitte beachten Sie die eckigen Klammern! Sie signalisieren dem Assembler, dass indirekt adressiert werden soll. Analog der direkten Adressierung liest sich dies: »Kopiere das Datum, auf das der Zeiger in Register EBX zeigt, in Register EAX«. Ohne die eckigen Klammern hieße es: »Kopiere das Datum in Register EBX in Register EAX«.



Theoretisch könnte auch eine Speicherstelle dazu verwendet werden, eine Adresse aufzunehmen, mit der eine indirekte Adressierung erfolgt. Dies ist jedoch von Intel nicht vorgesehen worden! Das bedeutet, dass bei rein indirekter Adressierung nur Register zu Einsatz kommen können. Soll ein Zeiger aus einer Speicherstelle zur indirekten Adressierung verwendet werden, muss er vorher in ein Register kopiert werden:

```
MOV EBX, [PointerVar]
MOV EAX, [EBX]
```

Dies ist z.B. immer dann nötig, wenn Parameter via »call by reference« übergeben werden (»Variable«).

Kommen wir zurück zu unserem Demonstrationsbeispiel »Verschlüsselung«. Wie bereits gesagt, hängt die Adresse des auszutauschenden Datums von einem Zeiger in eine Tabelle und damit vom Input zur Laufzeit ab. Die Tabelle selbst aber ist bereits zur Assemblierungszeit bekannt: Der Programmierer muss sie ja vorgeben. Das bedeutet, die Adresse besteht bei genauerer Betrachtung aus einem konstanten und einem variablen Teil. Dementsprechend kann die Adresse eines beliebigen Datums in einer Tabelle wie folgt berechnet werden:

```
Adresse = Offset + [Index]
```

wobei *Offset* die Adresse der Tabelle selbst ist und somit zur Zeit der Assemblierung feststeht, also als Konstante angegeben werden kann,

und *Index* der variable Teil, der zur Laufzeit berechnet werden muss und den Zeiger in die Tabelle (sozusagen einen Offset auf den Offset) darstellt. Er wird zur Laufzeit einem Register entnommen und zu der Konstanten addiert. Dies würde unter Assembler wie folgt programmiert werden:

```
MOV     EAX, [Table + Index]
```

Gehen wir einen Schritt weiter: Zur Codierung und Decodierung existieren zwei Tabellen. Deren Lage ist zwar zur Assemblierungszeit bekannt, nicht aber, welche der beiden Tabellen verwendet werden soll, je nachdem, ob codiert oder decodiert wird. Dies könnte erfolgen, indem auch der Offset, also die Adresse der zum Einsatz kommenden Tabelle, indirekt angegeben und erst zur Laufzeit berechnet wird:

```
MOV EAX, [Basis + Index]
```

Der Assembler erzeugt daraus Code, der zur Laufzeit die Adresse wie folgt berechnet:

```
Adresse = [Basis] + [Index]
```

In diesem Fall musste irgendwann vor der Verwendung an die durch *[Basis]* spezifizierte Stelle der Offset der korrekten Tabelle geschrieben werden.



Bitte beachten Sie, dass beide Angaben in Registern stehen müssen! Wie bereits weiter oben gesagt, stellen Angaben, die durch eine Variable gemacht werden, Konstanten dar, die zur Assemblierungszeit bereits bekannt sind (sein müssen) und daher einen direkten Anteil darstellen, der hier nicht vorkommt!

Nehmen wir schließlich an, beide Tabellen lägen unmittelbar hintereinander und könnten durch einen einfachen Abstand, die jeweilige Tabellengröße, voneinander angegeben werden. Dann ließe sich dies wiederum in Form einer Mischung aus direkter und indirekter Adressierung angeben:

```
MOV     EAX, [Tables + Basis + Index]
```

was den Assembler zu folgender Berechnung führen würde:

```
Adresse = Offset + [Basis] + [Index]
```

Um schließlich dem Ganzen die Krone aufzusetzen: Schön wäre es, wenn wir *Index* auch noch skalieren könnten, da wir eventuell nicht nur

Bytes konvertieren wollen, sondern vielleicht sogar DoubleWords. Dann würde uns ein Skalierungsfaktor gemäß

```
MOV     EAX, [Tables + Basis + (Faktor * Index)]
```

hier also der Faktor 4, erheblich helfen. Andernfalls müssten wir *Index* »von Hand« skalieren, bevor er benutzt würde. Der Assembler berechnet dies nach

```
Adresse = Offset + [Basis] + (Faktor * [Index])
```



Wie Sie bereits an der Darstellung der Adressberechnung sehen können, benötigt man für diese Form zwei Register (für die beiden indirekten Anteile) und eine Konstante (für den direkten Anteil). Das bedeutet, dass bei indirekter Adressierung mindestens ein und höchstens zwei Register zum Einsatz kommen.

LEA

Und hier kommt der Befehl LEA ins Spiel. Vielleicht ist Ihnen bislang nicht ausreichend klar geworden, worin der Unterschied zwischen *MOV OFFSET* und *LEA* liegt. Hier die Antwort: *MOV OFFSET* kann nur bei der direkten Adressierung benutzt werden, wenn die Adresse der gewünschten Speicherstelle zur Assemblierungszeit bekannt ist. Denn der Operator OFFSET erwartet ein Argument, das bereits deklariert wurde, also zur Assemblierungszeit bereits bekannt sein muss. *LEA* dagegen berechnet eine beliebige Adresse der eben hergeleiteten allgemeinen Form, wobei alle in geschweiften Klammern stehende Angaben optional sind und nur mindestens ein Ausdruck angegeben werden muss:

```
Adresse := {Const} + {[Reg1]} + {{F *} [Reg2]}
```



Die meisten Assembler optimieren den Code. Sobald nur konstante, zur Laufzeit bestimmbare Adressen ins Spiel kommen (*Adresse := Const*), übersetzen sie einen *LEA*-Befehl in den effektiveren *MOV*-Befehl mit *OFFSET*-Operanden und verwenden LEA nur dann, wenn indirekte Adressierungsanteile berücksichtigt werden müssen (z.B. *Adresse := [Reg1]*).

Das Kapitel »Befehlscodierung« (in der Assembler-Referenz) zeigt ausführlich, wie diese Möglichkeiten zur direkten und indirekten Adressierung in der Befehlssequenz codiert werden. Notwendig hierzu ist mindestens ein ModR/M-Byte (weil entweder eine Speicherstelle direkt angegeben wird und/oder ein Register für den indirekten Anteil

erforderlich ist). Je nach Umgebung (16- vs. 32-Bit-Umgebung) und Komplexizität der indirekten Adressierung kann auch ein SIB-Byte erforderlich werden. Einzelheiten stehen im genannten Kapitel.



Und noch eine Information: Der Assembler fasst alle konstanten Teile in einer Adressangabe zusammen. So ist durchaus folgendes Konstrukt denk- und programmierbar:

```
C1 = 12345678h
C2 = 01020304h

       MOV     EAX, [Var + (EBX + C1) + (2 * (ECX + C2))]
```

Der Assembler macht daraus:

```
8B 84 4B 14385C80r      mov eax, [Var+(EBX+C1)+(2*(ECX+C2))]
```

Sie erkennen nach dem Opcode-Byte $8B ein ModR/M- und ein zusätzliches SIB-Byte und dann die Konstante $1438_5C80, die sich wie folgt errechnet: $0000_0000r (= Var, das kleine »r« bedeutet: vom Linker noch zu berechnen!) + $1234_5678 (= C1) + $02040608 (= 2 * C2) = $1438_5C80r. Die Zeile hätte somit auch wie folgt programmiert werden können:

```
       MOV EAX, [(Var + 14385C80h) + EBX + (2 * ECX)]
```



Und noch ein Wort zum Thema direkter oder konstanter Anteil! Wie so häufig bei Assembler ist es ein Frage der Interpretation, wie welcher Teil einer Adressberechnung zu betrachten ist. So kann der direkte Anteil einer indirekten Adresse die Adresse der Tabelle sein und der indirekte ein zu berechnender Index in diese Tabelle: *Adresse = Offset + [Index]*. Allerdings ist auch vorstellbar, einen konstanten Index zu benutzen und die Adresse variabel sein zu lassen: *Adresse = [Offset] + Index*. Hierzu das Listing:

```
.data
Table DD ?

.code
Index = 12345678h

       MOV     EAX, [Table + EBX]  ; Index in EBX, Offset direkt
       MOV     EAX, [EBX + Index]  ; Offset in EBX, Index direkt
```

Dem Assembler ist das egal. Er prüft nur: Gibt es einen direkten (= konstanten, zur Assemblierungszeit berechenbaren) und einen indirekten

(= erst zur Laufzeit berechenbaren, vom Programmlauf abhängigen) Anteil und sind die Angaben valide. Was dahinter steckt, interessiert ihn nicht, wie man an folgendem Fragment des Assemblerlistings vom Quelltext von eben sehen kann:

```
8B 83   12345678        mov eax, [EBX + Index]
8B 83   00000000r       mov eax, [Table + EBX]
```

Der eigentliche Opcode und das ModR/M-Byte sind in beiden Fällen identisch: $8B $83. Nur die dem ModR/M-Byte folgende Adresse unterscheidet sich entsprechend (wobei das kleine »r« hinter der Adresse in der unteren Zeile wiederum dafür steht, dass der Linker hier noch den korrekten Offset der Variablen berechnen muss).



Das Thema Adressierung scheint einfacher, als es ist! Vielleicht haben Sie damit keine Schwierigkeiten, dann darf ich Sie aufrichtig beglückwünschen. Wenn Sie jedoch manchmal ein wenig »schwimmen«, wenn es um direkte und indirekte Adressierung geht und vor allem, wie man was dem Assembler kundtut, dann fühlen Sie sich in bester Gesellschaft. Immerhin war das Durcheinander, das hier teilweise herrscht, mit ein wesentlicher Grund für die Entwicklung des Ideal-Modus von TASM. Warum?

Schauen Sie sich bitte folgende Zeilen an:

```
ADD     EBX, DWordVar
ADD     EBX, [DWordVar]
ADD     EBX, DWORD PTR DWordVar
ADD     EBX, DWORD PTR [DWordVar]
```

Wie würden Sie diese Zeilen interpretieren? Hand aufs Herz!!

Wenden wir die Regeln von Hochsprachen wie C++ oder Delphi an, so müsste durch die Anweisung in Zeile 1 der Inhalt der Variablen *DWordVar* in das Register EBX kopiert werden. Denn immerhin vergleicht beispielsweise der Delphi-Compiler in *IF (I = 1)* nicht die Adresse der Speicherstelle, für die I steht, mit 1, sondern deren Inhalt. In Zeile 2 dagegen, so würden wir vermuten, würde der Inhalt von *DWordVar* einen Zeiger auf eine Speicherstelle beinhalten, die indirekt auszulesen ist. Da dieser Zeiger keinen Hinweis auf das Datum gibt, das an der bezeichneten Adresse steht, erwarten wir entweder eine Fehlermeldung des Assemblers mit entsprechendem Inhalt oder eine Warnung, dass dieser die Adresse als Zeiger auf ein DWord interpretiert, da es ja in ein DWord-Register kopiert werden soll. Um aber etwaige Probleme auszuschalten, programmieren wir Zeile 4, mit der der Sachverhalt klarge-

stellt wird. Zeile 3 würden wir als redundante Form von Zeile 1 betrachten, da ja *DWordVar* weiter oben als Datum vom Typ DWord deklariert wurde.

Lassen wir das Ganze von MASM oder TASM im MASM-Modus assemblieren, erhalten wir tatsächlich keine Fehlermeldung. Das ist eigentlich ein ganz gutes Indiz dafür, das wir mit unseren Annahmen Recht haben. Spaßeshalber – oder weil wir den IDEAL-Modus einmal testen möchten – assemblieren wir mit TASM in eben diesem Modus. Und erhalten – oh Schreck – folgende Kommentare von TASM:

```
03 1D 00000000r     add ebx, test1              ; Quelltextzeile 14
*Warning* test.asm(14) Pointer expression needs brackets
03 1D 00000000r     add ebx, [test1]            ; Zeile 15
03 1D 00000000r     add ebx, dword ptr test1    ; Zeile 16
*Warning* test.asm(16) Pointer expression needs brackets
03 1D 00000000r     add ebx, dword ptr [test1]  ; Zeile 17
**Error** test.asm(17) Illegal memory reference
*Warning* test.asm(17) Pointer expression needs brackets
```

(Die mit den Zahlen beginnenden Zeilen sind wie folgt zu interpretieren: Befehlssequenz bestehend aus Opcode ($03), ModR/M-Byte ($1D), Adresse ($0000_0000r), Quelltext und meinem Kommentar.)

Tolle Ausbeute: 75% Fehler bzw. Warnungen!! Warum? Ganz einfach! *DWordVar* ist ein Symbol, hinter dem sich eine Adresse verbirgt. Also ein Zeiger auf ein Datum – hier, wie Sie anhand des obigen Listingausschnitts leicht erkennen können, die Adresse $0000_0000 im Datensegment. Das bedeutet aber, dass Sie mit Zeile 14 einen Fehler machen, denn rein formal sagen Sie mit dieser Zeile: Lade den Wert von *DWordVar* in EBX. Und das wäre die Adresse $0000_0000, der Pointer, was Sie ja aber eigentlich gar nicht wollen! Sie wollen ja den Wert laden, der an der Stelle $0000_0000 im Speicher steht. Wollten Sie den Pointer laden, würden Sie ja den Operator OFFSET benutzen: *MOV EBX, OFFSET DWordVar*. Folglich darf Sie die Warnung Zeile 14 betreffend nicht wundern.

Zeile 15 scheint kommentarlos akzeptiert zu werden und damit richtig zu sein. Und wenn man es sich recht überlegt, auch mit Recht, lässt sie sich doch »übersetzen« mit: »Entnimm der mit *DWordVar* bezeichneten Adresse den Wert und lade ihn in EBX.«

Zeile 16 bemeckert TASM wiederum, was uns mit unserem neuen Wissen nicht wundert, da sie ja, wie bereits bemerkt, redundant zu Zeile 1 ist. Und die wurde bemängelt!

Tja – und dass Zeile 17 einen Fehler erzeugt, ist eigentlich fast klar! Denn indirekt lässt sich, wie oben gesagt, nur mit Registern, nicht aber mit Speicherstellen adressieren! Sobald eine Speicherstelle involviert ist, lässt sie sich nur als Konstante und damit als direkt angegebener Anteil einer Adresse einsetzen.

Also: Unter Assembler herrschen ein wenig andere Regeln als in Hochsprachen. Mal ehrlich: Hätten Sie's gewusst?

Bleibt noch die Frage, warum MASM und TASM im MASM-Modus nicht gemeckert haben. Die Antwort erkennen Sie auch aus dem Listingfragment oben: Alle Zeilen wurden in die gleiche Bytesequenz übersetzt. Dass dabei unser Vorhaben einer indirekten Adressierung mit Speicherstellen (Zeile 3 in unserem Quelltextfragment) geflissentlich – und ohne Benachrichtigung! – ignoriert und Code erzeugt wurde, den wir an dieser Stelle gar nicht wollten, hätten wir um ein Haar nicht bemerkt! Schlussfolgerung? Gewöhnen Sie sich zumindest für den Anfang gar nicht erst die »saloppe« Art vom MASM an, Dinge zu interpretieren. Falls Sie TASM haben, verwenden Sie vor allem als Neuling den IDEAL-Modus – solange, bis Ihnen das korrekte Assemblieren in Fleisch und Blut übergegangen ist. Und wenn Sie das nicht wollen, so zwingen Sie sich zumindest in Ihrem eigenen Interesse zu Disziplin.



Wenn Sie nun fragen, warum dann MASM diese laxe Semantik erlaubt, so sei auf die Frage der Interpretation weiter oben verwiesen. TASM im IDEAL-Modus lässt nur Ausdrücke innerhalb von einem eckigen Klammerpaar zu. Zusammengesetzte Ausdrücke wie *MOV EAX, Table[Index]* oder gar *Table2D[Index1][Index2]* sind strikt verboten – um der Disziplin und »reinen Lehre« willen. Gerade diese Form der Darstellung ist aber besonders übersichtlich, wenn man mit Tabellen oder Strukturen arbeitet. Insofern akzeptiert MASM – und damit auch TASM im MASM-Modus – diese Form der Adressangabe. Auf Kosten der Disziplin ...



Daher mein Tipp: Gewöhnen Sie sich zunächst an die verschiedenen Arten der Adressierung und wie sie »korrekt« verwendet werden. Je anspruchsvoller Ihre Assemblermodule dann werden, umso mehr können Sie die Zügel lockern. Denn wenn dann einmal etwas nicht so läuft, wie Sie es vermuten, so fällt Ihnen als eine mögliche Quelle diese Lockerung ein – und Sie überprüfen sie. Andernfalls bleibt das große Wundern ...



Zum Schluss noch eine wichtige Bemerkung! Wir sprachen in diesem Kapitel immer über Adressen und haben hierbei nur die Offsets betrachtet (entweder in Form einer Konstanten oder eines indirekt berechneten Wertes). Heißt das, dass alle Befehle, die auf Adressen zurückgreifen, nur die effektiven Adressen verwenden (vgl. »Zugriffe auf den Speicher: Von Adressen und Adressräumen« auf Seite 434? Klare Antwort: Nein! Jeder Befehl, der mit Adressen arbeitet, benutzt grundsätzlich vollständige logische Adressen bestehend aus Segment-Selektor und Offset – nur sieht man dies nicht auf den ersten Blick!

Der Grund ist einfach: Jede effektive Adresse bezieht sich auf irgendein Segment. So werden Adressen in Datensegmenten (und somit alle dort deklarierten Konstanten und Variablen) auf den Beginn des Datensegments bezogen, dessen Selektor ja im Segmentregister DS steht. Labels und Daten im Codesegment werden auf das Codesegment bezogen, dessen Selektor im Segmentregister CS steht. Und selbst Registern, die Adressen für die indirekte Adressierung verwenden, haben ein vorgegebenes Segment, auf das sie die Adressen beziehen. So wird immer das im Segmentregister SS stehende Stacksegment bemüht, wenn das Register, das bei der indirekten Adressierung die Basis aufnimmt, ESP oder EBP ist. Bei Stringbefehlen dienen je nach Befehl DS oder ES als Segmentregister für die Offsets in (E)SI oder (E)DI. Alle anderen Register beziehen sich standardmäßig auf DS.

Das bedeutet aber, dass eigentlich eine Adressangabe lege artis mit Angabe des betreffenden Segmentes zu erfolgen hat:

```
MOV EAX, Segmentregister:[Offset]
```

Nur wenn das zu verwendende Segmentregister das Standard-Segmentregister ist, kann es weggelassen werden, da die CPU es bei der Adressberechnung entsprechend berücksichtigt:

```
MOV EAX, SS:[EBP]
MOV EAX, [EBP]

MOV EAX, DS:[DWordVar + EBX + 2 * ECX]
MOV EAX, [DWordVar + EBX + 2 * ECX]
```

Ist das nicht der Fall, z.B. wenn EBP nicht mit dem Stack-, sondern mit dem Datensegment verwendet werden soll, so muss tatsächlich ein Segmentregister angegeben werden:

```
MOV EAX, [DS:EBP]
```

Der Assembler setzt dann ein *segment override prefix* (vgl. Seite 135) vor den Opcode, der das entsprechende Segmentregister als Bezugspunkt angibt – hier $3E für das Datensegmentregister DS:

```
8B 45  00      mov eax, [EBP]
3E: 8B 45 00   mov eax, [DS:EBP]
```

(Warum dem ModR/M-Byte eine Konstante $00 folgt, wird im Kapitel »Befehlscodierung« in der Assembler-Referenz erläutert).

Halten wir somit fest: Alle Adressangaben sind grundsätzlich logische Adressen, selbst wenn es den Anschein hat, als handelte es sich »nur« um effektive Adressen. Denn der betreffende Segment-Selektor wird entweder implizit durch die CPU oder explizit über segment override prefixes angegeben.



Das bedeutet aber auch, dass *Sie* verantwortlich dafür sind, die Segment-Register korrekt zu laden, falls Sie einmal Daten in Segmenten verwenden wollen, die nicht im Standard-Daten- oder -Stack-Segment liegen und damit vom Assembler und/oder Compiler verwaltet werden. Das erfolgt über die Segmentregister-Ladebefehle (vgl. Seite 141). Und vergessen Sie ASSUME nicht (vgl. Seite 635) ...

5.4 Ports

Die wohl wesentlichste, weil am häufigsten konsultierte Quelle von Daten für den Prozessor ist der Speicher. Um ihn anzusprechen hat die CPU so genannte Adress- und Datenleitungen. Über die Adressleitungen legt sie die Adresse der Speicherstelle an, die sie auslesen oder beschreiben will. Über die Datenleitungen erfolgt dann der Datenaustausch zwischen CPU und Speicher. Wie Sie bereits hinreichend wissen, verfügen moderne Prozessoren hierzu über mindestens 32 Adressleitungen und mindestens 32 Datenleitungen. Die Adressleitungen spannen einen Raum von 2^{32} = 4.294.967.296 Speicherzellen (= Bytes) auf, aus denen mit den Datenleitungen bis zu 32 Bit gleichzeitig ausgelesen oder beschrieben werden können. Es bleibt festzuhalten, dass die Adressen byteorientiert sind, was bedeutet, dass jede Adresse ein Byte im Speicher referenziert.

Doch ist die Kommunikation mit dem Speicher nicht die einzige Art, wie CPU oder FPU an Daten kommen können, ja vielleicht sogar noch nicht einmal die wichtigste. Denn nach dem Einschalten des Rechners

müssen ja Daten zunächst einmal in den Speicher geschrieben werden, um sie von dort auch wieder holen zu können. Es ist daher auch möglich, mit anderen PC-Komponenten, seiner »Umwelt«, zu kommunizieren. Unter Umwelt ist hierbei alles zu verstehen, das nicht zum Gespann CPU/FPU – Speicher gehört. Insbesondere sind das Peripheriegeräte wie Tastatur, Maus, Drucker, Diskettenlaufwerke und Festplatten, DVD-/CD-ROM-Laufwerke, DVD-/CD-Brenner, Scanner, Modem, Netzwerkkarte usw. usf. Um Datenaustausch mit diesen »Geräten« zu ermöglichen, greift der Prozessor auf dieselben 32 Adress- und 32 Datenleitungen zurück wie beim Zugriff auf den Speicher. Allerdings hat er einen weiteren Pin, mit dem er signalisiert, dass nun nicht der Speicher, sondern der »Input/Output«-Raum adressiert werden soll. Die Frage, wer sich angesprochen zu fühlen hat, wenn auf den Adressleitungen eine Adresse zum Datenaustausch anliegt, hat sich somit jedes Peripheriegerät selbst anhand des Status dieses M/IO#-Pins zu beantworten.



Bei der I/O-Adressierung benutzt der Prozessor jedoch nicht alle 32 Adressleitungen. Der I/O-Raum ist auf 2^{16} = 65.536 Bytes beschränkt, benötigt also lediglich die »unteren« 16 der 32 Adressleitungen. Kommunizieren jedoch kann er mit allen 32 Datenleitungen. Das bedeutet, dass der analog dem Speicher Byte-weise aufgebaute I/O-Raum ebenfalls Byte-, Word- oder DoubleWord-weise angesprochen werden kann. Die Adressen beziehen sich jedoch auch hier auf Bytes als kleinste Dateneinheit. Und auch hier, wie beim Speicher, sollten die I/O-Adressen ausgerichtet sein, um möglichst effektiv genutzt werden zu können. Das bedeutet, für Word-weisen Zugriff sollten die benutzten Adressen restlos durch 2 teilbar sein, bei DoubleWord-weisen Zugriffen durch 4. Abbildung 5.35 zeigt die ersten $0400 = 1.024 I/O-Plätze exemplarisch.

Während der Speicher-Raum physikalisch aus RAM-Chips gebildet wird, die einen bestimmten Bereich Adressen kontinuierlich abbilden, ist das beim I/O-Raum nicht der Fall. Jedes Peripheriegerät verfügt über eine Speicherstelle der für sie geeigneten Größe (Byte, Word, DoubleWord). Diese Speicherstelle wird aber an einer für dieses Gerät einzigartigen, reservierten Adresse/Adressbereich im I/O-Raum eingeblendet. Auf diese Weise wird der I/O-Raum physikalisch wie ein Mosaik zusammengesetzt. Und das bedeutet: Nicht an jeder Stelle im I/O-Raum muss auch tatsächlich physikalisch adressierbarer Speicherplatz vorhanden sein! Der I/O-Raum sieht somit eher aus wie Schweizer Käse: Er hat Löcher. Und zwar umso mehr, je weniger Peripherie-

geräte eingebunden wurden. (Wenn man jedoch Abbildung 5.35 betrachtet, kommt einem der Gedanke, dass da mehr Löcher sind als Käse ...)

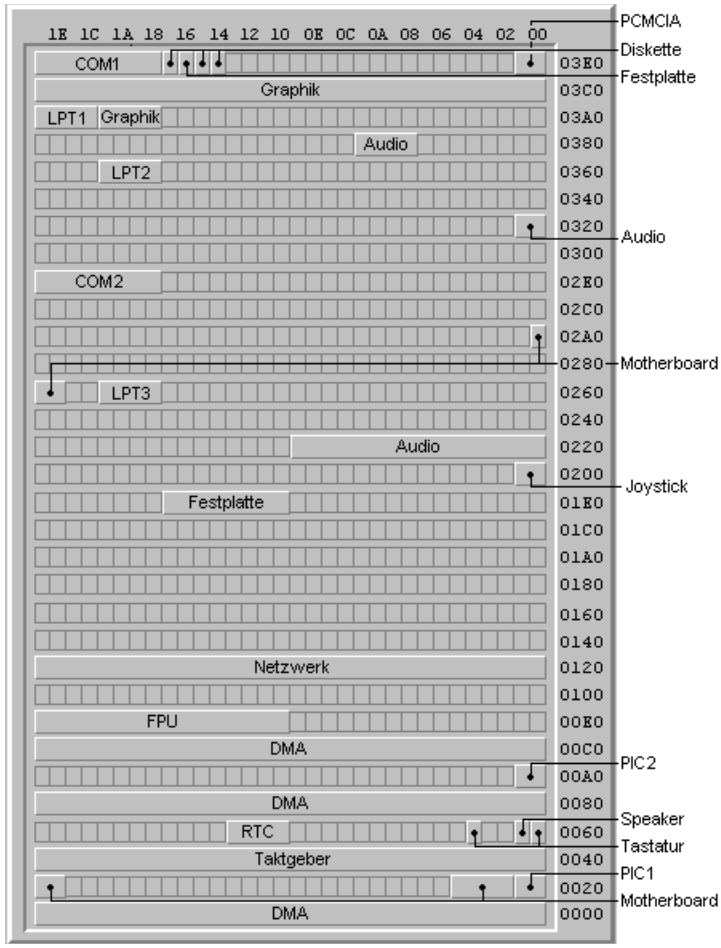


Abbildung 5.35: Beispielhafte Belegung des I/O-Adressraums mit Peripheriegeräten

So benutzt beispielsweise der Festplattencontroller eines alten Laptops von mir die I/O-Bereiche $01F0 bis $01F7 sowie $03F6; das Diskettenlaufwerk $03F2, $03F4 bis $03F5 und $03F7; die Graphik-Karte den Bereich $03B0 bis $03BB sowie $03C0 bis $03DF; die serielle Schnittstelle COM1 $03F8 bis 03FF; COM2 $02F8 bis $02FF; und die parallele Schnittstelle LPT1$0378 bis $037B. Der PC-Lautsprecher nutzt $0061;

der System-Taktgeber den Bereich $0040 bis $005F. Die Tastatur benutzt $0060 und $0064; die CMOS-Echtzeituhr (RTC, real time clock) den Bereich $0070 bis $0073; die Netzwerkkarte $0120 bis $013F; der Audio-Controller $0220 bis $022F, $0388 bis $038B und $0320 bis $0321. Der Bereich $00F0 bis $00FF ist durch Intel (für die FPU) reserviert, die beiden PICs (programmable interrupt controller) blenden sich bei $0020 bis $0021 bzw. $00A0 bis $00A1 ein. Die benutzten I/O-Adressen nennt man auch »Ports« (engl.: Tor, Pforte), da sie quasi das »Tor zur Peripherie« darstellen, die ansonsten CPU-unabhängig vor sich hinwerkelt.

Das bedeutet also, dass mit jedem Peripheriegerät kommuniziert werden kann, wenn man seine »Ports« kennt und weiß, was sich dahinter auf Seite der Peripherie verbirgt. Hierzu besitzt die CPU mit den Befehlen IN, INS, OUT und OUTS vier Befehle: IN und INS übernehmen das Auslesen der Daten aus dem Port, OUT und OUTS schreiben Daten in einen Port. Dies können je nach »Breite« des Ports Bytes, Words und DoubleWords sein, so wie man auch Byte-, Word- und DoubleWord-Variablen im Speicher deklarieren kann.



Bitte verlassen Sie sich nicht darauf, dass auch tatsächlich an der bezeichneten Stelle das genannte Peripheriegerät steckt. In der Vergangenheit hat hier fast jeder sein eigenes Süppchen gekocht, was für eine große Reihe von Inkompatibilitäten gesorgt hat. Auch besteht die Möglichkeit, dass I/O-Adressen »von Hand« oder durch das Betriebssystem zugeordnet werden. Abbildung 5.35 soll nur exemplarisch der Veranschaulichung dienen, wie sich der I/O-Raum des Prozessors aufbauen kann!

An dieser Stelle 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? Also Bereiche im Speicher, die für die Kommunikation mit Peripherie dienen? Denn ganz offensichtlich hat die Verwendung von Daten via Ports nicht nur Vorteile: So ist z.B. der gesamte I/O-Raum auf 64 kByte beschränkt!

Ja, man kann. Und man tut das – Sie kennen auch 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.



Wenn Sie nochmals in Abbildung 5.35 schauen, werden Sie auch im I/O-Adressraum Bereiche finden, die die Grafikkarte benutzt. Das ist kein Widerspruch! Über den/die Ports kommuniziert die CPU mit den Kontroll- und Status-Registern der Grafikkarte, die Daten werden über den Speicher übergeben.

Diese Verhältnisse aber erklären ganz selbstverständlich einen Begriff, der häufig herumgeistert und den viele nicht so richtig verstehen: memory-mapped I/O, *auf den Speicher abgebildete Ein-/Ausgabe*. Peripheriegeräte, die in der Lage sind, zwecks Datenübergabe 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.

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 daher 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 die Prozessorbefehle IN, INS, OUT und OUTS eingesetzt werden.

5.5 Befehls-Decodierung

In diesem Kapitel sind die Tabellen angegeben, die Sie benötigen, wenn Sie einen Befehl anhand seiner Befehlssequenz decodieren möchten. Dies erfolgt in vier Schritten:

- Decodierung der Präfixe
- Decodierung des Opcodes
- Decodierung des eventuell vorhandenen ModR/M- und ggf. SIB-Bytes
- Decodierung einer eventuell vorhandenen Adresse oder Konstante.

5.5.1 Decodierung des/der Präfixe(s)

Die Präfixe sind Ein-Byte-Codes, die Sie in der Tabelle für Ein-Byte-Opcodes (Tabelle 5.14) finden.

5.5.2 Decodierung des Opcodes

Generell gibt es Ein-, Zwei- und Drei-Byte-Opcodes. Ein-Byte-Opcodes sind in Tabelle 5.14 für CPU- und SIMD-Befehle und in Tabelle 5.20 für FPU-Befehle dargestellt. Die Zwei-Byte-Opcodes sind im Falle von CPU- und FPU-Befehlen in Tabelle 5.15, im Falle von FPU-Befehlen in Tabelle 5.21 dargestellt. Einige SIMD-Befehle verwenden Präfixe zur weiteren Codierung. Solche »Drei-Byte-Opcodes« finden Sie in Tabelle 5.16 bis Tabelle 5.18.

Im Falle von CPU- oder SIMD-Befehlen können sowohl Ein-Byte- wie auch Zwei-Byte-Opcodes ein ModR/M- (und ggf. SIB-)Byte besitzen, das die Operanden und die Art der Adressierung codiert. In Tabelle 5.14 sind die Befehle grau unterlegt, die ein solches ModR/M-Byte benötigen. Bei FPU-Befehlen haben grundsätzlich alle Ein-Byte-Opcodes ein ModR/M-Byte im Gefolge, da mit Ein-Byte-Opcodes generell Befehle codiert werden, die Operanden besitzen. Die Zwei-Byte-FPU-Opcodes dagegen haben kein ModR/M-Byte.

Bei CPU- und SIMD-Befehlen können, bei FPU-Befehlen müssen die Bits 5 bis 3 zur Codierung herangezogen werden. In diesen Fällen spricht man vom *Spec*-Feld (»special«) des ModR/M-Bytes. Solche Befehle sind in Tabelle 5.14 ebenfalls grau unterlegt und mit »group x« bezeichnet. Sie werden in Tabelle 5.19 dargestellt.

5.5.3 Decodierung eines ModR/M- und ggf. eines SIB-Byte

Zur Decodierung eines eventuellen ModR/M und ggf. vorhandenen SIB-Bytes gibt es ein Flussdiagramm, das die Decodierung vereinfacht, sobald ein Speicheroperand im ModR/M-Byte codiert ist. Dieses Flussdiagramm finden Sie in Abbildung 5.36. Falls als Operand keine Speicherstelle involviert ist, ist die Decodierung des ModR/M-Bytes einfach: mod hat dann den Wert 11b und reg und R/M codieren jeweils ein Register.

5.5.4 Decodierung einer Adresse oder Konstanten

Die Dekodierung einer Adresse und/oder einer Konstanten, die im Rahmen des ModR/M-Bytes oder aufgrund des Opcodes erforderlich sind, ist ebenfalls einfach. In den Tabellen zu den Opcodes ist über eine Fußnote verzeichnet, mit welchen Operandengrößen dieser Befehl arbeitet. Stellt sich anhand der Decodierung des ModR/M-Bytes heraus, dass eine Adresse involviert ist, folgt sie unmittelbar auf das ModR/M- bzw. SIB-Byte. Die Anzahl der Bytes, mit der eine solche Adresse codiert ist, ergibt sich aus der aktuellen Umgebung sowie dem eventuell vorhandenen Präfix address size override. Zeigt eine Fußnote in einer Opcode-Tabelle an, dass der Befehl eine Konstante involviert, so folgt diese in der Befehlssequenz der Adresse, dem ModR/M-Byte oder direkt dem Opcode.

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	add[1,4]	add[2,4]	add[1,3]	add[2,3]	add[1,5,6]	add[2,5,6]	push es	pop es	or[1,4]	or[2,4]	or[1,3]	or[2,3]	or[1,5,6]	or[2,5,6]	push cs	EXT
1_	adc[1,4]	adc[2,4]	adc[1,3]	adc[2,3]	adc[1,5,6]	adc[2,5,6]	push ss	pop ss	sbb[1,4]	sbb[2,4]	sbb[1,3]	sbb[2,3]	sbb[1,5,6]	sbb[2,5,6]	push ds	pop ds
2_	and[1,4]	and[2,4]	and[1,3]	and[2,3]	and[1,5,6]	and[2,5,6]	ES:	DAA	sub[1,4]	sub[2,4]	sub[1,3]	sub[2,3]	sub[1,5,6]	sub[2,5,6]	CS:	DAS
3_	xor[1,4]	xor[2,4]	xor[1,3]	xor[2,3]	xor[1,5,6]	xor[2,5,6]	SS:	AAA	cmp[1,4]	cmp[2,4]	cmp[1,3]	cmp[2,3]	cmp[1,5,6]	cmp[2,5,6]	DS:	AAS
4_[2]	inc eax	inc ecx	inc edx	inc ebx	inc esp	inc ebp	inc esi	inc edi	dec eax	dec ecx	dec edx	dec ebx	dec esp	dec ebp	dec esi	dec edi
5_[2]	push eax	push ecx	push edx	push ebx	push esp	push ebp	push esi	push edi	pop eax	pop ecx	pop edx	pop ebx	pop esp	pop ebp	pop esi	pop edi
6_	pusha(d)	popa(d)	bound[F]	arpl[4,C]	FS:	GS:	opsize	addrsize	push[2,5]	imul[2,3,5]	push[1,5]	imul[2,3,5]	ins[1]	ins[2]	outs[1]	outs[2]
7_[1,5]	jo	jno	jb/jnae/jc	jnb/jae/jnc	jz/je	jnz/jne	jbe / jna	jnbe / ja	js	jns	jp/jpe	jnp/jpo	jl/jnge	jnl/jge	jle/jng	jnle/jg
8_	immediate group 1				test[1]	test[2]	xchg[1]	xchg[2]	mov[1,3]	mov[2,3]	mov[1,4]	mov[2,4]	mov[B]	lea	mov[A]	pop
9_	nop[E]	xchg[2,8] ecx	xchg[2,8] edx	xchg[2,8] ebx	xchg[2,8] esp	xchg[2,8] ebp	xchg[2,8] esi	xchg[2,8] edi	cbw/cwde	cwd/cdq	call (far)	wait/fwait	pushf(d)	popf(d)	sahf	lahf
A_	mov[1,5,6]	mov[2,5,6]	mov[1,5,7]	mov[2,5,7]	movs[1]	movs[2]	cmps[1]	cmps[2]	test[1,5,6]	test[2,5,6]	stos[1]	stos[2]	lods[1]	lods[2]	scas[1]	scas[2]
B_	mov[5,9] al	mov[5,9] cl	mov[5,9] dl	mov[5,9] bl	mov[5,9] ah	mov[5,9] ch	mov[5,9] dh	mov[5,9] bh	mov[5,9] eax	mov[5,9] ecx	mov[5,9] edx	mov[5,9] ebx	mov[5,9] esp	mov[5,9] ebp	mov[5,9] esi	mov[5,9] edi
C_	shift group 2		ret[5] (near)	ret (near)	les[3]	lds[3]	group 11		enter[D]	leave	ret[5] (far)	ret (far)	int3	int[5]	into	iret
D_	shift group 2				aam[1,5]	aad[1,5]	reserved	xlat(b)	ESC							
E_	loopne	loope	loop	j(e)cxz	in[1,5,6]	in[2,5,6]	out[1,5,7]	out[2,5,7]	call[5] (near)	jmp[5] (near)	jmp[5] (far)	jmp[5] (short)	in[1,6]	in[2,6]	out[1,7]	out[2,7]
F_	lock	reserved	repne	rep(e)	hlt	cmc	unary group 3		clc	stc	cli	sti	cld	std	group 4	group 5

Die hellgrau unterlegten Felder verweisen entweder auf Befehlsgruppen, deren Unterscheidung anhand des spec-Feldes eines folgenden ModR/M-Bytes erfolgt, oder auf Befehle, denen zur Codierung eines zweiten (Speicher-)Operanden ein ModR/M-Byte folgt. Die dunkelgrau unterlegten Felder verweisen auf Befehle mit Zwei-Byte-Opcodes, deren erstes Byte entweder das Opcode-Shift-Byte $0F ist (CPU-Befehle) oder die mit der ESC-Sequenz beginnen (FPU-Befehle).

Ferner bedeuten: [1]: Byte-Operanden; [2]: je nach Umgebung DoubleWord- bzw. Word-Operanden; [3]: Ziel ist Register, Quelle ist Register/Speicher; [4]: Ziel ist Register/Speicher, Quelle ist Register; [5]: Konstante folgt; [6]: Akkumulator Zieloperand; [7]: Akkumulator Quelloperand; [8]: Austausch des angegebenen Registers mit EAX; [9]: Kopieren einer Konstante in das angegebene Register; [A]: Ziel ist Segmentregister; [B]: Quelle ist Segmentregister; [C]: Word-Operanden; [D]: Double Word- und Byte-Konstante folgt; [E]: nop = xchg eax, eax; [F]: Ziel ist Register, Quelle ist Speicherstelle.

Tabelle 5.14: Ein-Byte-Opcodes

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	group 6	group 7	lar[2,3]	lsl[2,3]	reserved	reserved	clts	reserved	invd	wbinvd	reserved	ud2	reserved	prefetch *)	femms *)	*)
1_	movups[3]	movups[4]	movlps[3]	movlps[4]	unpcklps	unpckhps	movhps[3]	movhps[4]	group 16	reserved	reserved	reserved	reserved	reserved	reserved	reserved
2_	mov[7]	mov[A]	mov[8]	mov[9]	reserved	reserved	reserved	reserved	movaps[3]	movaps[4]	cvtpi2ps[3]	movntps[I]	cvttps2pi[3]	cvtps2pi[3]	ucomiss[3]	comiss[3]
3_	wrmsr	rdtsc	rdmsr	rdpmc	sysenter	sysexit	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
4_[B]	cmovo	cmovno	cmovb	cmovae	cmove	cmovne	cmovbe	cmova	cmovs	cmovns	cmovp	cmovnp	cmovl	cmovnl	cmovle	cmovnle
5_	movmskps[G]	sqrtps[3]	rsqrtps[3]	rcpps[3]	andps[3]	andnps[3]	orps[3]	xorps[3]	addps[3]	mulps[3]	cvtps2pd[3]	cvtdq2ps[3]	subps[3]	minps[3]	divps[3]	maxps[3]
6_	punpcklbw[3]	punpcklwd[3]	punpckldq[3]	packsswb[3]	pcmpgtb[3]	pcmpgtw[3]	pcmpgtd[3]	packuswb[3]	punpckhbw[3]	punpckhwd[3]	punpckhdq[3]	packssdw[3]	reserved	reserved	movd[E]	movq[3]
7_	pshufw	group 12	group 13	group 14	pcmpeqb	pcmpeqw	pcmpeqd	emms	MMX UD						movd[F]	movq[4]
8_[2,5]	jo	jno	jb/jnae/jc	jnb/jae/jnc	jz/je	jnz/jne	jbe / jna	jnbe / ja	js	jns	jp/jpe	jnp/jpo	jl/jnge	jnl/jge	jle/jng	jnle/jg
9_[B]	seto	setno	setb	setae	sete	setne	setbe	seta	sets	setns	setp	setnp	setl	setnl	setle	setnle
A_	push fs	pop fs	cpuid	bt[2,3]	shld[2,4,5]	shld[2,4,6]	reserved	reserved	push gs	pop gs	rsm	bts[2,3]	shrd[2,4,5]	shrd[2,4,6]	group 15	imul[2]
B_	cmpxchg[1]	cmpxchg[2]	lss[3]	btr[2,3]	lfs[3]	lgs[3]	movzx[3,C]	movzx[3,D]	reserved	group 10	group 8	btc[2,3]	bsf[2,3]	bsr[2,3]	movsx[3,C]	movsx[3,D]
C_	xadd[1]	xadd[2]	cmpps[3,5]	movnti[H]	pinsrw[F,5]	pextrw[F,5]	shufps	group 9	bswap eax	bswap ecx	bswap edx	bswap ebx	bswap esp	bswap ebp	bswap esi	bswap edi
D_	reserved	psrlw[3]	psrld[3]	psrlq[3]	reserved	pmulw[3]	reserved	pmovmskb[3]	psubusb[3]	psubusw[3]	pminub[3]	pand[3]	paddusb[3]	paddusw[3]	pmaxub[3]	pandn[3]
E_	pavgb[3]	psraw[3]	psrad[3]	pavgw[3]	pmulhuw[3]	pmulhw[3]	reserved	movntq[I]	psubsb[3]	psubsw[3]	pminsw[3]	por[3]	paddsb[3]	paddsw[3]	pmaxsw[3]	pxor[3]
F_	reserved	psllw[3]	pslld[3]	pshllq[3]	reserved	pmaddwd[3]	psadbw[3]	maskmovq[J]	psubb[3]	psubw[3]	psubd[3]	reserved	paddb[3]	paddw[3]	paddd[3]	reserved

Die hellgrau unterlegten Felder verweisen entweder auf Befehlsgruppen, deren Unterscheidung anhand des spec-Feldes eines folgenden ModR/M-Bytes erfolgt, oder auf Befehle, denen zur Codierung eines zweiten (Speicher-)Operanden ein ModR/M-Byte folgt.

Ferner bedeuten: [1]: Byte-Operanden; [2]: je nach Umgebung DoubleWord- bzw. Word-Operanden; [3]: Ziel ist Register, Quelle ist Register/Speicher; [4]: Ziel ist Register/Speicher, Quelle ist Register; [5]: Konstante folgt; [6]: CL-Register ist Operand; [7]:Ziel ist Kontrollregister ; [8]: Quelle ist Kontrollregister; [9]: Ziel ist Debugregister; [A]: Quelle ist Debugregister [B]: Alias-Namen nicht aufgeführt; [C]: 8-Bit-Quelloperand; [D]: 16-Bit-Quelloperand; [E]: Ziel ist MMX-Register, Quelle ist 32-Bit-CPU-Register; [F]: Ziel ist 32-Bit-CPU-Register, Quelle ist MMX-Register; [G]: Ziel ist 32-Bit-CPU-Register, Quelle ist XMM-Register; [H]: Ziel ist 32-Bit-Speicherstelle, Quelle ist 32-Bit-CPU-Register; [I]: Ziel ist Speicherstelle, Quelle ist Register; [J]: Beide Operanden sind Register.

*) Der Befehl FEMMS ($0F $0E), der Befehl PREFETCH/PREFETCHW ($0F $0D) und der »Umschaltcode« $0F $0F für die 3D-Now-Berehle sind bei Intel-Prozessoren nicht implementiert. Sie finden nur in AMD-Prozessoren mit 3D-Now!-Erweiterung Verwendung. Bei Intel gelten die Zwei-Byte-Opcodes $0F $0D, $0F $0E , und $0F $0F als reserviert.

Tabelle 5.15: Zwei-Byte-CPU- und SIMD-Opcodes. Das erste Byte ist immer $0F.

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_																
1_	movss[1]	movss[2]	movhlps				movhps									
2_											cvtsi2ss[1]		cvttss2si[1]	cvtss2si[1]		
3_																
4_[B]																
5_		sqrtss[1]	rsqrtss[1]	rcpss[1]					addss[1]	mulss[1]	cvtss2sd[1]	cvttps2dq[1]	subss[1]	minss[1]	divss[1]	maxss[1]
6_																movdqu[1]
7_															movq[1]	movdqu[2]
8_[5]																
9_[B]																
A_																
B_																
C_			cmpss[1,3]													
D_							movq2dq[4]									
E_							cvtdq2pd[1]									
F_																

Die hellgrau unterlegten Felder verweisen entweder auf Befehlsgruppen, deren Unterscheidung anhand des spec-Feldes eines folgenden ModR/M-Bytes erfolgt, oder auf Befehle, denen zur Codierung eines zweiten (Speicher-)Operanden ein ModR/M-Byte folgt. Das erste (»Präfix-«)Byte dieser Befehle ist $F3, das zweite Byte das Opcode-Shift-Byte $0F. Das dritte Byte ist in dieser Tabelle dargestellt.
Es bedeuten: [1]: Ziel ist Register, Quelle ist Register/Speicher; [2]: Ziel ist Register/Speicher, Quelle ist Register; [3]: Konstante folgt, [4]: Ziel ist XMM-Register, Quelle ist MMX-Register

Tabelle 5.16: »Drei-Byte«-SIMD-Opcodes mit führendem Präfix $F3 und erstem Opcode-Byte $0F

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_																
1_	movupd[1]	movupd[2]	movlpd[1]	movlpd[2]	unpcklpd[1]	unpckhpd[1]	movhpd[1]	movhpd[2]								
2_									movapd[1]	movapd[2]	cvtpi2pd[1]	movntpd[1]	cttpd2pi[1]	cvtpd2pi	ucomisd[1]	comisd[1]
3_																
4_[B]																
5_	movmskpd[5]	sqrtpd[1]			andpd[1]	andnpd[1]	orpd[1]	xorpd[1]	addpd[1]	mulpd[1]	cvtpd2ps[1]	cvtps2dq[1]	subpd[1]	minpd[1]	divpd[1]	maxpd[1]
6_	punpcklbw[1]	punpcklwd[1]	punpckldq[1]	packsswb[1]	pcmpgtb[1]	pcmpgtw[1]	pcmpgtd[1]	packuswb[1]								
7_	pshufw[1,9]	group 12	group 13	group 14	pcmpeqb[1]	pcmpeqw[1]	pcmpeqd[1]									
8_[5]																
9_[B]																
A_																
B_																
C_			cmppd[1,3]		pinsrw[3,4]	pextrw[3,5]	shufpd[1,3]									
D_		psrlw[1]	psrld[1]	psrlq[1]		pmulw[1]	movq[2]	pmovmskb[1]	psubusb[1]	psubusw[1]	pminub[1]	pand[1]	paddusb[1]	paddusw[1]	pmaxub[1]	pandn[1]
E_	pavgb[1]	psraw[1]	psrad[1]	pavgw[1]	pmulhuw[1]	pmulhw[1]	cvttpd2dq[1]	movntdq[8]	psubsb[1]	psubsw[1]	pminsw[1]	por[1]	paddsb[1]	paddsw[1]	pmaxsw[1]	pxor[1]
F_		psllw[1]	pslld[1]	pshllq		pmaddwd[1]		maskovdqu[7]	psubb[1]	psubw[1]	psubd[1]		paddb[1]	paddw[1]	paddd[1]	

Die hellgrau unterlegten Felder verweisen entweder auf Befehlsgruppen, deren Unterscheidung anhand des spec-Feldes eines folgenden ModR/M-Bytes erfolgt, oder auf Befehle, denen zur Codierung eines zweiten (Speicher-)Operanden ein ModR/M-Byte folgt. Das erste (»Präfix-«)Byte dieser Befehle ist $66, das zweite Byte das Opcode-Shift-Byte $0F. Das dritte Byte ist in dieser Tabelle dargestellt.
Es bedeuten: [1]: Ziel ist Register, Quelle ist Register/Speicher; [2]: Ziel ist Register/Speicher, Quelle ist Register; [3]: Konstante folgt, [4]: Ziel ist MMX-Register, Quelle ist 32-Bit-CPU-Register; [5]: Ziel ist 32-Bit-CPU-Register, Quelle ist XMM-Register; [6]: Ziel ist Speicherstelle, Quelle ist Register; [7]: beide Operanden sind Register, [8]: Ziel ist Speicherstelle, Quelle XMM-Register, [9]: Konstante folgt.

Tabelle 5.17: »Drei-Byte«-SIMD-Opcodes mit führendem Präfix $66 und erstem Opcode-Byte $0F

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_																
1_	movsd[1]	movsd[2]														
2_											cvtsi2sd[1]		cvttsd2si[1]	cvtsd2si[1]		
3_																
4_[B]																
5_		sqrtsd[1]							addsd[1]	mulsd[1]	cvtsd2ss[1]		subsd[1]	minsd[1]	divsd[1]	maxsd[1]
6_																
7_																
8_[5]																
9_[B]																
A_																
B_																
C_			cmpsd[1,3]													
D_							movdq2q[4]									
E_							cvtpd2dq[1]									
F_																

Die hellgrau unterlegten Felder verweisen entweder auf Befehlsgruppen, deren Unterscheidung anhand des spec-Feldes eines folgenden ModR/M-Bytes erfolgt, oder auf Befehle, denen zur Codierung eines zweiten (Speicher-)Operanden ein ModR/M-Byte folgt. Das erste (»Präfix-«)Byte dieser Befehle ist $F2, das zweite Byte das Opcode-Shift-Byte $0F. Das dritte Byte ist in dieser Tabelle dargestellt.
Es bedeuten: [1]: Ziel ist Register, Quelle ist Register/Speicher; [2]: Ziel ist Register/Speicher, Quelle ist Register; [3]: Konstante folgt, [4]: Ziel ist MMX-Register, Quelle ist XMM-Register

Tabelle 5.18: »Drei-Byte«-SIMD-Opcodes mit führendem Präfix $F2 und erstem Opcode-Byte $0F

Gruppe	spec field (bits 5 bis 3 des ModR/M-Bytes)								Opcode existiert mit		Opcode
	000	001	010	011	100	101	110	111	mod = 11	mod ≠ 11	
1	add	or	adc	sbb	and	sub	xor	cmp	ja	ja	80: xxx r/m8, c8; 81: xxx r/m, c; 83: xxx r/m, c8
2	rol	ror	rcl	rcr	shl/sal	shr	reserved	sar	ja	ja	C0: xxx r/m8, c8; C1: xxx r/m, c8; D0:xxx r/m8, 1; D1: XXX r/m, 1; D2: xxx r/m8, CL; D3: xxx r/m, CL
3	test	reserved	not	neg	mul	imul	div	idiv	ja	ja	F6: xxx r/m8, c8; F7: xxx r/m, c
4	inc	dec	reserved	reserved	reserved	reserved	reserved	reserved	ja	ja	FE: xxx r/m8
5	inc	dec	call (near)	call (far)	jmp (near)	jmp (far)	push	reserved	ja	ja	FF: xxx r/m
6	sldt	str	lldt	ltr	verr	verw	reserved	reserved	ja	ja	0F 00: xxx r/m
7	sgdt	sidt	lgdt	lidt	smsw	reserved	lmsw	invlpg	(ja)	(ja)	0F 01: xxx m (sgdt, sidt, lgdt, lidt), xxx r (smsw, lmsw, invdpg)
8	reserved	reserved	reserved	reserved	bt	bts	btr	btc	ja	ja	0F BA:: xxx r/m, c8
9	reserved	cmpxchg8b	reserved	reserved	reserved	reserved	reserved	reserved	nein	ja	0F C7: cmpxchg8b m
10	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved			0F B9:
11	mov	reserved	reserved	reserved	reserved	reserved	reserved	reserved	ja	ja	C6: mov r/m8, c8; C7: mov r/m, c
12	reserved	reserved	psrlw	reserved	psraw	reserved	psllw	reserved	ja	nein	0F 71: xxx mm, c8
13	reserved	reserved	psrld	reserved	psrad	reserved	pslld	reserved	ja	nein	0F 72: xxx mm, c8
14	reserved	reserved	psrlq	reserved	reserved	reserved	psllq	reserved	ja	nein	0F 73: xxx mm, c8
15	fxsave	fxrstor	ldmxcsr	stmxcsr	reserved	lfence	mfence	*)	(nein)	ja	0F AE: xxx m; bis auf 0F AE /7 (clflush/sfence) mod = 11 nicht erlaubt
16	prefetchnta	prefetcht0	prefetcht1	prefetcht2	reserved	reserved	reserved	reserved	nein	ja	0F 18: xxx m

Es bedeuten: xxx: betreffender Befehl; r/m8: Byte-Register oder -Speicheroperand; r/m: je nach Umgebung 32- bzw. 16-Bit-Register oder -Speicheroperand; c8: 8-Bit-Konstante; c: zum Operanden passende Konstante (32 bzw. 16 Bit); mm: MMX-Register; m: Speicheroperand der passenden Größe.

*) wenn mod ≠ 11 (d. h. Speicherzugriff erlaubt): clflush; wenn mod = 11 (d.h. nur Register als Operanden): sfence

Beispiel zur Verwendung der Tabelle: Die Gruppe 1 erstreckt sich laut Tabelle 5.14 über die Opcodes $80 bis $83 und ist hier in Zeile 1 repräsentiert. Die Opcodes verlangen ein ModRM-Byte, in dem das mod-Feld alle Werte annehmen kann (»Opcode existiert mit mod = 11 (ja) und mod ≠ 11 (ja)«. Das bedeutet sowohl ein Register (mod = 11) als auch eine Speicherstelle (mod ≠ 11) mit direkter oder indirekter Adressierung als Operand infrage kommt. Das spec-Feld des ModRM-Bytes definiert nun den Befehl (z.B. spec = 000; Add; spec = 101: SUB), das übrig bleibende R/M-Feld entweder einen Register- (mod = 11) oder Speicheroperanden (mod ≠ 11). Die in der Gruppe definierten Opcodes stehen in der Spalte »Opcodes«: In Gruppe 1 die Opcodes $80 (Byte-Operanden und -Konstante; xxx r/m8, c8), $81 (Standard-Operand und -Konstante:xxx r/m, c) oder $83 (Standard-Operand und Byte-Konstante: xxx r/m, c8). $82 ist nicht vergeben und gilt als reserviert. In Gruppe 9 (Tabelle 5.15, $0F $C7) sind keine Register- (mod = 11: nein) sondern nur Speicheroperanden (mod ≠ 11: ja) gestattet. Derzeit gibt es nur einen Befehl in dieser Gruppe – CMPXCHG8B – der mit spec = 001 kodiert wird. Alle anderen Kodierungen sind reserviert.

Tabelle 5.19: Spec-Feld im ModR/M-Byte der Befehle der Gruppen 1 bis 16

Opcode-Byte	spec field (bits 5 bis 3 des ModR/M-Bytes), wenn mod ≠ 11								spec field (bits 5 bis 3 des ModR/M-Bytes), wenn mod = 11							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
D8	fadd[1]	fmul[1]	fcom[1]	fcomp[1]	fsub[1]	fsubr[1]	fdiv[1]	fdivr[1]	fadd[8]	fmul[8]	fcom[8]	fcomp[8]	fsub[8]	fsubr[8]	fdiv[8]	fdivr[8]
D9	fld[1]	reserved	fst[1]	fstp[1]	fldenv	fldcw	fstenv	fstcw	fld[8]	fxch[8]	group A					
DA	fiadd[5]	fimul[5]	ficom[5]	ficomp[5]	fisub[5]	fisubr[5]	fidiv[5]	fidivr[5]	fcmovb[8]	fcmove[8]	fcmovbe[8]	fcmovu[8]	reserved	group B	reserved	reserved
DB	fild[5]	reserved	fist[5]	fistp[5]	reserved	fld[3]	reserved	fstp[3]	fcmovnb[8]	fcmovne[8]	fcmovnbe[8]	fcmovnu[8]	group C	fucomi[8]	fcomi[8]	reserved
DC	fadd[2]	fmul[2]	fcom[2]	fcomp[2]	fsub[2]	fsubr[2]	fdiv[2]	fdivr[2]	fadd[9]	fmul[9]	reserved	reserved	fsubr[9]	fsub[9]	fdivr[9]	fdiv[9]
DD	fld[2]	reserved	fst[2]	fstp[2]	frstor	reserved	fsave	fstsw	ffree[A]	reserved	fst[A]	fstp[A]	fucom[9]	fucomp[A]	reserved	reserved
DE	fiadd[6]	fimul[6]	ficom[6]	ficomp[6]	fisub[6]	fisubr[6]	fidiv[6]	fidivr[6]	faddp[9]	fmulp[9]	reserved	group D	fsubrp[9]	fsubp[9]	fdivrp[9]	fdivp[9]
DF	fild[6]	reserved	fist[6]	fistp[6]	fbld[4]	fild[7]	fbstp[4]	fistp[7]	reserved	reserved	reserved	reserved	group E	fucomip[8]	fcomip[8]	reserved

Diese Tabelle stellt die FPU-Befehle dar, die ein Opcode-Byte und ein zusätzliches ModR/M-Byte zur Kodierung der verwendeten Operanden verwenden. Das in dieser Tabelle ebenfalls dargestellte ModR/M-Byte besteht aus drei Feldern: *mod* (Bits 7 und 6), *spec* (Bits 5 bis 3) und *R/M* (Bits 2 bis 0). Hat *mod* den Wert 11_b, so ist der zweite Operand ein FPU-Register. *R/M* gibt dann den Code des Registers an, der verwendet wird (000 = ST(0) bis 111 = ST(7). Für diesen Fall gilt die rechte Seite der Tabelle. Ist dagegen *mod* = 00_b, 01_b oder 10_b, so verwendet der Befehl einen Speicheroperanden. Für diesen Fall gilt die linke Seite der Tabelle.

Die hellgrau unterlegten Felder verweisen auf Befehlsgruppen, deren Befehle von Operanden unabhängig sind und die daher mit einzelnen Opcodes belegt sind. Diese Zwei-Byte-FPU-Opcodes sind in Tabelle 5.21 dargestellt.

Es bedeuten: [1]: SingleReal als Operand; [2]: DoubleReal als Operand; [3]: ExtendedReal als Operand; [4]: BCD als Operand; [5]: DoubleWord als Operand; [6]: Word als Operand; [7]: QuadWord als Operand; [8]: Zieloperand ist ST(0), R/M bezeichnet Quelloperanden; [9]: Zieloperand wird durch R/M angegeben, Quelloperand ist ST(0); [A]: Operand wird durch R/M angegeben.

Tabelle 5.20: Ein-Byte-FPU-Opcodes

Gruppe	Byte 1	Byte 2	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
A	D9	D_	fnop	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
		E_	fchs	fabs	reserved	reserved	ftst	fxam	reserved	reserved	fld1	fldl2t	fldl2e	fldpi	fldlg2	fldln2	fldz	reserved
		F_	f2xm1	fyl2x	fptan	fpatan	fxtract	fprem1	fdecstp	fincstp	fprem	fyl2xp1	fsqrt	fsincos	frndint	fscale	fsin	fcos
B	DA	E_	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	fucompp	reserved	reserved	reserved	reserved	reserved	reserved
		F_	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
C	DB	E_	feni	fdisi	fclex	finit	reserved	reserved	reserved	reserved								
		F_									reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
D	DE	D_	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	fcompp	reserved	reserved	reserved	reserved	reserved	reserved
E	DF	E_	fstsw ax	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved

In dieser Tabelle sind die Befehle dargestellt, die in Tabelle 5.20 (rechte Seite) als Gruppen-Befehle markiert sind. Diese Befehle haben keinen oder nur implizierte Operanden, sodass das ModR/M-Byte nicht als solches fungiert, sondern als »echtes« Opcode-Byte. Opcodes, die in dieser Tabelle nicht dargestellt sind (z.B. $D8 $xx oder $D9 $Cx) oder deren Eintrag frei bleibt (z.B. $DB $E8 oder $DB $F0), sind Ein-Byte-FPU-Opcodes mit den Opcodes $D8 bis $DF, denen zur Angabe der Operanden ein Mod/RM-Byte folgt. Sie sind in Tabelle 5.20 dargestellt.

Tabelle 5.21: Zwei-Byte-FPU-Opcodes

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_													pi2fw	pi2fd		
1_													pf2iw	pf2id		
2_																
3_																
4_ [B]																
5_																
6_																
7_																
8_ [5]											pfnacc					
9_ [B]	pfcmpge				pfmin		pfrcp	pfrsqrt			pfsub				pfadd	
A_	pfcmpgt				pfmax		pfrcpit1	pfrsqit1			pfsubr				pfacc	
B_	pfcmpeq				pfmul		pfrcpit2	pmulhrw				pswapd				pavgusb
C_																
D_																
E_																
F_																

AMD codiert die 3DNow!-Befehle mittels des Opcodes $0F $0F. Daran schließen sich ggf. Adressen und/oder Konstanten an. Die Unterscheidung der 3D-Now!-Befehle erfolgt dann über ein Suffix-Byte, das in dieser Tabelle dargestellt wird.

Tabelle 5.22: »Drei-Byte«-Opcodes für die 3DNow!-Befehle der AMD-Prozessoren

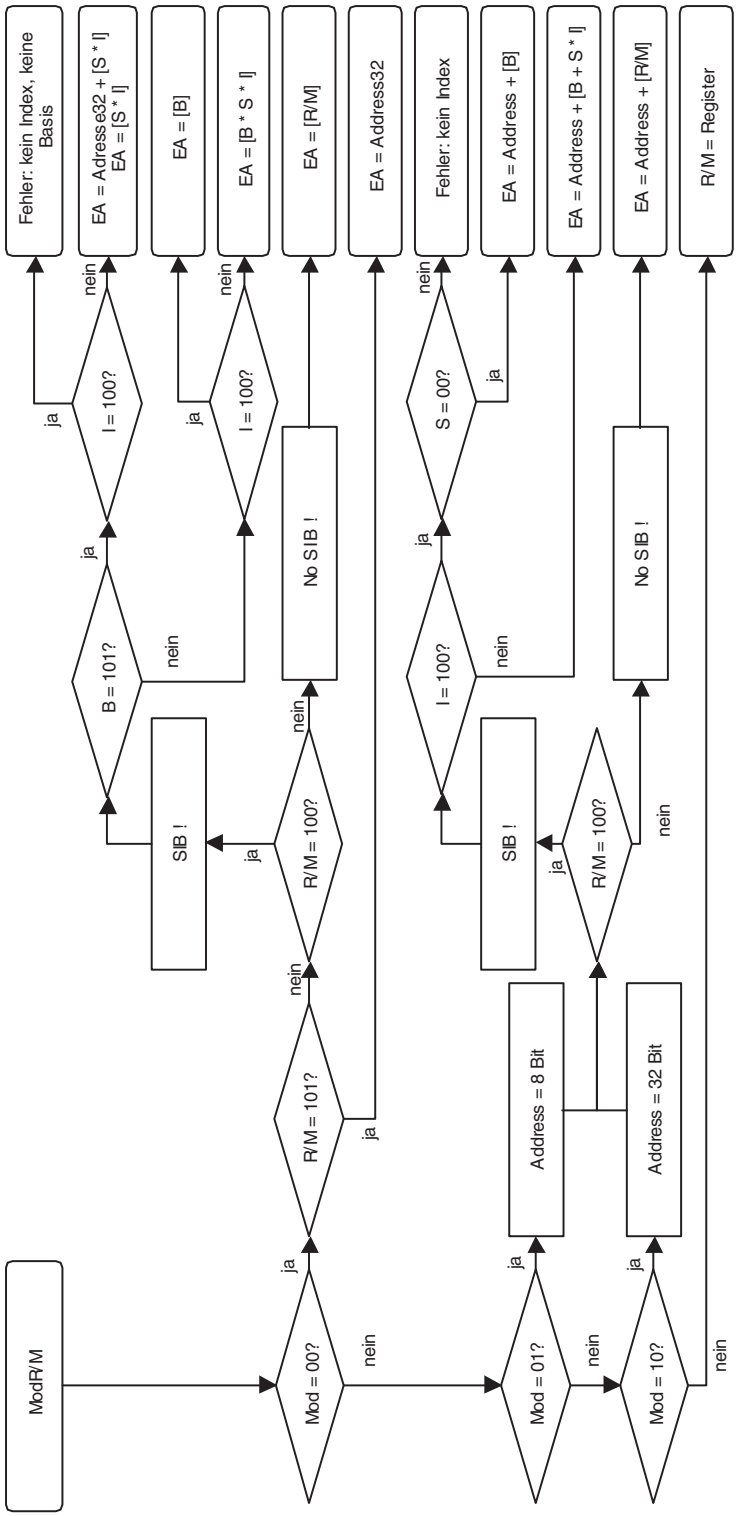


Abbildung 5.36: Flussdiagramm zur Decodierung des ModR/M- und SIB-Bytes einer Befehlssequenz

5.6 Tabellen zur Single-Instruction-Multiple-Data-Technologie (SIMD)

5.6.1 Unter SIMD auf Intel-Prozessoren verfügbare Datenformate

verfügbar unter	Struktur	Größe	Struktur	Struktur	Anzahl	Element	Bytes
MMX, SSE, SSE2 Register: MMX	ShortPacked-Integer	64 Bit 8 Byte	ShortPackedShort	ShortPackedByte	8	Byte	1
				ShortPackedShortInt		ShortInt	1
			ShortPackedSmall	ShortPackedWord	4	Word	2
				ShortPackedSmallInt		SmallInt	2
			ShortPackedLong	ShortPackedDWord	2	DoubleWord	4
				ShortPackedLongInt		LongInt	4
						QuadWord	8
						QuadInt	8
SSE2 Register: XMM	Packed-Integer	128 Bit 16 Byte	PackedShort	PackedByte	16	Byte	1
				PackedShortInt		ShortInt	1
			PackedSmall	PackedWord	8	Word	2
				PackedSmallInt		SmallInt	2
			PackedLong	PackedDWord	4	DoubleWord	4
				PackedLongInt		LongInt	4
			PackedQuad	PackedQWord	2	QuadWord	8
				PackedQInt		QuadInt	8
						OctelWord	16
						OctelInt	16
SSE/ SSE2 Register: XMM	Scalar-Real	32 Bit 4 Byte	ScalarSingle		1	SingleReal	4
SSE2		64 Bit 8 Byte	ScalarDouble		1	DoubleReal	8
SSE/ SSE2	Packed-Real	128 Bit 16 Byte	PackedSingle		4	SingleReal	4
SSE2			PackedDouble		2	DoubleReal	8

Tabelle 5.23: Unter den einzelnen Erweiterungen der SIMD-Technik von Intel (MMX, SSE, SSE2) verfügbare Datenformate

Tabelle 5.24 stellt die in diesem Buch verwendeten Datenbezeichnungen der Intel-Nomenklatur gegenüber.

Intel-Nomenklatur	In diesem Buch verwendete Nomenklatur
64 bit packed byte integers (signed and unsigned)	ShortPackedByte, ShortPackedShortInt
64 bit packed word integers (signed and unsigned)	ShortPackedWord, ShortPackedSmallInt
64 bit packed doubleword integers (signed and unsigned)	ShortPackedDoubleWord, ShortPackedLongInt
128 bit packed byte integers (signed and unsigned)	PackedByte, PackedShortInt
128 bit packed word integers (signed and unsigned)	PackedWord, PackedSmallInt
128 bit packed doubleword integers (signed and unsigned)	PackedDoubleWord, PackedLongInt
128 bit packed quadword integers (signed and unsigned)	PackedQuadWord, PackedQuadInt
128 bit packed single-precision floating-point value	PackedSingle
128 bit scalar single-precision floating-point value	ScalarSingle
128 bit packed double-precision floating-point value	PackedDouble
128 bit scalar double-precision floating-point value	ScalarDouble

Tabelle 5.24: Intel-Nomenklatur für die unter SIMD verwendeten Datenformate und deren in diesem Buch verwendeten Pendants

5.6.2 Unter SIMD auf Intel-Prozessoren verfügbare Instruktionen

Tabelle 5.26 enthält eine Übersicht über die unter den einzelnen Stufen der SIMD-Implementationen verfügbaren SIMD-Operationen. Bitte beachten Sie, dass die Instruktionen in unterschiedlichen Registern durchgeführt werden, je nachdem, mit welchen Datenstrukturen gearbeitet werden soll. Sollen die 128 Bit breiten PackedIntegers bzw. das DoubleQuadWord (OctelWord, OctelInt) und/oder ScalarReals oder PackedReals zum Einsatz kommen, werden die XMM-Register verwendet. Die MMX-Register werden benutzt, falls die 64 Bit breiten ShortPackedIntegers oder QuadWords als Operatoren angegeben werden.

In der folgenden Tabelle 5.25 sind die Instruktionen aufgelistet, mit denen unter SSE2 die Konvertierung einer Realzahl in eine Integer und umgekehrt möglich ist.

Fließkommazahl XMM-Register		Integer		
		CPU-Integer (Allzweckregister)	ShortPackedInteger (MMX-Register)	PackedInteger (XMM-Register)
ScalarSingle	→	CVTSS2SI	-	-
	→	CVTTSS2SI		
	←	CVTSI2SS	-	-
PackedSingle	→	-	CVTPS2PI	CVTPS2DQ
	→		CVTTPS2PI	CVTTPS2DQ
	←	-	CVTPI2PS	CVTDQ2PS
ScalarDouble	→	CVTSD2SI	-	-
	→	CVTTSD2SI		
	←	CVTSI2SD	-	-
PackedDouble	→	-	CVTPD2PI	CVTPD2DQ
	→		CVTTPD2PI	CVTTPD2DQ
	←	-	CVPTPI2PD	CVTDQ2PD

Tabelle 5.25: SSE2-Instruktionen zur Konvertierung von Fließkommazahlen in Integers und umgekehrt

Operation	ShortPackedInteger				PackedInteger				PackedReal			
	wrap-around	unsigned saturation	signed saturation	QuadWord QuadInt	wrap-around	unsigned saturation	signed saturation	OctelWord OctelInt	ScalarSingle	PackedSingle	ScalarDouble	PackedDouble
Arithmetische Operationen												
Addition	PADDB	PADDUSB	PADDSB		PADDB	PADDUSB	PADDSB		ADDSS	ADDPS	ADDSD	ADDPD
	PADDW	PADDUSW	PADDSW		PADDW	PADDUSW	PADDSW					
	PADDD			PADDQ	PADDD							
					PADDQ							
Subtraktion	PSUBB	PSUBUSB	PSUBSB		PSUBB	PSUBUSB	PSUBSB		SUBSS	SUBPS	SUBSD	SUBPD
	PSUBW	PSUBUSW	PSUBSW		PSUBW	PSUBUSW	PSUBSW					
	PSUBD			PSUBQ	PSUBD							
					PSUBQ							
Multiplikation	PMULHW				PMULHW				MULSS	MULPS	MULDS	MULPD
	PMULLW				PMULLW							
	PMULHUW				PMULHUW							
	PMULUDQ				PMULUDQ							
Mult. + Add.	PMADDWD				PMADDWD							
Division									DIVSS	DIVPS	DIVSD	DIVPD
Reziprokwert									RCPSS	RCPPS		
Wurzel									SQRTSS	SQRTPS	SQRTSD	SQRTPD
rezip. Wurzel									RSQRTSS	RSQRTPS		
Maximum	PMAXUB				PMAXUB				MAXSS	MAXPS	MAXSD	MAXPD
	PMAXSW				PMAXSW							
Minimum	PMINUB				PMINUB				MINSS	MINPS	MINSD	MINPD
	PMINSW				PMINSW							
Mittelwert	PAVGB				PAVGB							
	PAVGW				PAVGW							
S. A. D.	PSADBW				PSADBW							

Nicht unterlegte Befehle wurden im Rahmen der MMX-Technologie implementiert, hellgrau unterlegte mit den SSE- und dunkelgrau unterlegte mit den SEE2-Erweiterungen.

Tabelle 5.26: Übersicht der unter SIMD der Intelprozessoren verfügbaren Befehle

Operation	ShortPackedInteger				PackedInteger				PackedReal			
	wrap-around	unsigned saturation	signed saturation	QuadWord QuadInt	wrap-around	unsigned saturation	signed saturation	OctelWord OctelInt	ScalarSingle	PackedSingle	ScalarDouble	PackedDouble
Logische Operationen												
AND				PAND				PAND		ANDPS		ANDPD
AND NOT				PANDN				PANDN		ANDNPS		ANDNPD
OR				POR				POR		ORPS		ORPD
XOR				PXOR				PXOR		XORPS		XORPD
Shift-Operationen												
logisch links	PSLLW			PSLLQ	PSLLW			PSLLDQ				
	PSLLD				PSLLD							
					PSLLQ							
logisch rechts	PSRLW			PSRLQ	PSRLW			PSRLDQ				
	PSRLD				PSRLD							
					PSRLQ							
arithm. links	PSRAW				PSRAW							
	PSRAD				PSRAD							
Vergleichs-Operationen												
»prospektiv«	PCMPEQB				PCMPEQB				CMPSS	CMPPS	CMPSD	CMPPD
	PCMPEQW				PCMPEQW							
	PCMPEQD				PCMPEQD							
	PCMPGTB				PCMPGTB							
	PCMPGTW				PCMPGTW							
	PCMPGTD				PCMPGTD							
»retrospektiv«									COMISS		COMISD	
									UCOMISS		UCOMISD	

Nicht unterlegte Befehle wurden im Rahmen der MMX-Technologie implementiert, hellgrau unterlegte mit den SSE- und dunkelgrau unterlegte mit den SEE2-Erweiterungen.

Tabelle 5.26: Übersicht der unter SIMD der Intelprozessoren verfügbaren Befehle (Forts.)

Operation	ShortPackedInteger				PackedInteger				PackedReal			
	wrap-around	unsigned saturation	signed saturation	QuadWord QuadInt	wrap-around	unsigned saturation	signed saturation	OctelWord OctelInt	ScalarSingle	PackedSingle	ScalarDouble	PackedDouble
Datenkonvertierung												
Pack		PACKUSWB	PACKSSWB			PACKUSWB	PACKSSWB					
			PACKSSWD				PACKSSWD					
Unpack High	PUNPCKHBW				PUNPCKHBW					UNPCKHPS		UNPCKHPD
	PUNPCKHWD				PUNPCKHWD							
	PUNPCKHDQ				PUNPCKHDQ							
UnpackLow	PUNPCKLBW				PUNPCKLBW					UNPCKLPS		UNPCKLPD
	PUNPCKLWD				PUNPCKLWD							
	PUNPCKLDQ				PUNPCKLDQ							
Real ↔ Real									CVTSS2SD	CVTPS2PD	CVTSD2SS	CVTPD2PS
Real ↔ Int.									siehe Tabelle 5.25 auf Seite 846			
Datenaustausch												
Mem↔ Reg	MOVD			MOVQ	MOVD			MOVDQA	MOVSS	MOVAPS	MOVSD	MOVAPD
					MOVQ			MOVDQU		MOVUPS		MOVUPD
										MOVHPS		MOVHPD
										MOVLPS		MOVLPD
non-temp.				MOVNTQ				MOVNTDQ		MOVNTPS		MOVNTPD
Mem ↔Mem	PEXTRW				PEXTRW					MOVHLPS		
	PINSRW				PINSRW					MOVLHPS		
	PMOVMSKB			MASKMOVQ	PMOVMASKB			MASKMOVDQ		MOVMSKPS		MOVMSKPD
MMX ↔ XMM				MOVQ2DQ				MOVDQ2Q				
Shuffle	PSHUFW				PSHUFLW					SHUFPS		SHUFPD
					PSHUFHW							
					PSHUFD							

Nicht unterlegte Befehle wurden im Rahmen der MMX-Technologie implementiert, hellgrau unterlegte mit den SSE- und dunkelgrau unterlegte mit den SEE2-Erweiterungen.

Tabelle 5.26: Übersicht der unter SIMD der Intelprozessoren verfügbaren Befehle (Forts.)

5.6.3 Unter SIMD auf AMD-Prozessoren verfügbare Datenformate

Die 3DNow!-Technologie von AMD benutzt die MMX- und damit die FPU-Register zur Berechnung von Fließkommazahlen. Aber Vorsicht! Die Tatsache, dass in den FPU-Registern und mit gepackten Fließkommazahlen gearbeitet wird, heißt *nicht*, dass auch die FPU-Befehle zum Einsatz kämen oder die gepackten Fließkommazahlen 80 Bit Breite hätten. Vielmehr müssen Sie sich die Situation so vorstellen, dass auch mit den gepackten Fließkommazahlen in einer MMX-Umgebung mit den gleichen Bedingungen wie unter MMX gearbeitet wird. Es werden lediglich die 64 Bits der Register als Fließkommazahlen interpretiert, die mit den neuen 3DNOW!-Instruktionen verarbeitet werden.

verfügbar unter	Struktur	Größe	Struktur	Struktur	An-zahl	Element	Bytes
3DNow!, 3DNow-X, MMX Register: MMX	ShortPacked-Integer	64 Bit 8 Byte	ShortPackedShort	ShortPackedByte	8	Byte	1
				ShortPackedShortInt		ShortInt	1
			ShortPackedSmall	ShortPackedWord	4	Word	2
				ShortPackedSmallInt		SmallInt	2
			ShortPackedLong	ShortPackedDWord	2	DoubleWord	4
				ShortPackedLongInt		LongInt	4
						QuadWord	8
						QuadInt	8
	Short-PackedReal	64 Bit 8 Byte	ShortPackedSingle		4	SingleReal	4

Tabelle 5.27: Unter der 3DNow!-Technik verfügbare Datenformate

Da unter 3DNow! auch mit den beim Athlon eingeführten Erweiterungen die Register »nur« 64 Bit breit sind, können maximal zwei Fließkommazahlen mit jeweils vier Bytes Umfang, also zwei SingleReals gleichzeitig bearbeitet werden. Aus diesem Grunde wurden die Datenformate »ShortPackedReal« und »ShortPackedSingle« genannt, um den Unterschied zu den unter der SIMD-Technik von Intel verfügbaren »normalen« gepackten Realzahlen mit 128 Bit Breite darzustellen.

In Tabelle 5.28 sind die in diesem Buch verwendeten Datenbezeichnungen der AMD-Nomenklatur gegenübergestellt:

AMD-Nomenklatur	In diesem Buch verwendete Nomenklatur
packed byte integers (signed and unsigned)	ShortPackedByte, ShortPackedShortInt
packed word integers (signed and unsigned)	ShortPackedWord, ShortPackedSmallInt
packed doubleword integers (signed and unsigned)	ShortPackedDoubleWord, ShortPackedLongInt
two packed single-precision floating-point doublewords	ShortPackedSingle
3DNOW!-Data-Type	ShortPackedSingle

Tabelle 5.28: AMD-Nomenklatur für die unter SIMD verwendeten Datenformate und deren in diesem Buch verwendeten Pendants

5.6.4 Unter SIMD auf AMD-Prozessoren verfügbare Instruktionen

Tabelle 5.29 enthält eine Übersicht über die unter den einzelnen Stufen der SIMD-Implementationen verfügbaren SIMD-Operationen.

Operation	ShortPackedInteger				PackedInteger				PackedReal	
	wrap-around	unsigned saturation	signed saturation	QuadWord QuadInt	wrap-around	unsigned saturation	signed saturation	OctelWord OctelInt	ShortPackedSingle	ShortPackedDouble
Arithmetische Operationen										
Addition	PADDB	PADDUSB	PADDSB						PFADD	
	PADDW	PADDUSW	PADDSW							
	PADDD									
Subtraktion	PSUBB	PSUBUSB	PSUBSB						PFSUB	
	PSUBW	PSUBUSW	PSUBSW						PFSUBR	
	PSUBD									
Multiplikation	PMULHW								PFMUL	
	PMULLW									
	PMULHRW	PMULHUW								
Mult. + Add.	PMADDWD									
Akkumulation						nicht verfügbar			PFACC	PFNACC
										PFPNACC
Reziprokwert									PFRCP	PFRCPIT1
										PFRCPIT2
rezip. Wurzel									PFRSQRT	PFRSQIT1
Maximum	PMAXUB									
	PMAXSW									
Minimum	PMINUB									
	PMINSW									
Mittelwert	PAVGB	PAVGUSB								
	PAVGW									
S. A. D.	PSADBW									

Nicht unterlegte Befehle wurden im Rahmen der MMX-Technologie implementiert, hellgrau unterlegte mit den 3DNow!- und dunkelgrau unterlegte mit den 3DNow!-X-Erweiterungen.

Tabelle 5.29: Übersicht der unter SIMD der AMD-Prozessoren verfügbaren Befehle

Operation	ShortPackedInteger				PackedInteger				PackedReal	
	wrap-around	unsigned saturation	signed saturation	QuadWord QuadInt	wrap-around	unsigned saturation	signed saturation	OctelWord OctelInt	ShortPackedSingle	ShortPackedDouble
Logische Operationen										
AND				PAND						
AND NOT				PANDN						
OR				POR						
XOR				PXOR						
Shift-Operationen										
logisch links	PSLLW			PSLLQ						
	PSLLD									
logisch rechts	PSRLW			PSRLQ						
	PSRLD									
arithm. links	PSRAW									
	PSRAD									
Vergleichs-Operationen										
»prospektiv«	PCMPEQB								PFCMPEQ	
	PCMPEQW									
	PCMPEQD									
	PCMPGTB								PFCMPGT	
	PCMPGTW									
	PCMPGTD									
									PFCMPGE	

nicht verfügbar (PackedInteger)

nicht verfügbar (PackedReal)

nicht verfügbar (Shift-Operationen)

nicht verfügbar (Vergleichs-Operationen)

Nicht unterlegte Befehle wurden im Rahmen der MMX-Technologie implementiert, hellgrau unterlegte mit den 3DNow!- und dunkelgrau unterlegte mit den 3DNow!-X-Erweiterungen.

Tabelle 5.29: Übersicht der unter SIMD der AMD-Prozessoren verfügbaren Befehle (Forts.)

Operation	ShortPackedInteger				PackedInteger				PackedReal	
	wrap-around	unsigned saturation	signed saturation	QuadWord QuadInt	wrap-around	unsigned saturation	signed saturation	OctelWord OctelInt	ShortPackedSingle	ShortPackedDouble
Datenkonvertierung										
Pack		PACKUSWB	PACKSSWB							
			PACKSSWD							
Unpack High	PUNPCKHBW									
	PUNPCKHWD									
	PUNPCKHDQ									
UnpackLow	PUNPCKLBW									
	PUNPCKLWD									
	PUNPCKLDQ									
Real ↔ Int.									PI2FD	PI2FW
									PF2ID	PF2IW
Datenaustausch										
Mem↔ Reg	MOVD			MOVQ						
non-temp.				MOVNTQ						
Mem ↔Mem	PEXTRW								PSWAPD	
	PINSRW									
	PMOVMASKB			MASKMOVQ						
Shuffle	PSHUFW									

nicht verfügbar

nicht verfügbar

nicht verfügbar

nicht verfügbar

Nicht unterlegte Befehle wurden im Rahmen der MMX-Technologie implementiert, hellgrau unterlegte mit den 3DNow!- und dunkelgrau unterlegte mit den 3DNow!-X-Erweiterungen.

Tabelle 5.29: Übersicht der unter SIMD der AMD-Prozessoren verfügbaren Befehle (Forts.)

5.6.5 Entsprechungen und Unterschiede der Intel- und AMD-SIMD-Befehle

In der folgenden Tabelle 5.30 werden die verfügbaren 3DNow!-X-Befehle den analogen SSE-/SSE2-Instruktionen gegenübergestellt:

3DNow!-Instruktion	analoge SSE-/SSE2-Instruktion	3DNow!-Instruktion	analoge SSE-/SSE2-Instruktion
PAVGUSB	nicht verfügbar	PFRCP	andere Lösung: RCPPS
PF2ID	CVTTSS2SI	PFRCPIT1	nicht notwendig
PF2IW	nicht verfügbar	PFRCPIT2	nicht notwendig
PFACC,PFNACC, PFPNACC	nicht verfügbar	PFRSQRIT1	andere Lösung: RSQRTPS
PFADD	ADDPS	PFRSQRT	nicht notwendig
PFCMPEQ	andere Lösung: CMPPS	PFSUB	SUBPS
PFCMPGE	andere Lösung: CMPPS	PFSUBR	nicht verfügbar
PFCMPGT	andere Lösung: CMPPS	PI2FD	CVTPI2PS
PFMAX	MAXPS	PI2FW	nicht verfügbar
PFMIN	MINPS	PMULHRW	PMULHW
PFMUL	MULPS	PSWAPD	andere Lösung: PSHUFW

Tabelle 5.30: Der 3DNow!-Technologie von AMD vergleichbare Intel-Instruktionen unter SSE/SSE2

Es ergeben sich zwei Inkompatibilitäten der MMX-Befehlssätze unter 3DNow!-X bzw. SSE2:

SSE2	3DNow!-X	Bemerkungen
PMULUDQ		Dieser Befehl multipliziert zwei *unsigned doublewords* und legt das resultierende *quadword* im Register ab. Er wurde mit den SSE2-Erweiterungen eingeführt und steht sowohl als 64-Bit-Version für die MMX-Register als auch als 128-Bit-Version für die XMM-Register zur Verfügung. Er ist im *instruction set* von AMD-Prozessoren nicht vorhanden.
	PMULHRW	Dieser Befehl multipliziert zwei ShortPackedWords und legt die jeweils resultierenden höherwertigen Worte der entstehenden *doublewords* nach Rundung als ShortPackedWord ab. Er ist damit eine Variante des in beiden Befehlssätzen vorhandenen Befehls PMULHW, ist aber im *instruction set* von Intel-Prozessoren nicht vorhanden

Tabelle 5.31: Unterschiede in den instruction sets der SIMD-Befehle von AMD und Intel

5.7 Weitere Register der CPU

Neben den bereits auf Seite 31 dargestellten »Basisregistern« der CPU sowie den auf Seite 188 vorgestellten Registern der FPU und den Registern, die bei SIMD erforderlich sind (XMM-Register und MXCSR, vgl. Seite 309 und 361), gibt es noch drei große Klassen von Registern mit wesentlicher Bedeutung:

- Kontrollregister
- Debugregister
- modellspezifische Register

Bis auf die modellspezifischen Register, die, wie der Name schon sagt, bei jedem Prozessor unterschiedlich ausfallen, werden im Folgenden die zusätzlichen Register kurz vorgestellt.

5.7.1 Kontroll-Register

Mit der Einführung des protected mode beim 80286 wurde es notwendig, Register zu schaffen, die diesen neuen Betriebsmodus kontrollieren konnten. Beim 80286 war dies noch ein als machine status word bezeichnetes 16-Bit-Register, das jedoch schnell durch das seitdem bestehende Kontrollregister CR0 abgelöst oder besser erweitert wurde. Diesem und dem reservierten Register CR1 wurden mit Einführung des Paging-Mechanismus als Teil der virtuellen Speicherverwaltung die Kontrollregister CR2 und CR3 zur Seite gestellt. Kontrollregister CR4 schließlich wurde geschaffen, um weitere, neue Fähigkeiten zu kontrollieren, wie z.B. die Unterstützung des virtual 8086 mode, SIMD oder die Fähigkeit, physikalische Adressen jenseits der »magischen« 4-GByte-Grenze anzusprechen.

CR0 Das Kontrollregister CR0 ist in Abbildung 5.37 dargestellt.

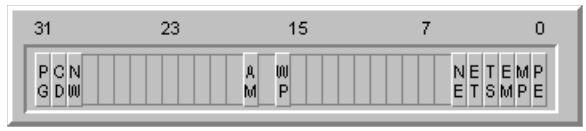


Abbildung 5.37: Speicherabbild des Kontrollregisters 0

Im Folgenden wird der Processor, bei dem das Flag eingeführt wurde, in Klammern angegeben. Es bedeuten:

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 co-processor* (80386); siehe TS-Flag

EM *emulation* (80386); im gesetzten Zustand signalisiert dieses Flag, dass weder eine interne FPU (80486DX, Pentium, P6-Familie, Pentium 4) noch eine externe NPX (80386, 80486SX) zur Verfügung steht und FPU-Befehle emuliert werden müssen. Im gelöschten Zustand ist eine FPU/NPX 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- und XMM-Befehle Einfluss. Ist EM gesetzt, so führt jede Ausführung eines MMX- oder XMM-Befehls (außer PAUSE, PREFETCHx, SFENCE, LFENCE, MFENCE, MOVNTI und CLFLUSH) zu einer #UD-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, zeitaufwändige 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 flag) dieses Flag wieder zurücksetzen, da es von niemand anderem verändert wird.

ET *extension type* (80386); zeigt im gesetzten Zustand an, dass die 80387-FPU-Instruktionen auf 80386- und 80486-Systemen verwendet werden können. Bei Pentium-, P6- und Pentium-4-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. Im gesetzten Zustand wird der (interne) »native« Mechanismus benutzt, im gelöschten der »PC-typische« Mechanismus, bei dem verschiedene Prozessor-Pins und eine externe Logik eingesetzt werden. Zu Einzelheiten verweise ich auf Sekundärliteratur.

WP *write protect* (80486); ist dieses Flag gesetzt, so verhindert es einen schreibenden Zugriff von Prozeduren im supervisor level auf read-only pages im user level. 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, P6 und Pentium 4) bzw. den Write-Through-Mechanismus (80486) ein, mit dem bei Schreibzugriffen auf den Cache dieser geleert und invalidiert wird. NW arbeitet mit CD zusammen. Zu Einzelheiten verweise ich auf Sekundärliteratur.

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. Zu Einzelheiten verweise ich auf Sekundärliteratur.

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. Ist PG gelöscht, werden alle linearen (»virtuellen«) Adressen als physikalische Adressen behandelt.

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/XMM-Befehl**
0	0	0	ausführen	ausführen	#UD-Exception
0	0	1	ausführen	#NM-Exception	#UD-Exception
0	1	0	ausführen	ausführen	ausführen
0	1	1	#NM-Exception	#NM-Exception	#NM-Exception
1	0	0	ausführen	#NM-Exception	#UD-Exception
1	0	1	ausführen	#NM-Exception	#UD-Exception
1	1	0	ausführen	#NM-Exception	#UD-Exception
1	1	1	#NM-Exception	#NM-Exception	#UD-Exception

Kontrollregister CR1 ist, der Form halber, in Abbildung 5.38 dargestellt. Dieses Register gilt als reserviert. **CR1**



Abbildung 5.38: Speicherabbild des Kontrollregisters 1

Das in Abbildung 5.39 dargestellte Kontrollregister CR2 nimmt im Falle einer page fault 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 Vvirtuelle 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.) **CR2**

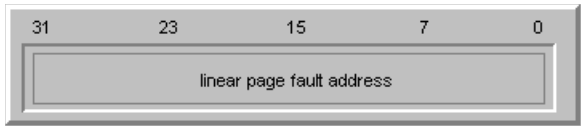


Abbildung 5.39: Speicherabbild des Kontrollregisters 2

CR3 (PDBR) Das Kontrollregister CR3 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. Realiter erfolgt das, indem der Inhalt von CR3 mit der Maske $FFFFF000 UND-verknüpft wird. Ferner beinhaltet dieses Register noch einige Verwaltungsflags, die beim Umgang mit der durch die Adresse referenzierten Tabelle eine Rolle spielen. Abbildung 5.40 zeigt dieses Register:



Abbildung 5.40: Speicherabbild des Kontrollregisters 3 (PDBR)

Es bedeuten:

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 PWT-Flag.

CR4 Das Kontrollregister CR4 (siehe Abbildung 5.41) ist das zweite und neueste Register für Systemflags. Es beinhaltet die Flags, die spezifische Funktionen der Prozessoren ab dem Pentium steuern.

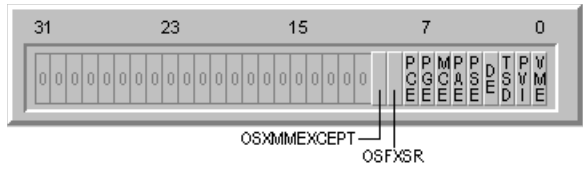


Abbildung 5.41: Speicherabbild des Kontrollregisters 4

Es bedeuten:

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 Flags 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.



Die Control Register sind modellspezifisch (nicht zu Verwechseln mit den MSR, den *model specific registers*, die natürlich auch modellspezifisch sind). Das heißt, dass nicht alle Features in einzelnen Modellen realisiert sein müssen. Konkretes Beispiel: SSE ist erst seit dem Pentium III implementiert, SSE2 erst ab dem Pentium 4. Das bedeutet, dass verschiedene Flags in CR4 eventuell gar nicht definiert sind, die entsprechenden Bitstellungen also falsche Ergebnisse liefern können (wenn z.B. Bit 9 und 10 in CR4 gelöscht sind – heißt das nun, dass OSXMMEXCEPT und/oder OSFXSR nicht unterstützt werden, oder sind sie, weil z.B. ein Pentium Pro vorliegt, gar nicht definiert und damit reserviert?)

Aus diesem Dilemma hilft wieder einmal der CPUID-Befehl. Ruft man ihn mit EAX = $00000001 auf, liefert er, wie bekannt, in EDX die feature flags zurück. Hier haben, bis auf PCE, alle Flags aus CR4 ihr Pendant:

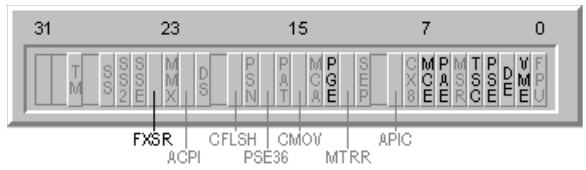


Abbildung 5.42: CR4-Pendants in den feature flags des CPUID-Befehls

So werden die CR4-Flags VME und PVI unterstützt, wenn das VME-Flag der feature flags gesetzt ist. Analoges gilt für TSD (heißt in den feature flags TSC!), DE, PSE, PAE, MCE und PGE. Ein in den feature bits gesetztes FXSR-Bit heißt nicht nur, dass die Befehle FXSAVE/FXRSTOR implementiert sind, sondern auch, dass es die Flags OSXMMEXCEPT und OSFXSR in CR4 gibt, die Auskunft darüber geben, ob das Betriebssystem FXSAVE/FXRSTOR unterstützt.

5.7.2 Debug-Register

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.

DR0 bis DR3

Die Debugregister DR0 bis DR3 (siehe Abbildung 5.43) 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.

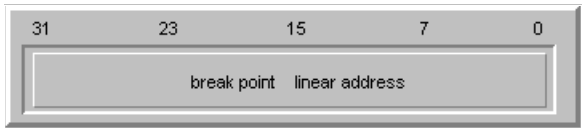


Abbildung 5.43: Speicherabbild der Debugregister 0 bis 3

DR4 und DR5

Die in Abbildung 5.44 dargestellten Debugregister 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.



Abbildung 5.44: Speicherabbild der Debugregister 4 und 5

DR6

Das *Debug-Status-Register* DR6, das Abbildung 5.45 zeigt, 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.

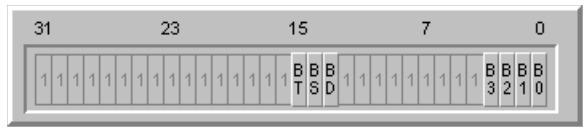


Abbildung 5.45: Speicherabbild des Debug-Registers 6

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-Kontroll-Register* DR7 in den Feldern *LENx* und *R/Wx* für jeden Breakpoint definiert wird, eingetreten ist, als die break point exception #BP 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-Modus 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-Modus 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.



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 zurückzuset-

zen, wenn der Grund für die Exception festgestellt worden ist. Andernfalls kann es zu falschen Reaktionen kommen.

Über das *Debug-Kontroll-Register* können die Bedingungen festgelegt werden, unter denen ein Breakpoint, dessen Adresse in eines der Breakpoint-Address-Register DR0 bis DR3 eingetragen wurde, eine Exception auslöst. **DR7**

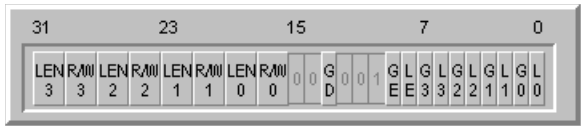


Abbildung 5.46: Speicherabbild des Debug-Registers 7

Diese Bedingungen können sein:

L0-L3 *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 break point exception #BP 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-G3 *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 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 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.

Bei gesetztem DE-Flag interpretiert der Prozessor die Bits wie folgt:

00_b break on instruction execution only

01_b break on data writes only

10_b break on I/O writes or reads

11_b break on data writes or reads but not on instruction fetches

Bei gelöschtem DE-Flag dagegen interpretiert der Prozessor die Bits in der Weise, wie 80386er oder 80486er das tun:

00_b break on instruction execution only

01_b break on data writes only

10_b undefined

11_b break on data writes or reads but not on instruction fetches

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_b Ein-Byte-Daten

01_b Zwei-Byte-Daten

10_b undefined

11_b Vier-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_b ist, so müssen die Daten entsprechend ihrer Größe im Speicher ausgerichtet sein: bei LENx = 01_b an Wortgrenzen, bei LENx = 11 an Doppelwortgrenzen.

5.7.3 Modellspezifische Register (MSRs)

Über die bereits besprochenen Register hinaus verfügen Prozessoren ab dem Pentium auch noch über so genannte modellspezifische Register (model specific register, MSR). Wie der Name bereits ausdrückt, ist ihre Existenz, Funktion, Zahl und Adresse abhängig vom aktuellen Modell. Sie können mit den Befehlen RDMSR (read MSR) und WRMSR (write MSR) ausgelesen bzw. beschrieben werden.



Beachten Sie, dass WRMSR ein privilegierter Befehl ist, der nur unter CPL = 0 verwendet werden kann! MSRs können somit nicht aus der Anwenderebene (CPL = 3) heraus verändert werden!

Die detaillierte Besprechung dieser Register erfolgt hier nicht! Der Grund dafür ist, dass praktisch für jede Prozessor-Familie der Satz an MSRs vorgestellt werden müsste. Zwar sind einige von ihnen »IA-32-architectural«, was bedeutet, dass sie in allen Prozessoren mit IA-32-Architektur ab dem Pentium vorzufinden sind. Doch ist ihre Menge insgesamt so groß, dass ein weiteres Buch bedenkenlos mit den Informationen zu füllen ist, die wichtig wären.

Ferner nutzt auch die Besprechung dieser Register nicht viel. Soweit wesentliche oder interessante Möglichkeiten mit einem Register verbunden sind, gibt es Instruktionen, die genutzt werden können, so z.B. RDTSC, das das modellspezifische Register ausliest, das den time stamp counter enthält. Ein direkter Zugriff auf das MSR an Adresse $0010, das diesen Counter enthält, ist also nicht zwingend erforderlich. Und die anderen Register sind im Zweifel zugriffsgeschützt, da sie in irgendeiner Weise mit Schutzkonzepten in Konflikt geraten könnten (siehe SYSENTER/SYSEXIT und das SYSENTER_CS-Register und seine Kollegen).

Nicht zuletzt hat der privilegierte Befehl WRMSR einer Veränderung der Register einen Riegel vorgeschoben. Und das bloße Auslesen von Registerinhalten ist auch langweilig! Daher verweise ich auf Sekundärliteratur, falls Sie Informationen zu diesem Themenkreis benötigen.

5.8 FPU-, MMX- und XMM-Umgebung

Je nachdem, welche Daten Sie aus den FPU-, MMX- und/oder XMM-Registern sichern oder restaurieren möchten, haben Sie mehrere Möglichkeiten:

- vollständige Sicherung/Restauration aller Daten, unabhängig ob FPU-, MMX- oder XMM-Daten
- Sicherung und Restauration aller FPU-Daten (ohne MMX bzw. XMM)
- Sicherung und Restauration der »FPU-Umgebung« oder
- Sicherung und Restauration des FPU-Status.

FXSAVE FXRSTOR

Zur vollständigen Sicherung/Restauration aller Daten gibt es das Befehlspaar FXSAVE – FXRSTOR, das mit einer in Abbildung 5.47 dargestellten Datenstruktur arbeitet.

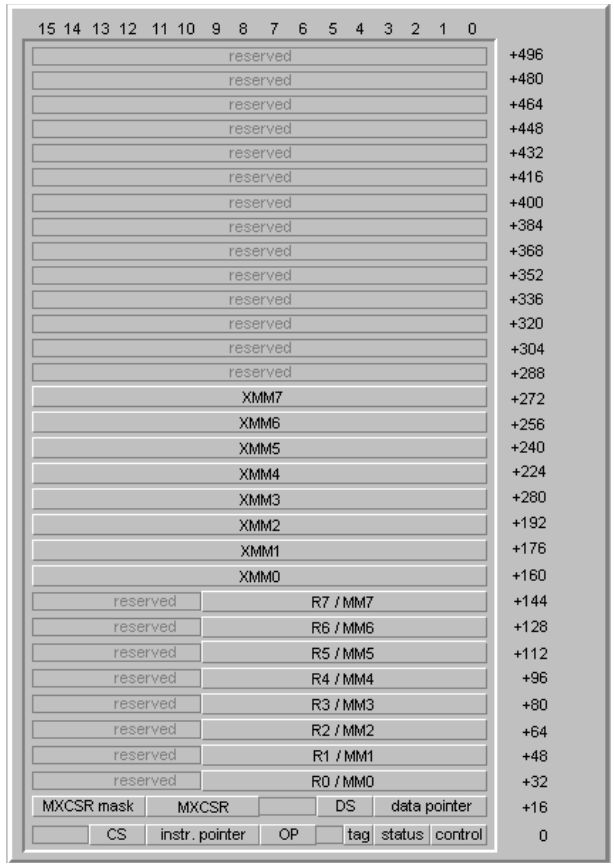


Abbildung 5.47: Speicherabbild der durch FXSAVE/FXRSTOR verwalteten Daten

Die Datenstruktur ist unabhängig vom Betriebsmodus (real mode oder protected mode) und der Betriebsumgebung (16- bzw. 32-Bit-Umgebung).



Bitte beachten Sie, dass die Felder *MXCSR mask* und *MXCSR* sowie die Speicherbereiche, die die Inhalte der XMM-Register aufnehmen, erst ab dem Pentium III mit seinen SSE-Implementierungen definiert sind. Unter MMX, mit dem die Befehle FXSAVE und FXRSTOR eingeführt wurden, gelten diese Bereiche als reserviert.

Wie man aus der Abbildung ersehen kann, sind noch weite Bereiche dieser Struktur reserviert und bieten Möglichkeiten für Erweiterungen in der Zukunft. Wie man ebenfalls feststellen kann, werden die Inhalte aller XMM-Register in die Struktur aufgenommen sowie aller Register, die von der FPU als FPU-Datenregister (FPU-Stack) oder von der CPU als MMX-Datenregister verwendet werden. Darüber hinaus werden folgende FPU-Registerinhalte berücksichtigt: control word, status word und tag register, die Inhalte des Opcode-Registers sowie die far pointer (16-Bit-Selektor und 32-Bit-Offset) aus den Registern *last instruction* und *last data*. Von Seiten XMM sind noch das MXCSR und die dazugehörige Maske zu berücksichtigen.



Bitte beachten Sie, dass das Tag-Feld nicht die gleiche Information enthält wie das Tag-Register der FPU. Das hat seine Gründe darin, dass bei MMX-Daten ja lediglich verzeichnet wird, ob das Register »empty« ist oder nicht. Dazu ist aber nur ein Bit erforderlich. Somit ist im Tag-Feld der Datenstruktur dann und nur dann ein Bit gelöscht, wenn das korrespondierende MMX-Register leer ist (das korrespondierende Tag-Feld im Tag-Register also den Wert 11_b hat. Ein gesetztes Bit signalisiert dann, dass das korrespondierende MMX-Register ein gültiges Datum enthält. Die Zuordnung ist einfach: MMX-Register 0 wird durch Bit 0 repräsentiert, MMX7 von Bit 7.

Diese »Umcodierung« wird von FXSAVE vorgenommen. Es unterscheidet nicht zwischen »special« (10_b), »zero« (01_b) und »valid« (00_b) und setzt das entsprechende Bit in jedem der drei Fälle. Bei »empty« (11_b) wird es gelöscht.

Umgekehrt restauriert FXRSTOR aus diesen Tag-Bits einerseits und dem jeweiligen Registerinhalt wieder das »korrekte« Tag-Word. So wird das zum jeweiligen Register gehörige Tag-Feld im Tag-Word auf den Wert 11_b gesetzt, wenn das korrespondierende Tag-Bit der Daten-

struktur gelöscht ist. Ist es dagegen gesetzt, so wird der Registerinhalt nach Art von FXAM geprüft. Stellt die Zahl den Wert 0 dar, erhält das Tag-Feld den Inhalt 01_b, stellt es eine NaN oder ein nicht unterstütztes Datenformat dar, den Inhalt 10_b. Andernfalls wird 00_b eingetragen.



Wie Sie sehen können, befindet sich neben dem Inhalt des MXCSR auch eine MXCSR-Maske in der Datenstruktur. In sie trägt die CPU eine Maske ein, in der jedes Bit gelöscht ist, das im MXCS-Register reserviert ist. Ein in der Maske gesetztes Bit signalisiert somit ein nicht-reserviertes MXCSR-Bit. Ausnahme: Enthält die Maske den Wert »0«, so gilt die Default-Bit-Einstellung $0000FFBF.

Wozu das Ganze? Die Datenstruktur kann ja nicht nur dazu benutzt werden, bei task switches die FPU-, MMX- und XMM-Umgebung zu sichern! Vielmehr wird sie durchaus auch zu Analysen herangezogen. Und da das (versehentliche oder absichtliche) Setzen von reservierten Bits im MXCSR eine general protection exceptions nach sich zieht, tut man gut daran, solche Bits unangetastet zu lassen. Doch welche sind das? Im Standard-Pentium-4 sind es die Bits, die die Maske $0000FFBF bilden. Daher ist in diesem Fall auch die MXCSR-Maske mit »0« vorbelegt. In folgenden Prozessoren oder Prozessor-Familien dagegen könnten einige der heute noch reservierten Bits »legal« sein. Und das sollte der Programmierer dann wissen. Auskunft darüber erhält er – von der MXCSR-Maske im Speicherabbild der von FXSAVE/FXRESTOR verwendeten Datenstruktur.



Betrachten Sie daher das Word *MXCSR-Maske* als Möglichkeit, genauere Auskunft über verschiedene Features des Prozessors zu gewinnen, wenn es um das Thema XMM geht – ganz analog zu CPUID. Beachten Sie aber auch, dass Sie die Datenstruktur, die Sie FXSAVE als Operanden übergeben, ausrichten!

F(N)SAVE FRSTOR

Im Gegensatz zu FXSAVE/FXRSTOR benutzen die Befehle F(N)SAVE/FRSTOR sehr wohl Datenstrukturen, die vom Betriebsmodus (real mode) und der Umgebung (protected mode) abhängig sind. F(N)SAVE/FRSTOR sind die »Vorgänger« von FXSAVE/FXRSTOR und wurden zu einer Zeit implementiert, als die FPU alleiniger Herrscher über ihre Register war und an SIMD nicht im entferntesten zu denken war. Daher betreffen F(N)SAVE/FRSTOR und die verwendeten Datenstrukturen auch lediglich die FPU und ihre Register.

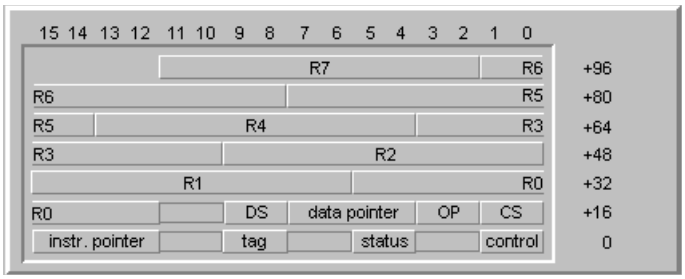


Abbildung 5.48: Speicherabbild der durch F(N)SAVE/FRSTOR im 32-Bit-Protected-Mode verwalteten Daten

Abbildung 5.48 zeigt die Datenstruktur, die die Befehle im 32-Bit-Protected-Mode benutzen. Die Darstellung wurde absichtlich in der realisierten Weise gewählt, um einen direkten Vergleich mit Abbildung 5.47 zu ermöglichen, der unmittelbar die Inkompatibilität der von F(N)SAVE/FRSTOR und FXSAVE/FXRSTOR verwendeten Datenstrukturen veranschaulicht.

Die Inhalte der Struktur geben keine Rätsel auf: Es handelt sich um die Inhalte des control registers, des status registers und des tag registers sowie der Register für den last instruction pointer, den last data pointer und den Opcode. Abgeschlossen wird die 108-Byte-Struktur durch die Inhalte der acht FPU-Register.

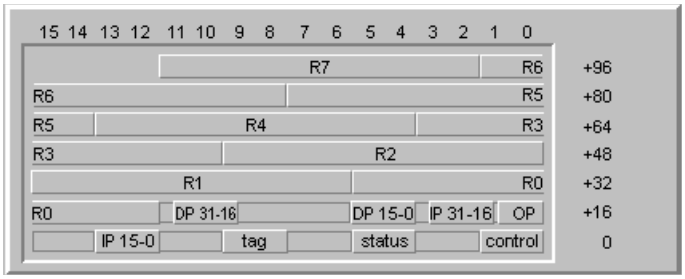


Abbildung 5.49: Speicherabbild der durch F(N)SAVE/FRSTOR im 32-Bit-Real-Mode verwalteten Daten

Auch im 32-Bit-Real-Mode wird, wie Abbildung 5.49 zeigt, eine 108-Byte-Datenstruktur verwendet. Sie unterscheidet sich von der analogen Protected-Mode-Struktur (Abbildung 5.48) nur dadurch, dass die 32-Bit-Offsets des last instruction pointers bzw. des last data pointers in zwei Words aufgesplittet vorliegen und keine Felder für Selektoren vorgesehen sind (der 32-Bit-Real-Mode kennt nur das »flache« Adressmodell, in dem jede Adresse durch einen 32-Bit-Offset dargestellt

werden kann. Segmente und damit Segment-Selektoren sind hier unbekannt!)

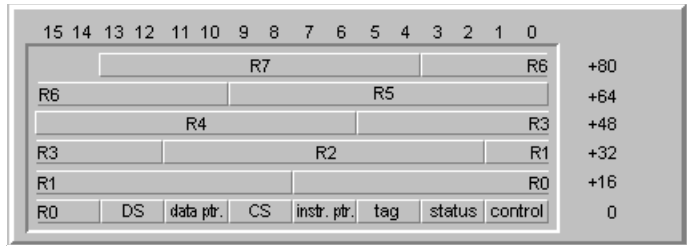


Abbildung 5.50: Speicherabbild der durch F(N)SAVE/FRSTOR im 16-Bit-Protected-Mode verwalteten Daten

Abbildung 5.50 zeigt die Struktur im 16-Bit-Protected-Mode. Hier ist sie, verglichen mit dem 32-Bit-Fall, deutlich kompakter, da Offsets hier nur 16 Bit umfassen und somit zweimal 16 Bit für die beiden Pointer eingespart werden können und die »Löcher« eliminiert sind. Resultat: Die Struktur umfasst nur noch 94 Byte.



Der Inhalt des Feldes Op für den Opcode fehlt in dieser Version der Datenstruktur! Falls der Opcode benötigt wird, muss er durch Auslesen des Speichers an der Adresse, die durch den last instruction pointer angegeben wird, eruiert werden.

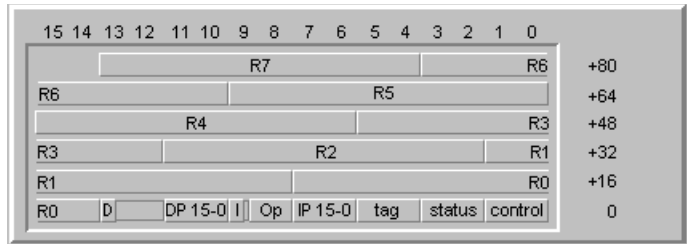


Abbildung 5.51: Speicherabbild der durch F(N)SAVE/FRSTOR im 16-Bit-Real-Mode verwalteten Daten

Im 16-Bit-Real-Mode schließlich finden sich alle Felder wieder, auch Opcode. Bemerkenswert in diesem Fall ist lediglich, dass die Felder für die Selektoren der Adressen des last instruction pointers und des last data pointers nur vier Bit umfassen. Dies ist jedoch nicht verwunderlich, beschränken sich doch Adressen im 16-Bit-Real-Mode auf 20 Bit. Die »fehlenden« vier Bit finden sich somit in den Feldern D (data pointer) und I (instruction pointer).

F(N)STENV FLDENV

F(N)STENV kann als »Schmalspurversion« des F(N)SAVE-Befehls aufgefasst werden, der lediglich die Kontroll-, Status- und Pointerregister der FPU sichert, nicht jedoch die Inhalte der FPU-Rechenregister. Und dies ist auch tatsächlich der Fall, wenn man die Abbildung 5.52 mit Abbildung 5.48 bis Abbildung 5.51 vergleicht. In diesem Fall wird man feststellen, dass die Datenstruktur für F(N)STENV/FLDENV genau an dem Punkt aufhört, an dem bei F(N)SAVE/FRSTOR die Inhalte des FPU-Registers R0 beginnt. Analog ergeben sich für die verschiedenen Betriebsmodi des Prozessors (real mode, protected mode) und die Umgebungen (16-Bit bzw. 32 Bit) verschiedene Aufteilungen und Größen der Strukturen. Sehr viel mehr gibt es zu diesen Strukturen nicht zu sagen.

Abbildung 5.52 zeigt die von F(N)STENV/FLDENV verwendeten Datenstrukturen in Abhängigkeit der verschiedenen Betriebsmodi und Umgebungen. Links oben ist der 32-Bit-Protected-Mode dargestellt, rechts oben der 32-Bit-Real-Mode. Es folgen links unten der 16-Bit-Protected-Mode sowie rechts unten der 16-Bit-Real-Mode.

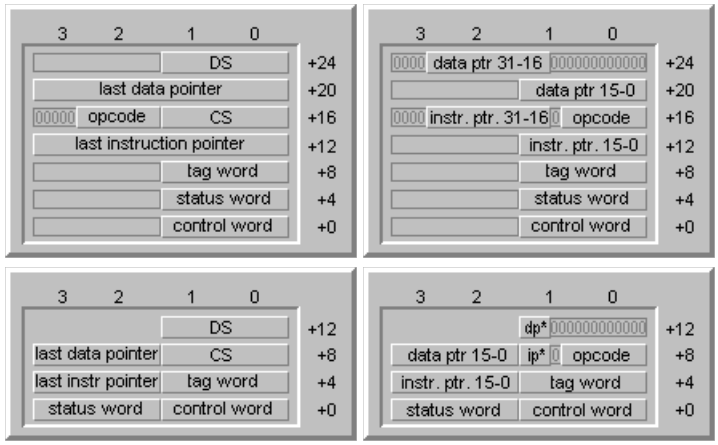


Abbildung 5.52: Speicherabbild der durch FSTENV/FLDENV in verschiedenen Betriebsmodi und Umgebungen verwalteten Daten

Bitte beachten Sie auch in diesem Fall: Der Inhalt des Feldes Op für den Opcode fehlt in der 16-Bit-Protected-Mode-Version der Datenstruktur! Falls der Opcode benötigt wird, muss er durch Auslesen des Speichers an der Adresse, die durch den last instruction pointer angegeben wird, eruiert werden.



5.9 Historie

In diesem Kapitel werden die Unterschiede der einzelnen Prozessoren untereinander angesprochen. Für jede Prozessorgeneration wird angegeben, welche Veränderungen im Vergleich zum Vorgänger stattgefunden haben. Um also einen chronologischen Überblick zu erhalten, sollten Sie auf Seite 896 mit dem 8086/8087 beginnen und sich rückwärts bis hierher vorarbeiten.

5.9.1 Pentium 4

Der aktuelle Pentium 4 ist der Gründungsvater der Pentium-4-Familie, einer Familie von Prozessoren, die mit einer neuen Architektur (der sog. NetBurst micro-architecture) ausgestattet sind. Dies verspricht eine deutliche Verbesserung der Performance unter anderem durch erhöhte Taktraten, eine *rapid execution engine*, eine *hyper pipelined technology, advance dynamic execution* und eine neuen Cache-Technologie.

Befehle Erweiterung der SIMD-Befehle im Rahmen der zweiten Stufe der *streaming single instruction multiple data extensions* (SSE2).

Flags Es wurden keine neuen Flags implementiert.

Exceptions Es wurden keine neuen Exceptions implementiert

5.9.2 Pentium III, Xeon

Wie der Pentium II Xeon eine High-End-Version des Pentium II ist, ist der Xeon eine High-End-Version des Pentium III.

Befehle Mit dem Pentium III wurden nicht nur der XMM-Befehlssatz und die damit verbundene Hardware (XMM-Register, MXCSR) eingeführt, sondern auch eine Technologie, die Intel SSE nennt: *streaming SIMD extension*.

Flags Der Pentium III erhielt ein neues Flag, das Kontrollregister CR4 (vgl. Seite 860) betrifft:

- Flag *OSXMMEXCEPT* (Bit 10) wurde implementiert. Das Betriebssystem signalisiert über dieses Flag, ob es einen *exception handler* für unmaskierte SIMD-Fließkomma-Exceptions zur Verfügung stellt.

Exceptions

Es wurde eine neue Exception definiert: SIMD-Fließkomma-Exception #XF (Interrupt 19).

Kompatibilität

Die Befehle FXSAVE und FXRSTOR wurden erweitert, um den Zustand der XMM-Register sowie des MXCS-Registers sichern bzw. restaurieren zu können.

5.9.3 Pentium II, Pentium II Xeon, Celeron

Pentium II Xeon und Celeron sind »Spielarten« des Pentium-II-Prozessors. Der Pentium II Xeon wurde durch verschiedene performancesteigernde Maßnahmen (2-MByte-Second-Level-Cache, 4-, 8-way und höhere Skalarität etc.) darauf spezialisiert, in High-End-Workstations und -Servern Dienst zu leisten. Der Celeron dagegen zielte eher auf den Low-Cost-Bereich von PCs.

Befehle

Der Pentium II fügte der P6-Familie die MMX-Befehle hinzu, die die späten Pentiums definierten und die dem Pentium Pro noch fehlten.

Flags

Mit dem Pentium II erhielt das Kontrollregister CR4 (vgl. Seite 860) ein neues Flag:

- Es wurde das Flag *OSFXSR* (Bit 9) implementiert. Das Betriebssystem signalisiert über dieses Flag, dass es bei context switches die Befehle FXSAVE und FXRSTOR benutzt, um den Zustand der FPU- und MMX-Register zu sichern bzw. zu restaurieren.

Exceptions

Der Pentium II definiert keine neuen Exceptions!

5.9.4 Pentium Pro

Mit dem Pentium Pro begründete Intel die »P6-Familie«. Zu dieser Familie gehören neben dem Pentium Pro auch der Pentium II, der Pentium II Xeon, der Celeron, der Pentium III und der Pentium III Xeon.

Der Pentium Pro ist ein »three-way superscalar processor«: Er kann bis zu drei Befehle gleichzeitig ausführen. Dazu benutzt er drei parallel arbeitende Dekodiereinheiten, die die Befehlssequenzen in »kleinere Einheiten«, *micro-ops* genannte micro-architecture op-codes zerlegen. Diese micro-ops werden in einen instruction pool eingeführt, aus dem fünf parallel arbeitende execution units bedient werden: zwei integer units, zwei floating-point units und eine memory unit. Diese units bearbeiten dann, falls die gegenseitigen Abhängigkeiten dies zulassen, die micro-ops »out of order«, also ohne die Reihenfolge einhalten zu müssen, die

der Befehlsstrom vorgibt. Eine retirement unit sorgt für die korrekte Re-Integration der bearbeiteten micro-ops in den Befehlsstrom und berücksichtigt dabei etwaige Programmverzweigungen.

Daneben wurde auch am Cache etwas getan: Neben dem first level cache des Pentium wurde ein second level cache mit 256 kByte Größe eingeführt.

Befehle Beim Pentium Pro wurden folgende Befehle implementiert:

- CMOVcc
- FCMOVcc
- FCOMI, FCOMIP, FUCOMI, FUCOMIP
- RDPMC; dieser Befehl wurde nachträglich auch bei einigen parallel zum Pentium Pro hergestellten Pentium-Prozessoren implementiert.
- UD2

Flags Der Pentium Pro erhielt drei neue Flags, die alle Kontrollregister CR4 (vgl. Seite 860) betreffen:

- Flag *PAE* (Bit 5) wurde implementiert. Wird dieses Flag gesetzt, wird der Paging-Mechanismus dahingehend verändert, 36-Bit- anstelle von 32-Bit-Adressierung bei der Berechnung physikalischer Adressen zu benutzen.
- Einführung des Flags *PGE* (Bit 7). Dieses Bit verhindert bei Taskwechseln das Entfernen von »global pages« (häufig oder gemeinsam benutzte Seiten) aus dem translation lookaside buffer (TLB).
- Implementation des Flags PCE (Bit 8). Dieses Flag erlaubt die Ausführung des Befehls RDPMC von jeder Privilegstufe aus. Bei gelöschtem Flag kann dieser Befehl nur aus Stufe 0 ausgeführt werden.

Exceptions Der Pentium Pro hat keine neuen Exceptions definiert!

Kompatibilität Da die Entwicklung des Pentium Pro unabhängig von der Einführung der SIMD-Technologie erfolgte, verfügte dieser Prozessor ähnlich wie die frühen Pentiums noch nicht über die MMX-Befehle. In Sachen Multimedia-Erweiterungen ist ein Pentium Pro somit mit einem frühen Pentium gleichzusetzen.

5.9.5 Pentium

Der Pentium führte eine zweite execution unit ein und implementierte damit die »Superskalar-Technologie«. Mit den beiden Pipelines, die üblicherweise als u und v pipeline bezeichnet werden, war es nun möglich geworden, bis zu zwei Instruktionen pro Takt auszuführen. Pferdefuß bei der ganzen Angelegenheit ist jedoch, dass nur sog. parallelisierbare Instruktionen auch tatsächlich die beiden Pipelines parallel nutzen können.

Weitere Anstrengung zur Verbesserung der Performance wurde unternommen, indem der first level cache auf 8 kByte vergrößert und zusätzlich ein 8-kByte-Datencache implementiert wurde. Eine neue »branch prediction unit« wurde damit beauftragt, in Schleifen das wahrscheinlichste Sprungziel vorherzusagen. Obschon die Datenleitungen »nach außen« 32-bittig blieben und auch die Register weiterhin mit 32 Bits arbeiteten, wurden die Datenströme intern auf 128-, ja sogar 256-Bit-Wegen herumgeschaufelt. Der Paging-Mechanismus wurde dahingehend erweitert, dass nun auch 4-MByte-Pages neben den beim 80486 eingeführten 4-kByte-Pages unterstützt werden.

Der Pentium definierte die »Pentium-Familie«. Zu ihr gehörte neben dem »Ur-Pentium« auch eine »späte« Pentium-Version, die über eine für Multimedia-Anwendungen wichtige Technologie verfügte: MMX. Mit den multi-media extensions begründete Intel die sog. SIMD-Technologie: single instruction, multiple data. Wenn man so will: eine nochmals erhöhte Parallelität in der Befehlsverarbeitung – zumindest bei einigen neuen, aber wichtigen Befehlen.

Befehle

Beim Pentium wurden folgende neue Befehle implementiert:

- CMPXCHG8B
- CPUID; dieser Befehl wurde nachträglich auch bei einigen parallel zum Pentium hergestellten 80486-Prozessoren implementiert.
- RDTSC
- RDMSR
- WRMSR
- MMX-Befehle (nur bei »Pentium mit MMX«) und damit auch die Befehle
- FXSAVE und
- FXRSTOR

Flags Das EFlags-Register (vgl. Seite 38) wurde um folgende Flags erweitert:

- VIF, *virtual interrupt flag*, (Bit 19). Es ist die virtuelle Form des Interrupt-Flags IF (Bit 9) im virtual 8086 mode.
- VIP, *virtual interrupt pending*, (Bit 20) zeigt im virtual 8086 mode an, dass ein Interrupt anhängig ist.
- ID, *identification flag*, (Bit 21) zeigt die Verfügbarkeit des Befehls CPUID an.

In Kontrollregister CR0 wurde das Flag ET wieder in den »reservierten Zustand« zurückgesetzt. Es diente lediglich in 80386ern und 80486ern der Nutzung der 80387DX-Instruktionen.

Der Pentium erhielt mit CR4 ein neues Kontrollregister (vgl. Seite 860). Mit diesem Kontrollregister war auch die Einführung neuer Flags verbunden:

- Das Flag VME (Bit 0) erlaubt die Unterstützung eines virtuellen Interrupt-Flags im virtual 8086 mode.
- Das Flag PVI (Bit 1) unterstützt wie VME ein virtuelles Interrupt-Flag im V86-Modus.
- Mit Flag TSD (Bit 2) kann die Ausführung des Befehls RDTSC auf die Privilegstufe 0 eingeschränkt werden.
- Flag DE (Bit 3) spielt in Verbindung mit den Debug-Registern eine Rolle und generiert undefinded opcode exceptions (#UD), wenn in Register DR4 oder DR5 Referenzen für eine verbesserte Performance stehen.
- 4-MByte-Pages können benutzt werden, wenn Flag PSE (Bit 4) gesetzt wird.
- Mit MCE (Bit 6) wird die Generierung einer neu eingeführten Exception, der machine-check exception #MC, ermöglicht.

Exceptions Der Pentium hat folgende neue Exceptions eingeführt bzw. erweitert:

- Die machine check exception #MC (Interrupt 18) wurde neu eingeführt. Diese Exception signalisiert Hardwarefehler und ist modellspezifisch, was bedeutet, dass sie in anderen Prozessoren anders oder gar nicht implementiert sein kann. Zu dieser Exception gehört das Flag MCE in CR4.

- Die general protection exception #GP (Interrupt 13) wurde dahingehend erweitert, dass das Schreiben einer »1« an reservierte Positionen in Registern zu einer #GP führt.
- Auch die page fault exception #PF (Interrupt 14) wurde analog erweitert: Wann immer während einer Adressberechnung eine »1« an einer reservierten Position im page table entry, page directory entry oder page directory pointer vorgefunden wird, wird eine #PF ausgelöst.

Kompatibilität

Mit dem Pentium wurden neue Algorithmen eingeführt, mit denen die transzendentalen Funktionen FISN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X und FYL2XP1 berechnet werden. Diese Algorithmen erlauben eine höhere Genauigkeit des Ergebnisses als bei vorangehenden FPUs/NPXen. Die Genauigkeit wird gemessen in ULPs (*units in the last place*) und wird definiert als

$$\text{Fehler} = |(f(x) - F(x)) / 2^{k-63}|$$

wobei f(x) der exakte und F(x) der berechnete Wert ist und k eine Integer, für die gilt:

$$1 \leq 2^{-k} \cdot f(x) < 2$$

Für den Pentium und folgende Prozessoren liegt der Fehler im ungünstigsten Fall bei weniger als einer ULP, wenn »in Richtung zur nächsten oder geraden Zahl« gerundet wird (bei FYL2X und FYL2XP1 ist dies nur richtig, wenn der Wert in ST(1) = 1 ist, andernfalls liegt der Fehler bei 1.35 ULPs), und bei weniger als 1.5 ULPs bei anderen Rundungsarten. Damit unterschieden sie sich ggf. von den Ergebnissen beim 80486 um 2 bis 3 ULPs. Als Folge kann daher auch das Flag C1 des condition codes, das ja das Ergebnis einer Rundung signalisiert, unterschiedlich gesetzt sein.

5.9.6 80486

Die wesentlichen Neuerungen beim 80486 lagen in der Erhöhung der Performance. Dies wurde dadurch bewerkstelligt, dass die mit dem 80386 begonnene »Parallelisierung« der Befehlsverarbeitung ein Stück weiter getrieben wurde. So wurde die Dekodierungs- und Ausführungseinheit (»decode and execution unit«) des 80386 in fünf hintereinander geschaltete Stufen (»pipelined stages«) aufgeteilt, in der die Stufen aller beteiligten Units parallel arbeiteten. Auf diese Weise konnten bis zu fünf Instruktionen in verschiedenen Zuständen gleichzeitig bear-

beitet werden. Der erste Schritt zur Eine-Instruktion-Pro-Takt-Technologie war getan. Um dies auch hinsichtlich der Kommunikation mit dem (vergleichsweise langsamen) Speicher hinzubekommen, wurde ein 8-kByte-First-Level-Cache eingeführt und somit der Grundstock zu der heutigen modernen Cache-Technologie geschaffen.

Und einen Aspekt sollte man ebenfalls nicht vergessen: Der 80486 war der erste Prozessor, bei dem der NPX auf dem Chip der CPU realisiert wurde: Der NPX war zur FPU mutiert.

Befehle Beim 80486 wurden folgende neue Befehle implementiert:

- BSWAP
- CMPXCHG
- INVLD
- INVLPG
- RSM; dieser Befehl wurde mit dem 80386SL bzw. dem 80486SL eingeführt
- WBINVLD
- XADD

Flags Das EFlags-Register (vgl. Seite 38) wurde um folgende Flags erweitert:

- Das Flag AC, alignment check, (Bit 18) hinzugefügt. Es dient dazu, die Fähigkeit des Prozessors zur Überprüfung der korrekten Ausrichtung von Daten (»alignment check«) zu steuern.

In Kontrollregister CR3 (vgl. Seite 860) wurden folgende Flags zusätzlich aufgenommen:

- Flag PCD (Bit 4), page level cache disable; der Zustand dieses Bits wird während Buszyklen, bei denen Paging nicht zum Einsatz kommt, wie z.B. bei interrupt acknowledge cycles, auf den Pin PCD# gelegt, wenn Paging eingeschaltet wurde. Dadurch kann ein externer cache das caching Zyklus für Zyklus kontrollieren.
- Flag PWT (Bit 3), page level write-through; analog PCD wird der Zustand dieses Bits bei »nicht gePAGEten« Buszyklen auf den PWT#-Pin gelegt, wenn Paging freigeschaltet ist. Ein externer cache kann auf diese Weise zyklusweise einen »write-through« kontrollieren.

Schließlich wurden in Kontrollregister CR0 (vgl. Seite 856) folgende Flags zusätzlich aufgenommen:

- NE (Bit 5), numeric error. Es steuert den normalen Mechanismus für das Signalisieren von FPU-Exceptions
- WP (Bit 16), write protect. Es verhindert, dass Pages im user mode durch Zugriffe aus dem supervisor mode beschrieben werden können.
- AM (Bit 18), alignment mask. Es ist eine Erweiterung des EFlags-Register-Flags AC (alignment check) auf den Paging-Mechanismus und steuert, ob ein alignment check durchgeführt werden soll.
- NW (Bit 29), not write-through. Wenn gesetzt, verhindert es bei Cache-Treffern Write-Throughs und Invalidierungszyklen.
- CD (Bit 30), cache disable. Durch dieses Bit wird ein caching kontrolliert.

Exceptions

Der 80486 führt eine neue Exception ein:

- Alignment-check exception #AC (Interrupt 17). Diese Exception wird, so das AC-Flag in EFlags und/oder das AM-Flag in CR0 gesetzt ist, ausgelöst, wenn ein Datum nicht »ausgerichtet« ist.

Kompatibilität

Bei 80486-Prozessoren wird ein WAIT/FWAIT, das einem FPU-Befehl vorangeht, der sich automatisch mit dem vorangehenden FPU-Befehl synchronisiert, als NOP behandelt. Dies führt dazu, dass anhängige Exceptions nun nicht im Kontext des als NOP interpretierten WAIT/FWAITs behandelt werden, sondern erst im Kontext des darauf folgenden FPU-Befehls. Die Auslösung der FPU-Exception erfolgt hier also einen Befehl »später« als bei 80386ern oder Pentium- oder folgenden Prozessoren.

5.9.7 80386 / 80387

Die wesentlichste Veränderung beim 80386 / 80387 im Vergleich zu den Vorgängern ist die Einführung einer neuen Architektur. Handelte es sich bei den Vorgängern noch um »16-Bit-Prozessoren«, deren Register maximal 16 Bit aufnehmen konnten, so ist der 80386 der erste Vertreter der »32-Bit-Prozessoren«. Allerdings konnte vollständige »Abwärtskompatibilität« dadurch gewährleistet werden, dass die jeweils »unteren« 16 Bits aller 32-Bit-Register die Funktion der 16-Bit-Register der Vorgänger emulieren konnten und heute noch können.

Als weiteres Highlight wurde der *virtual 8086 mode* als »Spielart« des protected mode eingeführt. Mit ihm war es möglich, Real-Mode-Programme in einer »geschützten Umgebung« ablaufen zu lassen. Das bedeutete, dass das Programm sich in einer Real-Mode-Umgebung fühlte, die jedoch in Wahrheit eine »Blase« im protected mode darstellte. Dies verhinderte, dass die zu diesem Zeitpunkt noch sehr beliebten DOS-Programme aufgrund fehlender Schutzkonzepte das gesamte System lahmlegen konnten.

Mit der Implementation des Paging-Mechanismus (vgl. Seite 445) wurde darüber hinaus eine virtuelle Speicherverwaltung eingeführt, die der Segment-Swapping-Technik des 80286 erheblich überlegen war. Hierzu wurden sog. *pages* definiert mit einer konstanten Größe von 4 kByte.

Weitere Neuerungen des 80386 spielten sich hauptsächlich im Hintergrund und für Programmierer verborgen ab. So wurden mit dem 80386 die ersten Schritte in Richtung paralleler Verarbeitung gemacht, um die Performance zu steigern. Der Zugriff auf den I/O- und Speicherraum erfolgte nun über eine *bus interface unit*, die die Daten allen Units zur Verfügung stellen konnte (was Auswirkungen auf das Zusammenspiel CPU – NPX hatte!). Befehlssequenzen bis zu 16 Bytes konnten von dieser bus unit in eine *code prefetch unit* (= »prefetch queue«) geladen werden, was bedeutete, dass die bis zu 16 Bytes eines Befehls gleichzeitig decodiert werden konnten. Aus dieser code prefetch unit wurde die *instruction decode unit* bedient, die die Befehle in Microcode-Instruktionen übersetzte, sowie die *segmentation unit*, die für die Berechnung der virtuellen Adresse der ggf. vorhandenen Speicheroperanden sowie für Zugriffsprüfungen im Rahmen der Schutzkonzepte zuständig war. Die *paging unit* schließlich setzte diese virtuellen Adressen in eine physikalische um und sorgte im Rahmen des *Pagings* dafür, dass die gewünschte Adresse physikalisch auch tatsächlich verfügbar war.

Befehle Alle Befehle, die von der Architektur abhängen (weil sie Register involvieren), wurden auf die 32-Bit-CPUs umgeschrieben. Dies erfolgte vollkommen transparent, sodass keine weiteren Maßnahmen erforderlich sind: Für 16-Bit-Prozessoren geschriebener Code läuft unter Berücksichtigung der jeweils genannten (In-) Kompatibilitäten unverändert auf 32-Bit-Prozessoren.

Der 80386 wurde aber auch um folgende Befehle erweitert:

- BSF, BSR
- BT, BTC, BTR, BTS
- LFS, LGS und LSS
- MOVSX, MOVZX
- RSM; dieser Befehl wurde mit dem 80386SL bzw. dem 80486SL eingeführt
- SETcc
- SHRD, SHLD

Folgende Befehle wurden in ihren Möglichkeiten erweitert:

- IMUL; Erweiterung des Befehls um die Zwei- und Drei-Operanden-Form
- Jcc können nun als Distanz nicht nur 8-Bit-Offsets verarbeiten (»short jumps«), sondern 16-/32-Bit-Offsets (»near jumps«).
- MOV; nun können auch Kontroll- und Debugregister angesprochen werden

Der 80387 erhielt folgende neue Befehle:

- FPREM1
- FSIN, FCOS, FSINCOS
- FUCOM, FUCOMP und FUCOMPP

Register

Der 80386 erweiterte die Breite der meisten CPU-Register (vgl. Seite 31) auf 32 Bit. Darüber hinaus wurde durch die Einführung des Paging-Mechanismus ein weiteres Kontrollregister erforderlich, das CR2 (vgl. Seite 856), und das »machine status word« des 80286 wurde zum »echten« Kontrollregister CR0 ausgebaut.

Flags

Das EFlags-Register (vgl. Seite 38) wurde um folgende Flags erweitert:

- Das Feld IOPL (Bits 12 und 13). Dieser *I/O privileg level* legt fest, welche Privilegstufe ein Programm haben muss, um Zugriff auf die Ports (vgl. Seite 827) zu haben.
- NT, nested task (Bit 14). Dieses Flag zeigt an, ob der aktuelle task mit einem vorangehenden »verkettet« ist. Dies ist immer dann der Fall, wenn der aktuelle task im Rahmen eines CALLs von einem anderen task aufgerufen wurde.

- RF, resume flag (Bit 16). Dieses Flag steuert die Antwort des Prozessors auf debug exceptions #DB
- VM, virtual 8086 mode flag (Bit 17). Das Flag wird gesetzt, um in den virtual 8086 mode umzuschalten, und gelöscht, um in den protected mode zurückzukehren.

Das Kontrollregister CR0 wurde um folgende Flags erweitert:

- ET (Bit 4), *extension type*; dieses Flag ermöglichte die Nutzung der 80387DX-Instruktionen.
- PG (Bit 31), *paging*; es ermöglicht im gesetzten Zustand die Nutzung der neu eingeführten Paging-Mechanismen.

Ferner wurden die Flags des Kontrollregisters 2 implementiert (vgl. Seite 856).

Exceptions Bei den Exceptions erfolgten folgende Veränderungen:

- Divide error exception #DE (interrupt 0): Bei Auslösung dieser Exception zeigt CS:(E)IP immer auf die Instruktion, die die Exception zu verantworten hat. In vorangehenden Prozessoren zeigte CS:(E)IP jeweils auf den folgenden Befehl.
- Eine divide error exception #DE wurde bei vorangehenden Prozessoren ausgelöst, wenn als Quotient einer IDIV-Instruktion die negative MaxInt resultierte. Ab dem 80386 kann diese Zahl (8 Bit: $80, 16 Bit: $8000) generiert werden.
- Mit #UD, invalid bzw. undefined opcode exception (Interrupt 6) wurde eine neue Exception eingeführt, die immer dann ausgelöst wird, wenn die Befehlssequenz »verbotene« Kombinationen von Präfixen, Opcode-Bytes und/oder Adressen aufweist. Ursprünglich war sie gedacht, den verbotenen Gebrauch des Präfixes *LOCK* zu signalisieren.
- Die page fault exception #PF (Interrupt 14) wurde erweitert und an die neue Art der virtuellen Speicherverwaltung angepasst.
- Auch die general protection exception #GP (Interrupt 13) wurde erweitert. Der 80386 beschränkt die Länge einer Befehlssequenz auf 15 Bytes, da diese Länge nur durch redundanten Gebrauch von Präfixen überschritten werden kann. Daher wird eine #GP ausgelöst, wenn die Befehlssequenzlänge 15 Bytes überschreitet.

Kompatibilität

Mit dieser Änderung der Architektur gingen einige Veränderungen an den Befehlen einher. Daher wird in diesem Kapitel ein großer Anteil der Beschreibungen zur Kompatibilität erfolgen, die für den 80386, aber eben auch die folgenden Prozessoren bis zum Pentium 4 gelten. Ferner unterstützte Intel ab den 32-Bit-FPUs den IEEE-Standard 754, der bei der Entwicklung der 16-Bit-NPXe noch nicht verabschiedet war. Auch dies führte dazu, dass sich in den FPU-Befehlen einige Inkompatibilitäten ergeben haben.

Transzendentale Befehle

Bei transzendentalen Befehlen ist das round-up flag C1 des condition codes bei 16-Bit-NPXen undefiniert. 32-Bit-FPUs signalisieren durch dieses Flag, in welche Richtung das Ergebnis gerundet wurde.

FDIV FPREM FSQRT

Die 32-Bit-FPUs unterstützen bei diesen Befehlen denormalisierte Operanden. Daher kann es aus Gründen der Kohärenz mit dem IEEE-Standard 754 zu einer underflow exception #U kommen, wenn eine denormale als Operand eingesetzt wird. 16-Bit-NPXe erlauben keine denormalisierten Operanden oder Ergebnisse, die eine #U auslösen könnten. Stattdessen generieren sie eine invalid operation exception #I, falls ein Unterlauf stattfindet.

FSCALE

32-Bit-FPUs beschränken den Bereich des Exponenten in ST(1) nicht. Wenn Der Exponent absolut kleiner als 1 ist ($0 < |\text{Exponent}| < 1$), wird er auf 0 gesetzt und der Inhalt von ST(0) bleibt unverändert: $\text{ST}(0) \cdot 2^0 = \text{ST}(0)$. Falls das Ergebnis nicht »exakt« oder nur mit Verlust an Genauigkeit dargestellt werden kann, wird eine precision exception #P ausgelöst. Bei 16-Bit-NPXen ist der Wertebereich des Exponenten beschränkt. Falls der Exponent absolut kleiner als 1 ist, ist das Ergebnis undefiniert! ACHTUNG: Eine Exception wird hierbei nicht ausgelöst.

FPREM

Nach FPREM auf 32-Bit-FPUs repräsentieren die Flags C0, C3 und C1 des condition codes im status word die niedrigstwertigen drei Bits des Quotienten. Auf 16-Bit-NPXen ist das auch der Fall, jedoch existiert ein Bug: Diese Bits werden falsch gesetzt, falls die Reduktion einer Zahl mit dem Wert $(64^n + m)$ durchgeführt werden soll, wenn $n \geq 1$ und $m = 1$ oder 2 ist.

Stack-Überlauf

Bei 32-Bit-FPUs gibt die FPU je nach ausgeführtem Befehl im Zieloperanden eine Indefinite (»$\pm\infty$«) im Real-, Integer- oder BCD-Format (vgl. Seite 788) zurück, falls ein FPU-Stack-Überlauf stattfindet und die invalid operation -exception #I maskiert ist. Bei 16-Bit-NPXen bleibt der ursprüngliche Operand unverändert, wird jedoch in ST(1) geladen.

F2XM1

Der Wertebereich für die Operanden ist bei 32-Bit-FPUs deutlich größer ($-1 < \text{ST}(0) < +1$) als bei 16-Bit-NPXen ($0 \leq \text{ST}(0) \leq 0.5$).

FLD Aufgrund der Unterstützung des IEEE-Standards 754 in 32-Bit-FPUs haben sich beim FLD-Befehl einige Inkompatibilitäten ergeben. So generieren 32-Bit-FPUs keine denormal operand exception #D, wenn eine ExtendedReal geladen werden soll. 16-Bit-NPXe dagegen generieren diese Exception. Ferner wird bei 32-Bit-FPUs eine im Single- oder DoubleReal-Format vorliegende Denormale beim Laden in das ExtendedReal-Format konvertiert. 16-Bit-NPXe dagegen konvertieren sie in eine Unnormale – diese Art »Zahlen« werden von 32-Bit-FPUs nicht unterstützt! Dies bedeutet, dass unterschiedliche Ergebnisse bei 32-Bit-FPUs und 16-Bit-NPXen auftreten, wenn die Ladebefehle FXTRACT oder FXAM ausgeführt werden. Schließlich führt das Laden einer sNaN im Single- oder DoubleReal-Format bei 32-Bit-FPUs zu einer invalid operation exception #I, während das bei 16-Bit-NPXen nicht der Fall ist.

Ladebefehl für Konstanten Anders als bei 16-Bit-NPXen ist bei 32-Bit-FPUs der Rundungsmechanismus aktiviert, wenn Konstanten geladen werden. Das bedeutet, dass bei FLDPI, FLDLN2, FLDLG2 und FLDL2E die geladene Konstante bei 32-Bit-FPUs den gleichen Wert hat wie bei 16-Bit-NPXen, wenn im Feld rounding control des control words der FPU der Code für »Runden zur nächsten oder geraden Zahl« oder »Runden in Richtung +∞« eingetragen ist. Andernfalls unterscheiden sich die Werte. Bei FLDL2T sind die Werte wiederum identisch, wenn RC auf »Runden zur nächsten oder geraden Zahl« oder »Runden in Richtung -∞« oder »Runden in Richtung 0« steht. Dieses unterschiedliche Verhalten wurde implementiert, um den IEEE-Standard 754 einzuhalten. Übrigens: Das betrifft die letzte Stelle – der eventuelle Fehler hält sich somit in Grenzen.

FXTRACT Bei 32-Bit-FPUs wird eine divide by zero exception #Z ausgelöst, wenn der Operand des FXTRACT-Befehls 0 ist. In diesem Fall wird in ST(1) eine negative Infinite abgelegt (»-∞«). Grund: FXTRACT ist ein verkappter Logarithmus-Befehl. Und nicht nur nach IEEE-Standard 754, der durch dieses Verhalten eingehalten werden soll, ist der Logarithmus von 0 nicht definiert! Stellt der Operand dagegen eine positive Infinite dar (»+∞«), wird keine Exception ausgelöst. Da sich 16-Bit-NPXe nicht an den Standard gehalten haben (weil er zum Zeitpunkt der Entwicklung noch nicht verabschiedet war), wird hier der Wert »0« in ST(1) abgelegt und keine Exception ausgelöst, falls der Operand »0« vorgefunden wird. Im Falle einer positiven Infiniten wird eine invalid operation exception ausgelöst.

FSAVE FSTENV Bei 32-Bit-FPUs ist nach FSAVE und FSTENV der Inhalt des Last-Operand-Pointer-Feldes undefiniert, wenn die vorangegangene FPU-

Instruktion keinen Speicheroperanden eingebunden hat! Bei 16-Bit-NP-Xen besteht er aus der Adresse des zuletzt verwendeten Speicheroperanden, der jedoch nicht zwangsläufig im letzten, sondern in weiter zurückliegenden FPU-Befehlen zum Einsatz gekommen sein kann.

FPATAN

Bei 32-Bit-FPUs ist der Wertebereich der Operanden nicht eingeschränkt. Bei 16-Bit-NPXen dagegen muss der absolute Betrag des Operanden im TOS (= ST(0)) kleiner sein als der absolute Betrag des Operanden in ST(1).

FPTAN

Bis zum 80387 war FPTAN der zentrale Befehl, wenn es um trigonometrische Berechnungen ging. Der 80287 kannte genauso wenig wie der 8087 den Befehl FSIN bzw. FCOS, geschweige denn FSINCOS. Alle periodischen Funktionen mussten von FPTAN hergeleitet werden.

Dazu erzeugte FPTAN aus dem im Radianten-Maß übergebenen Winkel zwei Werte, die er in ST(0) und ST(1) ablegte. Diese Werte waren Zwischenergebnisse bei der Berechnung des Tangens (daher auch der Name »partieller« Tangens). Mathematisch stellen sie die rechtwinkligen Koordinaten eines Punktes dar, der in »Polarkoordinaten« (Winkel) auf dem Einheitskreis (Abstand vom Ursprung = 1!) angegeben wurde. FPTAN führt somit die Transformation von Polar- in rechtwinklige Koordinaten durch, wobei logischerweise $0 \leq a, b \leq 1$ mit $a^2 + b^2 = 1$ ist.

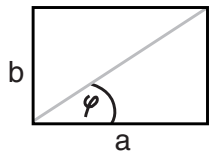


Abbildung 5.53: Bildung des partiellen Tangens

Nachdem gemäß Abbildung 5.53 der Tangens mathematisch definiert ist als

$$tan(\varphi) = b / a,$$

wobei »b« die Gegenkathete und »a« die Ankathete im rechtwinkligen Dreieck ist, kann, wenn man die rechtwinkligen Koordinaten a (X-Achsenabschnitt) und b (Y-Achsenabschnitt) in den FPU-Registern ablegt, der Tangens einfach und schnell durch Division dieser Werte berechnet werden.

Durch die Berechnung der rechtwinkligen Koordinaten mittels FPTAN können jetzt auch die anderen trigonometrischen Funktionen berechnet werden. So ist der Sinus definiert als

$$sin(\varphi) = b / c,$$

wobei nach Pythagoras $a^2 + b^2 = c^2$ ist, die Hypotenuse c also mit Hilfe der Katheten berechnet werden kann, die FPTAN ja liefert:

$$sin(\varphi) = b / c = b / \sqrt{(a^2 + b^2)},$$

Analog ist der Cosinus, Cotangens, Secans und Cosecans berechenbar nach

$$cos() = a / c = a / \sqrt{(a^2 + b^2)}$$
$$cot(\varphi) = a / b$$
$$sec(\varphi) = 1 / cos(\varphi) = c / a = \sqrt{(a^2 + b^2)} / a$$
$$csc(\varphi) = 1 / sin(\varphi) = c / b = \sqrt{(a^2 + b^2)} / b.$$

Das bedeutet: Die Bildung des partiellen Tangens ist eine zentrale Berechnung zur Bestimmung aller möglichen trigonometrischen Werte.

Doch eigentlich benötigt man die Katheten nicht wirklich für die Berechnung. Durch etwas geschickte mathematische Umformulierung lassen sich unter der Voraussetzung

$$tan(\varphi) = b / a,$$

alle trigonometrischen Berechnungen über den Tangens erledigen:

$$sin(\varphi) = b / (a \cdot \sqrt{(1 + (b/a)^2)}) = tan(\varphi) / \sqrt{(1 + tan^2(\varphi))}.$$

$$cos(\varphi) = a / (a \cdot \sqrt{(1 + (b/a)^2)}) = 1 / \sqrt{(1 + (b/a)^2)} = 1 / \sqrt{(1 + tan^2(\varphi))}.$$

$$cot(\varphi) = 1 / tan(\varphi)$$

$$sec(\varphi) = \sqrt{(1 + tan^2(\varphi))}$$

$$csc(\varphi) = \sqrt{(1 + tan^2(\varphi))} / tan(\varphi)$$

Somit bleibt als eigentlicher Grund für die Bildung des partiellen Tangens nur die Umrechnung von Polar- in rechtwinklige Koordinaten!



Durch die Beschränkung des Wertebereiches für den als Argument übergebenen Winkel auf max. $\pi/4$ (= 45°) beim 8087 und 80287 gilt diese Beziehung nur innerhalb des ersten Oktanten. Daher ist es wichtig und notwendig, die Skalierung des Arguments mittels FPREM1 durchzuführen. Diese liefert im condition code auch den Oktanten zurück, in dem der Punkt tatsächlich liegt. Über diese Information und die in Ta-

belle 5.32 dargestellten Zusammenhänge kann somit ein genereller Algorithmus entwickelt werden.

Oktant	Winkel	sin	cos	tan	cot	sec	csc
0	0 – 45	sin(φ)	cos(φ)	tan(φ)	cot(φ)	sec(φ)	csc(φ)
1	45 – 90	cos(C-φ)	sin(C-φ)	cot(C-φ)	tan(C-φ)	csc(C-φ)	sec(C-φ)
2	90 – 135	cos(φ)	-sin(φ)	-cot(φ)	-tan(φ)	-csc(φ)	-sec(φ)
3	135 – 180	sin(C-φ)	-cos(C-φ)	-tan(C-φ)	-cot(C-φ)	-sec(C-φ)	-csc(C-φ)
4	180 – 225	-sin(φ)	-cos(φ)	tan(φ)	cot(φ)	-sec(φ)	-csc(φ)
5	225 – 270	-cos(C-φ)	-sin(C-φ)	cot(C-φ)	tan(C-φ)	-csc(C-φ)	-sec(C-φ)
6	270 – 315	-cos(φ)	sin(φ)	-cot(φ)	-tan(φ)	csc(φ)	sec(ϕ)
7	315 - 360	-sin(C-φ)	cos(C-φ)	-tan(C-φ)	-cot(C-φ)	sec(C-φ)	csc(C-φ)

Tabelle 5.32: Oktanten-abhängige Berechnung der trigonometrischen Funktionen. C = π/4



Aus Kompatibilitätsgründen heißt auch beim 80387, beim 80487 und den FPUs der folgenden Prozessorgenerationen der für die Tangensbildung zuständige Befehl FPTAN, obwohl in ST(1) bereits der »echte« und kein partieller Tangens liegt und folgerichtig in ST(0) als Divisor die Konstante 1.0 abgelegt wird. Mathematisch betrachtet handelt es sich hierbei um eine Skalierung der An- und Gegenkathete in der Art, dass die Ankathete (X-Achsenabschnitt) immer den Wert 1.0 erhält. Dadurch arbeitet zwar Software, die den Tangens durch Division der beiden Register bildet (TAN = ST(1) / ST(0)), auf Coprozessoren/FPUs ab dem 80387 korrekt. Allerdings ist eine Umrechnung der Polarkoordinaten eines Punktes in rechtwinklige durch FPTAN nicht mehr möglich!



Und noch eine wesentliche Änderung hat sich mit dem 80387 ergeben: Der gültige Bereich für die als Argument übergebenen Werte lag bis zum 80387 bei $0 \leq \varphi < \pi/4$, sodass die Software dafür zuständig war, das korrekte Vorzeichen und den korrekten Wert des Tangens eines Winkels aus dessen Lage im entsprechenden Oktanten des Einheitskreises zu berechnen (vgl. Tabelle 5.32). Seit dem 80387 ist der Wertebereich erweitert worden auf $-2^{63} < \varphi < +2^{63}$, sodass die nachfolgenden rechenintensiven Nachberechnungen anhand der Oktanten und die Bildung eines partiellen Restes mittels FPREM1 in den meisten Fällen entfällt.

CPU - NPX

Auch in dieser Hinsicht hat sich etwas getan. Die Zusammenarbeit zwischen CPU und NPX ist sehr viel enger geworden und ähnelt bereits sehr stark derjenigen, die ab dem nächsten Prozessor, dem 80486, die übliche werden soll: Der NPX mutiert zur FPU, einer floating-point unit auf dem Chip der CPU. (Der 80487 war tatsächlich eine CPU mit integrierter FPU! Umgekehrt war der 80486 ein 80487 mit abgeschalteter FPU. Wurde ein 80486 mit einer »NPX«, einem 80487, nachgerüstet, hieß das, dass der 80487 den bestehenden 80486 einfach abschaltete.) Das bedeutete insbesondere, dass der mit dem 80287 eingeführte Befehl FSETPM, floating-point set protected mode, obsolet und entfernt wurde. Der 80387 kannte keine unterschiedlichen Betriebmodi mehr – er war bereits vollständig davon abhängig, dass die CPU gemäß des eingestellten Betriebsmodus die Daten bereitstellte oder entsorgte, mit denen die NPX umging.

EFlags

Die Zustände der Bits 15 bis 12 des EFlags-Registers nach einem PUSHF, nach Rückkehr aus einem Interrupthandler oder nach Exceptions unterscheidet sich bei 32-Bit-Prozessoren von denen bei 16-Bit-Prozessoren:

- Beim 8086 sind die Bits 15 bis 12 grundsätzlich gesetzt.
- Beim 80286 sind die Bits 15 bis 12 im real mode grundsätzlich gelöscht
- Bei 32-Bit-Prozessoren im real mode ist Bit 15 grundsätzlich gelöscht (reserviert) und die Bits 14 bis 12 haben den Inhalt, den sie vor der Operation hatten.

5.9.8 80286 / 80287

Der 80286 zeichnete sich seinen Vorgängern gegenüber durch zwei wesentliche Änderungen aus:

- Erhöhung der Adressleitungen von 20 auf 24 und
- Einführung des protected mode als »neuen« Betriebsmodus.

Die erste Veränderung hatte zur Folge, dass die CPU nun nicht mehr nur 1 MByte (= 2^{20} Byte) adressieren konnte, sondern (damals) »wahnsinnige« 2^{24} Bytes = 16 MByte. Das Problem war jedoch: Konnte bereits auf 8086-Prozessoren der verfügbare Adressraum von 1 MByte nur mit »Haken und Ösen« einigermaßen effektiv genutzt werden (Oldtimer wie ich werden sich an expanded und extended memory und Speichererweiterungskarten auf 4 MByte (!) erinnern), war gar nicht daran zu denken, die 16 MByte auch nur ansatzweise ausnutzen zu können.

Daher resultierte fast zwangsläufig die Einführung eines neuen Betriebsmodus, der die Restriktionen des real mode des 8086 nicht mehr aufwies: Der protected mode war geboren. In diesem Modus war aufgrund einer vollständig neuen Architektur (vor allem in der Adressberechnung) tatsächlich die Nutzung des gesamten adressierbaren Speicherraums möglich.

Das Problem dabei war nur: Die Register des 80286 waren 16-bittig geblieben. Das bedeutete, dass Segmente auch im neuen Betriebsmodus nur 64 kByte groß werden konnten. Der Adressraum konnte also bei dieser frühen Form des protected mode, den man daher auch 16-Bit-Protected-Mode nennt, noch nicht »flat« angesprochen werden, wie das heute der Fall ist.

Dieser Sachverhalt und die recht große Popularität des 16-Bit-Real-Mode-Betriebssystems DOS führte dazu, dass erst sehr viel später, weit nach Einführung der ersten 32-Bit-Prozessoren, der protected mode zu Amt und Würden gelangte. Der 16-Bit-Protected-Mode muss daher aus heutiger Sicht als wenig bedeutender Meilenstein angesehen werden.

Befehle

Die Veränderungen im Befehlssatz des 80286 / 80287 waren geprägt durch den neuen Modus, den dieses Prozessorgespann neuerdings beherrschte: den protected mode. Daher haben *alle* neu eingeführten Befehle mit diesem Modus zu tun:

- ARPL
- CLTS
- LAR
- LGDT, LIDT, SGDT, SIDT
- LLDT, SLDT
- LMSW, RSMW
- LSL
- LTR, STR
- VERR, VERW

Der 80287 wurde ebenfalls um einen Protected-Mode-Befehl ergänzt, mit dem die NPX in den protected mode umgeschaltet werden konnte:

- FSETPM

In seinen Fähigkeiten erweitert wurde der Befehl

- F(N)STSW, der nun das status word direkt in das AX-Register schreiben konnte. Damit musste nicht mehr der umständliche Weg über eine Speicherstelle genommen werden (*FSTSW WordVar – MOV AX, WordVar*).

Aber es gab auch schon die ersten obsoleten Befehle: F(N)ENI und F(N)DISI, die für das Setzen bzw. Löschen des interrupt enable flag im control word des NPX dienten, wurden überflüssig (s.u.) und daher entfernt. Sie entsprechen heute einem FNOP!

Register Mit dem »machine status word« wurde dem 80286 ein Register implementiert, das als Vorläufer des heutigen Kontrollregisters CR0 aufgefasst werden kann. Es umfasste die ersten 16 Bits des CR0 (vgl. Seite 856).

Flags Durch die Einführung des machine status word wurden auch neue Flags implementiert:

- PE (Bit 0), *protection enable*; dieses Flag ist gesetzt, wenn der protected mode gesetzt ist, und schaltet in den real mode, wenn es gelöscht wird.
- MP (Bit 1), *monitor co-processor*; steuert zusammen mit TS die Reaktion der CPU bei Ausführung eines WAIT/FWAIT-Befehls, wenn keine FPU/NPX-Hardware verfügbar ist.
- EM (Bit 2), *emulation*; dieses Flag zeigt an, dass keine interne FPU oder externe NPX zur Verfügung steht und FPU-Befehle emuliert werden müssen.
- TS (Bit3), *task switched*; zeigt an, dass seit dem letzten FPU-Befehl ein task switch erfolgte und die FPU/NPX-Umgebung gesichert werden muss.

Die weiteren Änderungen an den Flags beziehen sich auf das status und control word des 80287.

- Bit 7 des status word heißt beim 8087 interrupt enable flag IE. Wurde es gesetzt, so führte das dazu, dass im status word Bit 7, das interrupt request flag IR gesetzt wurde, wenn eine numerische Exception auftrat. Welche das war, signalisierten damals wie heute die Bits 5 bis 0. Aufgrund einer unterschiedlichen Art der Zusammenarbeit zwischen CPU und NPX ab dem 80286/80287 entfiel die Notwen-

digkeit eines interrupt enable flags (s.u.). Bit 7 ist seit dem 80287 nicht definiert.

- Bit 7 des status word war entsprechend das interrupt request flag IR. War es gesetzt, so hieß das, dass die FPU eine exception ausgelöst hatte. Es fasst sozusagen die Bits 0 bis 5 zusammen: war mindestens eines gesetzt, so war es IR auch. Aufgrund der Änderungen der CPU-NPX-Zusammenarbeit hat sich zwar inzwischen die Funktion des interrupt request flags nicht geändert, wohl aber sein Name. Es heißt ab dem 80287 exception summary flag ES.

Exceptions

Die neu eingeführten Exceptions basierten auf dem neuen Betriebsmodus, dem protected mode. Mit der Erweiterung der Adressierbarkeit einher ging nämlich auch die Einführung von Schutzkonzepten (vgl. Seite 405) und die Verwaltung von virtuellem Speicher durch »segment swapping« sowie Multitasking-Fähigkeit. An neuen Exceptions gab es daher

- Invalid TSS #TS (Interrupt 10); ausgelöst, wenn im Rahmen des task switching auf ein Segment zugegriffen werden soll, das kein gültiges task state segment ist.
- Segment not present #NP (Interrupt 11); ausgelöst im Rahmen der virtuellen Speicherverwaltung, wenn das Segment, das angesprochen werden soll, nicht physikalisch verfügbar ist.
- General protection #GP (Interrupt 13); sie wird immer dann ausgelöst, wenn eine Verletzung der Zugriffsrechte im Rahmen der Schutzkonzepte festgestellt wird.
- Page fault #PF (Interrupt 14); sie wird ausgelöst, wenn nicht alle 16-Bit-Tasks das gleiche page directory verwendeten. Grund: Ein 16-Bit-TSS sichert nicht den Inhalt des Kontrollregisters CR3 (PDBR) und damit die Adresse der page directory.

Kompatibilität

Ein großer Teil der Inkompatibilitäten zum 8086 beruhen eindeutig auf dem neu eingeführten Betriebsmodus protected mode. Dies betrifft vor allem die Kooperation von CPU und NPX, wie wir sehen werden:

CPU - NPX

Die Zusammenarbeit beim Gespann 8086/8087 erfolgte im Falle numerischer Exceptions noch wie folgt: Die NPX stellte eine Ausnahmesituation fest und prüfte anhand der Masken-Bits im control word, ob eine Exception ausgelöst werden sollte. War das der Fall, musste das der CPU mitgeteilt werden. Das tat sie, indem sie einen nicht maskierbaren Interrupt (NMI) auslöste. Ob sie das letztendlich tun sollte oder nicht, entschied das interrupt enable flag IE im control word der NPX. Es wur-

de durch F(N)ENI, *floating-point enable interrupt*, gesetzt und durch F(N)DISI, *floating-point disable interrupt*, gelöscht.

Dieser NMI hat (bis heute) in der Interrupt-Verarbeitungslogik einen sehr hohen Stellenwert, wird also mit einer der höchsten Prioritäten behandelt und kann, anders als »normale« Interrupts, die im Interruptkontroller (»PIC«, programmable interrupt controller) maskiert werden können, auch nicht unterdrückt werden. Er hat nur einen Nachteil: Er ist unspezifisch, was heißt, dass jede Komponente, die einen NMI auslöst, *den* NMI auslöst! Das bedeutet, dass der Handler, der einen NMI zu behandeln hat, selbst feststellen muss, wer Auslöser des NMI war. Dazu diente IR. War es gesetzt, so wusste der Handler, dass der NPX die Quelle des NMIs war, und agierte entsprechend.

Durch die Einführung des protected mode mit dem 80286 musste auch die Zusammenarbeit CPU/NPX vollständig neu definiert werden. Daher wurde auch eine andere Methode implementiert, die CPU im Falle von NPX-Exceptions zu alarmieren: Intel spendierte einen eigenen PIN, mit dem die NPX die CPU in Kenntnis setzen konnte. Diese Methode wurde jedoch praktisch nicht angewendet. Stattdessen wurde der entsprechende Ausgang der NPX mit einem Eingang des PIC (»programmable interrupt controller«) verbunden. Eine NPX-Exception führte nun zu einem »normalen« (und daher über den PIC maskierbaren) Interrupt, dem Interrupt $75, der mit IRQ $13, dem Co-Prozessor-Eingang beim zweiten, kaskadierten PIC, verknüpft war. Dies aber bedeutete: Sowohl IE als auch die beiden dieses Flag steuernden Befehle F(N)ENI und F(N)DISI waren überflüssig und wurden eliminiert. Das »Exception-Summenflag« IR dagegen blieb, wenn es auch fortan unter neuem Namen segelte: exception summary flag ES.

PUSH Ab dem 80286 verhalten sich die Prozessoren anders, wenn ein PUSH SP ausgeführt werden soll. Während die Prozessoren vor dem 80286 den Inhalt von SP auf den Stack schieben, *nachdem* er dekrementiert wurde, wird er ab dem 80286 *vor* dem Dekrementieren auf den Stack geschoben! Auch wenn es heute, im Zeitalter der Pentium-4-Prozessoren und Itaniums, lächerlich klingt: Wenn Sie Kompatibilität zum 8086 in diesem Punkt benötigen, verwenden Sie folgende Sequenz anstelle von PUSH SP:

```
PUSH    BP
MOV     BP, SP
XCHG    BP, [BP]
```

Dies emuliert die Funktion der 8086-Version des PUSH-SP-Befehls (was im Zeitalter von 2-GHz-Pentium-4 nur lächerlich klingt!).

FLAGS

Die Zustände der Bits 15 bis 12 des FLAGS-Registers nach einem PUSHF, nach Rückkehr aus einem Interrupthandler oder nach Exceptions unterschied sich wie folgt:

- Beim 8086 sind die Bits 15 bis 12 grundsätzlich gesetzt.
- Beim 80286 sind die Bits 15 bis 12 im real mode grundsätzlich gelöscht.

5.9.9 80186/80188

Mit dem 80186 und seinem »Zwilling« 80188 gab es einen Prozessor, der nur sehr wenig bekannt wurde. Die »PC-Gemeinde« hat den Wechsel vom 8086 direkt zum 80286 vollzogen und es gibt wenige Menschen, die überhaupt wissen, dass es einen 80186 gab. Eine gewisse Bedeutung hat der 80186 auch weniger im PC-Bereich gehabt – ich hatte niemals einen PC mit einem solchen Prozessor in den Händen –, sondern im Bereich Spezialanwendungen. So hat die Firma Xerox in einigen Bürosystemen einen 80186 auf einer Erweiterungs-Steckkarte verwendet, um PC-Software auf diesem Bürosystem verfügbar zu machen. Und bis vor kurzer Zeit waren auch einige Steuerungsautomaten mit diesem Prozessor bestückt.

Befehle

Als neue, zusätzliche Befehle wurden mit dem 80186 implementiert:

- BOUND
- ENTER, LEAVE
- PUSHA, POPA
- INS, INSB, INSW, OUTS, OUTSB, OUTSW

Erweiterungen erfuhren die Befehle

- IMUL; nun darf einer der Operanden eine Konstante sein
- PUSH; auf den Stack können nun auch Konstanten gePUSHt werden
- RCL, RCR, ROL, ROR, SAL, SHL und SHR; auch bei diesen Befehlen sind nun Konstanten als Operator erlaubt, der die Anzahl der zu verschiebenden Positionen angibt.

Flags

Der 80186 definierte keine neuen Flags.

Exceptions

Der neu implementierte Befehl BOUND machte eine neue Exception erforderlich, die bound range exceeded exception #BR.

5.9.10 8086 / 8087

Der 8086 gilt als »Urvater« der heutigen PCs, obschon er nicht der erste Prozessor war, der eine bestimmte Bedeutung erlangt hat. Es handelte sich um einen 16-Bit-Prozessor, was bedeutet, dass er einen 16-Bit-Datenbus hatte und somit auch über 16-Bit-Register verfügte. Der 8086 verfügte über 20 Adressleitungen, was zu einem Adressraum von 2^{20} Bytes = 1.048.576 = 1 MByte führte.

Ein »Zwilling« des 8086 war der 8088, der bis auf einen 8-Bit-Datenbus mit dem 8086 identisch war.

Befehle Der 8086 implementierte folgende Befehle, die hier alphabetisch angeführt werden:

AAA, AAD, AAM, AAS, ADC, ADD, AND, CALL, CBW, CLC, CLD, CLI, CMC, CMP, CMPS, CMPSB, CMPSW, CWD, DAA, DAS, DEC, DIV, HLT, IDIV, IMUL, IN, INC, INT, INTO, INT3, IRET, Jcc, JMP, LAHF, LDS, LEA, LES, LOCK, LODS, LODSB, LODSW, LOOP, LOOPcc, MOV, MOVS, MOVSB, MOVSW, MUL, NEG, NOP, NOT, OR, OUT, POP, POPF, prefixes, PUSH, PUSHF, RCL, RCR, REP, REPcc, RET, ROL, ROR, SAHF, SAL, SAR, SBB, SCAS, SCASB, SCASW, SHL, SHR, STC, STD, STI, STOS, STOSB, STOSW, SUB, TEST, WAIT, XCHG, XLAT, XLATB, XOR.

Der 8087 implementierte die ebenfalls im Folgenden alphabetisch aufgelisteten Befehle:

FABS, FADD, FADDP, FBLD, FBSTP, FCHS, F(N)CLEX, FCOM, FCOMP, FCOMPP, FDECSTP, F(N)DISI, FDIV, FIVP, FDIVR, FDIVRP, F(N)ENI, FFREE, FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FILD, FIMUL, FINCSTP, F(N)INIT, FIST, FISTP, FISUB, FISUBR, FLD, FLDCW, FLDENV, FLDLG2, FLDLN2, FLDL2E, FLDL2T, FLDPI, FLDZ, FLD1, FMUL, FMULP, FNOP, FPATAN, FPREM, FPTAN, FRNDINT, FRSTOR, F(N)SAVE, FSCALE, FSQRT, FST, F(N)STCW, F(N)STENV, FSTP, F(N)STSW, FSUB, FSUBP, FSUBR, FSUBRP, FTST, FWAIT, FXAM, FXCH, FXTRACT, FYL2X, FYL2XP1, F2XM1.

Register Der 8086 implementierte folgende CPU-Register (vgl. auch die Register von 32-Bit-Prozessoren auf Seite 32).

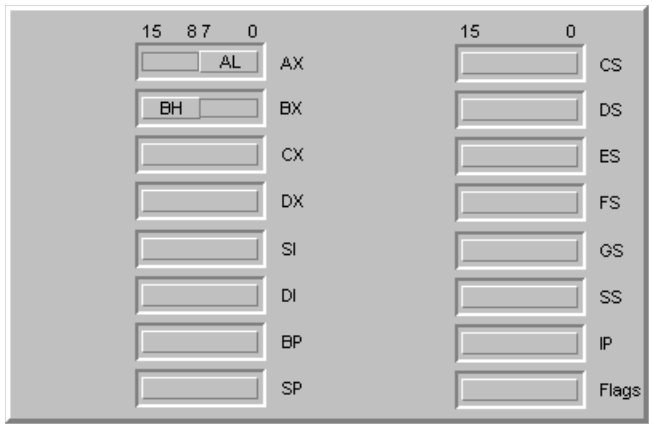


Abbildung 5.54: Die grundlegenden Register der 8086-CPU: Allzweck-, Segment-, Adressierungs- und Status-Register

Flags

Gemäß seiner Zugehörigkeit zu den 16-Bit-Prozessoren umfasste beim 8086 auch das Flag-Register »nur« 16 Bits. Die Funktion dieser Flags ist die gleiche wie die der entsprechenden Flags im heutigen EFlags-Register (vgl. Seite 38).

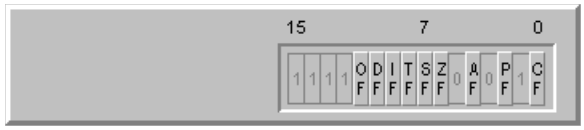


Abbildung 5.55: Das Flag-Register des 8086

Exceptions

Der 8086 definierte die in der folgenden Tabelle 5.33 dargestellten Exceptions. Sie entsprechen den korrespondierenden Exceptions der heutigen Prozessoren.

#		Beschreibung	Exception Interrupt	Ursache	Typ	Fehlercode
0	#DE	Divide Error	E	CPU	F	n
1	-	reserviert	-	-	-	-
2	-	Non-maskable Interrupt	I	H	-	n
3	#BP	Break Point	E	S	T	n
4	#OF	Overflow	E	S	T	n
5-31	-	reserviert	-	-	-	-
32-255	-	»frei verfügbare« Interrupts	I	S	-	-

Es bedeuten: E Exception; I Interrupt; H Hardware; S Software; F Fault; T Trap; n nein. Die grau unterlegten Interrupts/Exceptions galten als reserviert.

Tabelle 5.33: Die Exceptions des 8086

5.9.11 16-Bit-Protected-Mode

Im Folgenden werden einige Veränderungen dargestellt, die architekturbedingt im 16-Bit-Protected-Mode auftreten. Es handelt sich um die Deskriptoren, Gates und das task state segment.

16-Bit-Descriptor

Ein 16-Bit-Deskriptor unterscheidet sich von einem 32-Bit-Deskriptor nur dadurch, dass das höchste der vier Worte, aus denen ein Deskriptor besteht, als reserviert gilt und auf den Wert »0« gesetzt sein sollte. Wie Sie Abbildung 5.56 entnehmen können, stellt somit ein 16-Bit-Deskriptor ein 16-Bit-Limit (2^{16} = 65.536 Byte) zur Verfügung. Segmente können somit nicht größer als 64 kByte werden. Die Basisadresse umfasst 24 Bits und spiegelt damit die Voraussetzung auf einem 16-Bit-Prozessor (des Typs 80286!) wider. Die Flags P (present), S (system) und die Flags, die segmentspezifisch das Feld type zusammensetzen, haben die gleiche Bedeutung wie in 32-Bit-Deskriptoren (vgl. Abbildung 2.3 auf Seite 396), ebenso das Feld DPL. Es fehlen in 16-Bit-Deskriptoren die Flags G (granularity), D/B (default length bzw. big) und AVL (available), die auch nur in 32-Bit-Systemen Sinn machen. Die Positionen, an denen sie im 32-Bit-Deskriptor definiert sind (Bit 20, 22 und 23 des zweiten DoubleWords), sind in einem 16-Bit-Deskriptor auf Null gesetzt. Somit sind, falls ein 16-Bit-Deskriptor in 32-Bit-Systemen verwendet wird, automatisch die Flags G und D/B auf Null gesetzt (1-Byte-Granularität und 16-Bit-Segmente!).

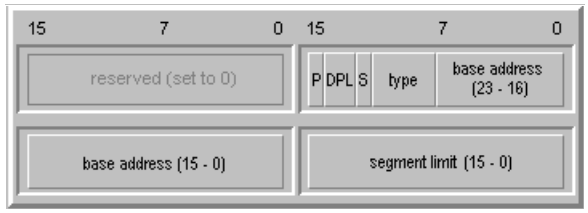


Abbildung 5.56: Speicherabbild eines 16-Bit-Deskriptors

16-Bit-Gate-Deskriptoren

Analoge Unterschiede finden sich auch in Gates. Abbildung 5.57 zeigt ein 16-Bit-Call-Gate. Beachten Sie bitte, dass auch in diesen Strukturen das höchste Word reserviert und auf Null gesetzt ist, weshalb der Offset in das Segment auch nur 16 Bits umfasst. Flag S (Bit 12 des dritten Words) und Flag D (Bit 11 des dritten Words) sind gelöscht, was sich als »16-Bit-Systemsegment« liest. Weitere Unterschiede zu 32-Bit-Gates bestehen nicht (vgl. Abbildung 2.10 auf Seite 422).

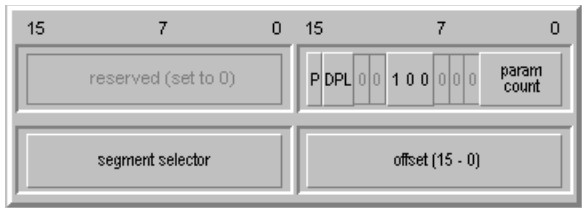


Abbildung 5.57: Speicherabbild eines 16-Bit-Gate-Deskriptors am Beispiel eines Call Gates

Task State segment

Die Bedeutung der Felder folgt der in 32-Bit-TSS (vgl. Seite 418) mit dem Unterschied, dass alle Register in 16-Bit-Systemen 16 Bit breit sind und die Inhalte des 16-Bit-TSS somit auch maximal Word-Breite haben. Beachten Sie bitte auch, dass es im 16-Bit-TSS keine Felder für eine I/O permission bit map oder interrupt redirection bit map gibt.

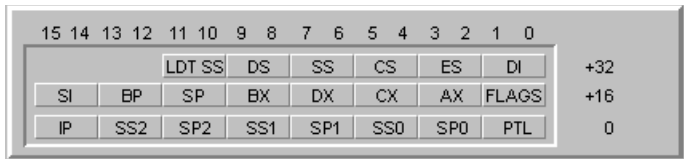


Abbildung 5.58: Speicherabbild des 16-Bit-Task-State-Segments



Dem TSS fehlt auch ein Feld für den Inhalt des PDB-Registers, in dem die Basisadresse der page directory steht. Somit können alle 16-Bit-Tasks bei tasks switches nur die gleiche page directory nutzen!



Beachten Sie bitte auch, dass ein 16-Bit-TSS nur die jeweils »unteren« 16 Bits der CPU-Register benutzt. Das bedeutet, dass auf 32-Bit-Prozessoren nur die »Word-Register« benutzt werden!

5.10 Verzeichnis der Abbildungen und Tabellen

5.10.1 Abbildungen

Abbildung 1.1: Die grundlegenden Register der CPU: Allzweck-, Segment-, Adressierungs- und Status-Register 32
Abbildung 1.2: Binäre Darstellung eines DoubleWords mit dem dezimalen Wert 53.416.551 35
Abbildung 1.3: Binäre Darstellung eines Words mit dem dezimalen Wert 4711 36
Abbildung 1.4: Binäre Darstellung eines Bytes mit dem dezimalen Wert 103 und einer BCD mit dem dezimalen Wert 7 36
Abbildung 1.5: Speicherabbild des EFlag-Registers 38
Abbildung 1.6: Status-, Kontroll- und Systemflags der CPU 40
Abbildung 1.7: Darstellung eines Überlaufs nach Addition zweier vorzeichenbehafteter Zahlen 42
Abbildung 1.8: Speicherabbild eines Bitfeldes als Ausgangssituation vor einem Bit-Schiebebefehl 75
Abbildung 1.9: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »einfachem« Verschieben nach rechts (SHR; oben) bzw. links (SHL; unten) 75
Abbildung 1.10: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »doppelt präzisem« Verschieben nach rechts (SHDR; oben) bzw. links (SHDL; unten) 76
Abbildung 1.11: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »arithmetischem« Verschieben nach rechts (SAR) 77
Abbildung 1.12: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »einfachem« Rotieren nach rechts (ROR) bzw. links (ROL) 77
Abbildung 1.13: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach Verschieben »über carry« nach rechts (RCL) bzw. links (RCL) 78
Abbildung 1.14: Speicherabbild des EAX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls bei Intel-Prozessoren 145
Abbildung 1.15: Speicherabbild des EAX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls bei AMD-Prozessoren 146
Abbildung 1.16: Speicherabbild des EBX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls bei Intel-Prozessoren 146
Abbildung 1.17: Speicherabbild des EDX-Registers nach Aufruf der Funktion 1 des CPUID-Befehls 147
Abbildung 1.18: Speicherabbilder der Inhalte der Register EAX, EBX, ECX und EDX nach Aufruf der Funktion 2 des CPUID-Befehls bei Intel-Prozessoren 150

Abbildung 1.19: Speicherabbild des Registers EAX nach Aufruf der Funktion 8001 des CPUID-Befehls bei AMD-Prozessoren 153
Abbildung 1.20: Speicherabbild des Registers EDX nach Aufruf der Funktion 8001 des CPUID-Befehls bei AMD-Prozessoren 154
Abbildung 1.21: Beispiel eines Speicherabbildes der Register EAX, EBX, ECX und EDX nach Aufruf des CPUID-Befehls mit den Funktionen 8002, 8003 und 8004 155
Abbildung 1.22: Speicherabbild der Register EAX und EBX nach Aufruf des CPUID-Befehls mit den Funktionen 8005 und 8006 bei AMD-Prozessoren 155
Abbildung 1.23: Speicherabbild der Register ECX und EDX nach Aufruf des CPUID-Befehls mit den Funktionen 8005 und 8006 bei AMD-Prozessoren 156
Abbildung 1.24: Speicherabbild des Registers EDX nach Aufruf der Funktion 8007 des CPUID-Befehls bei AMD-Prozessoren 157
Abbildung 1.25: Aufbau eines Stack-Rahmens durch ENTER mit einem nesting level von 0 159
Abbildung 1.26: Darstellung der durch den Befehl ENTER hergestellten stack frames 161
Abbildung 1.27: Entfernen eines Stack-Rahmens mittels LEAVE 164
Abbildung 1.28: *Die Register der FPU (»floating point unit«)* bzw. der NPX (*»numeric processing extension«, »arithmetic co-processor«*) 188
Abbildung 1.29: Speicherabbild des Tag-Registers der FPU 190
Abbildung 1.30: Speicherabbild des Status- und Kontrollregisters der FPU 191
Abbildung 1.31: Unterschiede der FPU-internen Zahlendarstellung gemäß der verschiedenen Werte für precision control 192
Abbildung 1.32: Korrespondenzen zwischen Statusflags der FPU (condition code) und Statusflags im EFlags-Register der CPU 196
Abbildung 1.33: Illustration der Arbeitsweise des FPU-Stacks mit Hilfe eines Stapels Umzugskisten 201
Abbildung 1.34: Speicherabbild der FPU-Register, ihre dynamische Ansprache und Zusammenhänge mit den Tag-, Status- und Kontrollregister der FPU 203
Abbildung 1.35: Die MMX-Register des Prozessors 276
Abbildung 1.36: Die XMM-Register des Prozessors 310
Abbildung 1.37: Speicherabbild der XMM-Register mit Integers im Rahmen Erweiterung des SIMD-Befehlssatzes unter SSE2 344
Abbildung 1.38: Speicherabbild der XMM-Register mit Realzahlen im Rahmen der Erweiterung des SIMD-Befehlssatzes unter SSE2 344
Abbildung 1.39: Das MXCS-Register 361
Abbildung 1.40: Die »3DNow!«-Register des Prozessors 368

Abbildung 2.1: Einrichtung eines stack frames 391
Abbildung 2.2: Entfernung eines stack frames 392
Abbildung 2.3: Speicherabbild eines Segment-Deskriptors 396
Abbildung 2.4: Speicherabbild eines Deskriptors, der ein als »not present« markiertes Segment beschreibt 399
Abbildung 2.5: Speicherabbild eines Codesegment-Deskriptors 409
Abbildung 2.6: Speicherabbild eines Datensegment-Deskriptors 412
Abbildung 2.7: Speicherabbild eines LDT-Segment-Deskriptors 418
Abbildung 2.8: Speicherabbild des Task State Segments 420
Abbildung 2.9: Speicherabbild eines TSS-Deskriptors 421
Abbildung 2.10: Speicherabbild eines Gate-Deskriptors 422
Abbildung 2.11: Speicherabbild eines Call-Gate-Deskriptors 423
Abbildung 2.12: Speicherabbild eines Interrupt-Gate-Deskriptors 425
Abbildung 2.13: Speicherabbild eines Trap-Gate-Deskriptors 425
Abbildung 2.14: Speicherabbild eines Task-Gate-Deskriptors 426
Abbildung 2.15: Speicherabbild eines Segment-Selektors 429
Abbildung 2.16: Speicherabbild eines »Nullselektors« 430
Abbildung 2.17: Speicherabbild des ErrorCodes, der im Rahmen von Exceptions dem exception handler übergeben wird 431
Abbildung 2.18: Speicherabbild der Systemregister des Prozessors 432
Abbildung 2.19: Speicherabbild eines Segmentregisters 433
Abbildung 2.20: Beziehung zwischen effektiver und physikalischer Adresse 434
Abbildung 2.21: Der Weg von der (relativen) effektiven zur (absoluten) physikalischen Adresse 434
Abbildung 2.22: *Berechnung einer virtuellen Adresse* aus einer *logischen* *439*
Abbildung 2.23: *Berechnung einer realen Adresse* aus einer *logischen* im real mode 440
Abbildung 2.24: Speicherabbild des PDBR im 32-Bit-Adressierungsmodus 444
Abbildung 2.25: Speicherabbild eines Page Directory Entry im 32-Bit-Adressierungsmodus 444
Abbildung 2.26: Speicherabbild eines Page-Table-Entry im 32-Bit-Adressierungsmodus 446
Abbildung 2.27: *Berechnung einer physikalischen Adresse* aus einer *virtuellen* unter Benutzung von 4-kByte-Pages im 32-Bit-Adressierungsmodus 446
Abbildung 2.28: Speicherabbild eines Page Table Entry und Page Directory Entry für eine Page, die nicht physikalisch verfügbar ist 447

Abbildung 2.29: Speicherabbild eines Page Directory Entry im 32-Bit-Adressierungsmodus bei Verwendung von 4-MByte-Pages 448
Abbildung 2.30: *Berechnung einer physikalischen Adresse* aus einer *virtuellen* unter Benutzung von 4-MByte-Pages im 32-Bit-Adressierungsmodus 448
Abbildung 2.31: Speicherabbild eines Page Directory Entry im PSE-36-Adressierungsmodus 449
Abbildung 2.32: Speicherabbild des PDPTR im PAE-Adressierungsmodus 450
Abbildung 2.33: Speicherabbild eines Page Directory Pointer Table Entry im PAE-Adressierungsmodus 451
Abbildung 2.34: Speicherabbild eines Page Directory Entry im PAE-Adressierungsmodus 452
Abbildung 2.35: Speicherabbild eines Page-Table-Entry im PAE-Adressierungsmodus bei Verwendung von 4-kByte-Pages 452
Abbildung 2.36: *Berechnung einer physikalischen Adresse* aus einer *virtuellen* unter Benutzung von 4-kByte-Pages im PAE-Adressierungsmodus 453
Abbildung 2.37: Speicherabbild eines Page Directory Entry im PAE-Adressierungsmodus bei Verwendung von 2-MByte-Pages 454
Abbildung 2.38: *Berechnung einer physikalischen Adresse* aus einer *virtuellen* unter Benutzung von 2-MByte-Pages im PAE-Adressierungsmodus 454
Abbildung 2.39: *Umsetzung einer effektiven Adresse* in eine *physikalische* durch Speichersegmentierung und Paging mit 4-kByte-Pages im 32-Bit-Adressierungsmodus 456
Abbildung 2.40: Aufteilung des virtuellen 4-GByte-Speichers unter Windows 98 und Windows 2000 459
Abbildung 2.41: Berechnung der Adresse eines Interrupt-Handlers im real mode 491
Abbildung 2.42: Berechnung der Adresse eines Interrupt-Handlers im Protected Mode 494
Abbildung 2.43: Interrupt-Behandlung über ein Task Gate 495
Abbildung 2.44: Speicherabbild der bei Exceptions verwendeten Fehlercodes 496
Abbildung 2.45: Speicherabbild des bei einer page fault exception #PF verwendeten Fehlercodes 497
Abbildung 3.1: Speicherabbild des Strings "Dies ist ein ASCII-String" 564
Abbildung 3.2: Speicherabbild des QuadWord-Strings "Hi, Fan!" 565
Abbildung 3.3: Speicherabbild des Strings "Das ist ein Test" mit verschiedenen Datentypen 565

Abbildung 5.1: Zusammenhang zwischen binärer und hexadezimaler Darstellung 786
Abbildung 5.2: Bedeutung der Bits bei der Codierung von Fließkommazahlen 788
Abbildung 5.3: Speicherabbild des Wertebereiches valider finiter Zahlen am Beispiel einer SingleReal 791
Abbildung 5.4: Speicherabbilder einer »Infiniten« und des Wertes Null am Beispiel einer SingleReal 791
Abbildung 5.5: Speicherabbild einer »Indefiniten« am Beispiel einer SingleReal 792
Abbildung 5.6: Speicherabbild einer »quiet NaN« am Beispiel einer SingleReal 792
Abbildung 5.7: Speicherabbild einer »signalling NaN« am Beispiel einer SingleReal 792
Abbildung 5.8: Speicherabbildung einer Denormalen am Beispiel einer SingleReal 794
Abbildung 5.9: Speicherabbild einer ExtendedReal 799
Abbildung 5.10: Speicherabbild einer DoubleReal 800
Abbildung 5.11: Speicherabbild einer SingleReal 801
Abbildung 5.12: Bedeutung der Bits bei der Codierung von Integers 801
Abbildung 5.13: Speicherabbild einer »Indefiniten Integer« am Beispiel einer LongInt 803
Abbildung 5.14: Speicherabbild eines vorzeichenlosen DoubleQuadWord (oben) und einer vorzeichenbehafteten DoubleQuadInt (unten) 806
Abbildung 5.15: Speicherabbild eines vorzeichenlosen DoubleQuadWord (oben) und einer vorzeichenbehafteten QuadInt (unten) 807
Abbildung 5.16: Speicherabbild eines vorzeichenlosen DoubleWord (oben) und einer vorzeichenbehafteten LongInt (unten) 808
Abbildung 5.17: Speicherabbild eines vorzeichenlosen Word (oben) und einer vorzeichenbehafteten SmallInt (unten) 808
Abbildung 5.18: Speicherabbild eines vorzeichenlosen Byte (oben) und einer vorzeichenbehafteten ShortInt (unten) 809
Abbildung 5.19: Speicherabbild einer ungepackten CPU-BCD 810
Abbildung 5.20: Speicherabbild einer gepackten CPU-BCD 810
Abbildung 5.21: Speicherabbild einer FPU-BCD 810
Abbildung 5.22: Speicherabbild einer »Indefiniten BCD« 811
Abbildung 5.23: Speicherabbild eines ShortPackedByte (oben) und einer ShortPackedShortInt 812
Abbildung 5.24: Speicherabbild eines ShortPackedWord (oben) und einer ShortPackedSmallInt 812

Abbildung 5.25: Speicherabbild eines ShortPackedDoubleWord (oben) und einer ShortPackedLongInt 812
Abbildung 5.26: Speicherabbild einer ShortPackedSingle 813
Abbildung 5.27: Speicherabbild eines PackedByte (oben) und einer PackedShortInt 813
Abbildung 5.28: Speicherabbild eines PackedWord (oben) und einer PackedSmallInt 813
Abbildung 5.29: Speicherabbild eines PackedDoubleWord (oben) und einer PackedLongInt 813
Abbildung 5.30: Speicherabbild eines PackedQuadWord (oben) und einer PackedQuadInt 814
Abbildung 5.31: Speicherabbild einer PackedSingle 814
Abbildung 5.32: Speicherabbild einer PackedDouble 814
Abbildung 5.33: Speicherabbild einer ScalarSingle 815
Abbildung 5.34: Speicherabbild einer ScalarDouble 815
Abbildung 5.35: Beispielhafte Belegung des I/O-Adressraums mit Peripheriegeräten 829
Abbildung 5.36: Flussdiagramm zur Decodierung des ModR/M- und SIB-Bytes einer Befehlssequenz 843
Abbildung 5.37: Speicherabbild des Kontrollregisters 0 856
Abbildung 5.38: Speicherabbild des Kontrollregisters 1 859
Abbildung 5.39: Speicherabbild des Kontrollregisters 2 859
Abbildung 5.40: Speicherabbild des Kontrollregisters 3 (PDBR) 860
Abbildung 5.41: Speicherabbild des Kontrollregisters 4 860
Abbildung 5.42: CR4-Pendants in den feature flags des CPUID-Befehls 862
Abbildung 5.43: Speicherabbild der Debugregister 0 bis 3 863
Abbildung 5.44: Speicherabbild der Debugregister 4 und 5 863
Abbildung 5.45: Speicherabbild des Debug-Registers 6 864
Abbildung 5.46: Speicherabbild des Debug-Registers 7 865
Abbildung 5.47: Speicherabbild der durch FXSAVE/FXRSTOR verwalteten Daten 868
Abbildung 5.48: Speicherabbild der durch F(N)SAVE/FRSTOR im 32-Bit-Protected-Mode verwalteten Daten 871
Abbildung 5.49: Speicherabbild der durch F(N)SAVE/FRSTOR im 32-Bit-Real-Mode verwalteten Daten 871
Abbildung 5.50: Speicherabbild der durch F(N)SAVE/FRSTOR im 16-Bit-Protected-Mode verwalteten Daten 872
Abbildung 5.51: Speicherabbild der durch F(N)SAVE/FRSTOR im 16-Bit-Real-Mode verwalteten Daten 872
Abbildung 5.52: Speicherabbild der durch FSTENV/FLDENV in verschiedenen Betriebsmodi und Umgebungen verwalteten Daten 873

Abbildung 5.53: Bildung des partiellen Tangens 887
Abbildung 5.54: Die grundlegenden Register der 8086-CPU: Allzweck-, Segment-, Adressierungs- und Status-Register 897
Abbildung 5.55: Das Flag-Register des 8086 897
Abbildung 5.56: Speicherabbild eines 16-Bit-Deskriptors 898
Abbildung 5.57: Speicherabbild eines 16-Bit-Gate-Deskriptors am Beispiel eines Call Gates 899
Abbildung 5.58: Speicherabbild des 16-Bit-Task-State-Segments 899

5.10.2 Tabellen

Tabelle 1.1: Mnemonics für die Kombination bestimmter Statusflags nach vergleichenden Befehlen 43
Tabelle 1.2: Explizite und implizite Operanden des MUL-/IMUL-Befehls 49
Tabelle 1.3: Explizite und implizite Operanden des DIV-/IDIV-Befehls 52
Tabelle 1.4: Darstellung der Bitstellungen nach den logischen Operationen AND, OR, XOR und NOT 64
Tabelle 1.5: Bedingte Sprungbefehle und die mit ihnen verbundenen Prüfungen der Statusflags 104
Tabelle 1.6: Bedingte LOOP-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags 106
Tabelle 1.7: CMOVcc-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags 117
Tabelle 1.8: Bedingte SET-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags 119
Tabelle 1.9: Dem brand index aus Funktion 1 des CPUID-Befehls zugeordnete brand strings 147
Tabelle 1.10: Codes für Eigenschaften von caches und translation look-aside buffers (TLBs) des Prozessors 151
Tabelle 1.11: Werte des Feldes associativity nach Aufruf der Funktion $8006 und ihre Bedeutung 156
Tabelle 1.12: Ergebnisse des Befehls FADD mit unterschiedlichen Werten für Quell- und Zieloperand 206
Tabelle 1.13: Ergebnisse des Befehls FSUB mit unterschiedlichen Werten für Quell- und Zieloperand 207
Tabelle 1.14: Ergebnisse des Befehls FMUL mit unterschiedlichen Werten für Quell- und Zieloperand 207
Tabelle 1.15: Ergebnisse des Befehls FDIV mit unterschiedlichen Werten für Quell- und Zieloperand 208
Tabelle 1.16: Ergebnisse des Befehls FIADD mit unterschiedlichen Werten für Quell- und Zieloperand 209

Tabelle 1.17: Ergebnisse des Befehls FISUB mit unterschiedlichen Werten für Quell- und Zieloperand 209
Tabelle 1.18: Ergebnisse des Befehls FIMUL mit unterschiedlichen Werten für Quell- und Zieloperand 209
Tabelle 1.19: Ergebnisse des Befehls FIDIV mit unterschiedlichen Werten für Quell- und Zieloperand 210
Tabelle 1.20: Ergebnisse des Befehls FSQRT mit unterschiedlichen Werten als Eingaben 212
Tabelle 1.21: Ergebnisse des Befehls FABS mit unterschiedlichen Werten als Eingaben 213
Tabelle 1.22: Ergebnisse des Befehls FCHS mit unterschiedlichen Werten als Eingaben 214
Tabelle 1.23: Ergebnisse der Befehle FPREM und FPREM1 mit unterschiedlichen Werten für Divisor und Dividenden 217
Tabelle 1.24: Ergebnisse der Befehle FSIN und FCOS mit unterschiedlichen Argumenten 219
Tabelle 1.25: Ergebnisse des Befehls FPTAN mit unterschiedlichen Argumenten 222
Tabelle 1.26: Ergebnisse des Befehls FPATAN mit unterschiedlichen Argumenten 224
Tabelle 1.27: Ergebnisse des Befehls FYL2X mit unterschiedlichen Argumenten 227
Tabelle 1.28: Ergebnisse des Befehls FYL2XP1 mit unterschiedlichen Argumenten. C = (1 - √2/2) 227
Tabelle 1.29: Ergebnisse des Befehls F2XM1 mit unterschiedlichen Argumenten 229
Tabelle 1.30: Stellung der Flags des Condition Code nach Vergleichen mit FCOM/FCOMP/FCOMPP 232
Tabelle 1.31: Stellung der Flags des Condition Code nach Vergleichen mit FICOM/FICOMP 234
Tabelle 1.32: Stellung der Flags des Condition Code nach Vergleichen mit FCOMI/FCOMIP/FUCOMI/FUCOMIP 236
Tabelle 1.33: Stellung der Flags des Condition Code nach Vergleichen mit FTST 237
Tabelle 1.34: Stellung der Flags des Condition Code nach Vergleichen mit FTST 238
Tabelle 1.35: Ergebnisse nach FIST/FISTP mit unterschiedlichen Werten als Eingaben 244
Tabelle 1.36: Ergebnisse nach FBSTP mit unterschiedlichen Werten als Eingaben 246
Tabelle 1.37: FCMOVcc-Befehle und die mit ihnen verbundenen Prüfungen der Statusflags 249
Tabelle 1.38: Ergebnisse des Befehls FSCALE mit unterschiedlichen Argumenten 252

Tabelle 1.39: Darstellung der Bitstellungen nach den logischen Operationen PAND, PANDN, POR und PXOR 295
Tabelle 1.40: »Prädikate« der Befehle CMPPS und CMPSS und ihre Bedeutung 326
Tabelle 1.41: Mögliche Makronamen für die Realisierung Prädikat-unabhängiger Vergleichsbefehle unter SSE 326
Tabelle 1.42: Stellung einiger Condition Code im EFlags-Register der CPU und ihre Bedeutung bei den Befehlen COMISS und UCOMISS 329
Tabelle 2.1: Flags und Flagstellungen zur Definition des verwendeten Paging-Modus und der Größen der Pages 455
Tabelle 2.2: Liste der möglichen Exceptions und Interrupts im protected mode. 500
Tabelle 2.3: Prioritätenliste für die Bearbeitung von Exceptions 502
Tabelle 2.4: Ausrichtung von Daten 524
Tabelle 2.5: Prioritätenliste für die Bearbeitung von numerischen Exceptions 531
Tabelle 2.6: Liste der möglichen FPU-Exceptions 532
Tabelle 2.7: Prioritätenliste für die Bearbeitung von numerischen Exceptions 544
Tabelle 2.8: Liste der möglichen SSE/SSE2-Exceptions 545
Tabelle 2.9: Liste der möglichen Exceptions und Interrupts im virtual 8086 und real mode 552
Tabelle 3.1: Ergebnis der Verwendung der Komponenten einer structure mit und ohne Operatoren 582
Tabelle 3.2: Ergebnis der Verwendung der Komponenten einer union mit und ohne Operatoren 588
Tabelle 3.3: Ergebnis der Verwendung der Komponenten eines Records mit und ohne Operatoren 593
Tabelle 3.4: Ergebnis der Verwendung der Komponenten eines Aufzählungstypen mit und ohne Operatoren 597
Tabelle 3.5: Konventionen, nach denen Hochsprachen-Prozeduren eingerichtet werden 612
Tabelle 3.6: Nach Konventionen üblicherweise benutzte Werte für bestimmte Segmentattribute 632
Tabelle 3.7: Standardwerte, die Anzahl von Code- und Datensegmenten, Umgebung, Code- und Datenpointer und ASSUME-Argumente in Abhängigkeit des gewählten Programmiermodells 640
Tabelle 3.8: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .CODE 644

Tabelle 3.9: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .CONST 645
Tabelle 3.10: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .DATA 646
Tabelle 3.11: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .DATA? 647
Tabelle 3.12: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .FARDATA 648
Tabelle 3.13: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .FARDATA? 650
Tabelle 3.14: Standardwerte für Segmentattribute in Abhängigkeit des gewählten Programmiermodells für die vereinfachte Segmentdeklaration mittels .STACK 651
Tabelle 3.15: Variationen der Direktiven IF und ELSEIF für die bedingte Assemblierung 663
Tabelle 3.16: Variationen der Direktiven .ERR für die Bedingte Fehlergenerierung 665
Tabelle 3.17: Direktiven zur Steuerung von Listings, ihre Synonyme in älteren MASM-Versionen und ihre Entsprechungen im MASM- und IDEAL-Modus von TASM 671
Tabelle 3.18: Direktiven zur Anwahl des Befehlssatzes und ihre Auswirkungen auf den verfügbaren Befehlssatz 672
Tabelle 3.19: TASM-spezifische Direktiven zur Anwahl des Befehlssatzes und ihre Auswirkungen auf den verfügbaren Befehlssatz 673
Tabelle 3.20: MASM-spezifische Direktiven zur Anwahl des Befehlssatzes und ihre Auswirkungen auf den verfügbaren Befehlssatz 675
Tabelle 3.21: MASMs Warnstufen 677
Tabelle 3.22: TASMs Warnklassen 678
Tabelle 3.23: Konventionen verschiedener Sprachinterfaces 723
Tabelle 3.24: Übergabekonvention von Funktionswerten 725
Tabelle 5.1: Einteilung der Befehlspräfixe in Gruppen 769
Tabelle 5.2: Bitstellungen und deren Bedeutung bei Fließkommazahlen 796
Tabelle 5.3: Nach IEEE Std. 754 nicht unterstützte »Pseudo«-Fließkommazahlen 798
Tabelle 5.4: Charakteristika einer ExtendedReal 799
Tabelle 5.5: Charakteristika einer DoubleReal 800
Tabelle 5.6: Charakteristika einer SingleReal 801
Tabelle 5.7: Charakteristika eines DoubleQuadWords und einer DoubleQuadInt 806

Tabelle 5.8: Charakteristika eines QuadWords und einer QuadInt 807
Tabelle 5.9: Charakteristika eines DoubleWords und einer LongInt 808
Tabelle 5.10: Charakteristika eines Words und einer SmallInt 809
Tabelle 5.11: Charakteristika eines Bytes und einer ShortInt 809
Tabelle 5.12: Charakteristika verschiedener BCDs 811
Tabelle 5.13: Gegenüberstellung der in diesem Buch und den in Assembler, Delphi und C++ benutzten Bezeichnungen für Daten 816
Tabelle 5.14: Ein-Byte-Opcodes 834
Tabelle 5.15: Zwei-Byte-CPU- und SIMD-Opcodes. Das erste Byte ist immer $0F. 835
Tabelle 5.16: »Drei-Byte«-SIMD-Opcodes mit führendem Präfix $F3 836
Tabelle 5.17: »Drei-Byte«-SIMD-Opcodes mit führendem Präfix $66 und erstem Opcode-Byte $0F 837
Tabelle 5.18: »Drei-Byte«-SIMD-Opcodes mit führendem Präfix $F2 und erstem Opcode-Byte $0F 838
Tabelle 5.19: Spec-Feld im ModR/M-Byte der Befehle der Gruppen 1 bis 16 839
Tabelle 5.20: Ein-Byte-FPU-Opcodes 840
Tabelle 5.21: Zwei-Byte-FPU-Opcodes 841
Tabelle 5.22: »Drei-Byte«-Opcodes für die 3DNow!-Befehle der AMD-Prozessoren 842
Tabelle 5.23: Unter den einzelnen Erweiterungen der SIMD-Technik von Intel (MMX, SSE, SSE2) verfügbare Datenformate 844
Tabelle 5.24: Intel-Nomenklatur für die unter SIMD verwendeten Datenformate und deren in diesem Buch verwendeten Pendants 845
Tabelle 5.25: SSE2-Instruktionen zur Konvertierung von Fließkommazahlen in Integers und umgekehrt 846
Tabelle 5.26: Übersicht der unter SIMD der Intelprozessoren verfügbaren Befehle 847
Tabelle 5.27: Unter der 3DNow!-Technik verfügbare Datenformate 850
Tabelle 5.28: AMD-Nomenklatur für die unter SIMD verwendeten Datenformate und deren in diesem Buch verwendeten Pendants 851
Tabelle 5.29: Übersicht der unter SIMD der AMD-Prozessoren verfügbaren Befehle 852
Tabelle 5.30: Der 3DNow!-Technologie von AMD vergleichbare Intel-Instruktionen unter SSE/SSE2 855
Tabelle 5.31: Unterschiede in den instruction sets der SIMD-Befehle von AMD und Intel 855
Tabelle 5.32: Oktanten-abhängige Berechnung der trigonometrischen Funktionen. C = $\pi/4$ 889
Tabelle 5.33: Die Exceptions des 8086 897

5.11 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	

Stichwortverzeichnis

32-Bit-Adressierungsmodus 444
32-Bit-Windows 458
3DNow! 144, 368
3DNow!-Befehlssatz 369
3DNow!-Codierung 771
3DNow!-Professional 382
3DNow!-Register 368
3DNow!-X 379
64-Bit-Windows 462

A
Aborts 501
absolut privilegiert 182
absolute Adressen 436
Accumulator 33
Address Size 766
Address Size Override 137
Adressgröße 765, 818
Adressraum 467
Alias-Namen 34
Allzweckregister 33
AMD 144
ANSI 127
Anweisungen siehe Direktiven
Anwendung 773
ASCII 127
Assembler 749
– Anweisungen siehe Direktiven
– Befehle siehe Instruktionen
Assembler-Routinen 752
Aufrufkonventionen 723
Auslagerungsdatei 442, 457
Ausnahmesituation 777

B
Base Pointer register 33
Base register 33
Basisadresse 396
BCD 36, 803
– CPU-~ 809
– FPU-~ 810
– gepackte ~ 804
BCD-Korrekturen 53
bedingt privilegiert 182
Befehle siehe auch Instruktionen
– CPU-~ 44
– FPU-~ 204
– XMM-~ 317
Befehlspräfix siehe Präfixe
Befehlssemantik 763
Befehlssequenz 768
Befehlszeiger 37
bias 790
biased exponent 789
biasing constant 790
Big-Endian-Format 781
binäre Zahlendarstellung 782, 803
Binary Coded Decimals 809
– siehe auch BCD
Branch Hints 140
Byte 809

C
call gate 422
Codesegment 394, 408, 477, 766
– conforming 410
– Deskriptoren 408
– non-conforming 410
Codierung
– Fließkommazahlen 788
– Integers 801
commitment to task switch 466
Compiler
– #pragma link 723
– Schalter {$L} 721

– Schalter {$Link} 721
– Schalter {$O} 754
– Schalter {$OPTIMIZATION} 754
Condition Code 195
Consumer-Windows 469
Control Word 191
Counter register 33
CPL 470
current privileg level 470
Customer-Windows 458

D
Data register 33
Daten in Codesegmenten 476
Datenformate 778
– Byte 780
– CPU-~ 30
– DoubleQuadInt 780
– DoubleQuadWord 780
– DoubleReal 780
– DoubleWord 780
– ExtendedReal 780
– FPU-~ 188
– LongInt 780
– OctelInt 780
– OctelWord 780
– Packed 780
– QuadInt 780
– QuadWord 780
– ShortInt 780
– ShortPacked 780
– SingleReal 780
– SmallInt 780
– Word 780
Datensegment 394, 411, 474, 766
– Deskriptoren 411
– expand-down 413
– expand-up 413
Decodierung 832
– Adresse 833
– Konstante 833
– ModR/M-Byte 833
– Opcode 832
– Präfix 832
– SIB-Byte 833
Denormale 794
descriptor privileg level 470
Deskriptor-Tabellen 417
Direkte Adressierung 818
Direktiven 561
– 676
– %BIN 669
– %CLTS 671
– %CREFALL 671
– %CREFREF 671
– %CREFUREF 671
– %DEPTH 669
– %INCL 672
– %LINUM 669
– %NOCLTS 671
– %NOINCL 672
– %NOSYMS 671
– %NOTRUNC 669
– %OUT 676
– %PCNT 669
– %POPCTL 688
– %PUSHCTL 688
– %SYMS 671
– %TABSIZE 669
– %TEXT 669
– %TRUNC 669
– .186 672
– .286 672
– .286C 674
– .286P 672
– .287 672
– .386 672
– .386C 674
– .386P 672
– .387 672
– .486 672
– .486C 674
– .486P 672
– .487 674
– .586 672
– .586C 674
– .586P 672
– .587 674
– .686 675
– .686P 675
– .8086 672
– .8087 672
– .ALPHA 638
– .BREAK 654
– .CODE 643, 718
– .CONST 644
– .CONTINUE 654
– .CREF 669

– .DATA 645, 716
– .DATA? 646
– .DOSSEG 638
– .ELSE 652
– .ELSEIF 652
– .ENDIF 652
– .ENDWHILE 653
– .ERR 664
– .ERRcc 664
– .EXIT 651
– .FARDATA 647
– .FARDATA? 649
– .IF 652
– .K3D 675
– .LIST 669
– .LISTALL 669
– .LISTIF 669
– .LISTMACRO 669
– .LISTMACROALL 669
– .MMX 675
– .MODEL 639
– .Model 715
– .NO87 672
– .NOCREF 669
– .NOLIST 669
– .NOLISTIF 669
– .NOLISTMACRO 669
– .RADIX 685
– .REPEAT 653
– .SEQ 638
– .STACK 650
– .STARTUP 651
– .TFCOND 669
– .UNTIL 653
– .UNTILCXZ 653
– .WHILE 653
– .XMM 675
– = 599
– ALIAS 603
– ALIGN 604
– ARG 618
– ASSUME 635
– BYTE 567
– CALL 619
– CATSTR 599
– COMM 624
– COMMENT 676
– DB 568
– DD 568
– DF 569
– DISPLAY 677
– DP 569
– DQ 568
– DT 568
– DW 568
– DWORD 567
– ECHO 676
– ELSE 662
– ELSEIF 662
– ELSEIFcc 663
– EMUL 686
– END 638, 719
– ENDIF 662
– ENDM 656
– ENDP 611
– ENDS 570, 584, 627
– ENUM 595
– EQU 598
– EVEN 605
– EVENDATA 606
– EXITM 659
– EXTERN 621
– EXTERNDEF 622
– EXTRN 621, 717
– FASTIMUL 689
– FLIPFLAG 690
– FOR 661
– FORC 661
– FWORD 569
– GETFIELD 689
– GLOBAL 623
– GOTO 660
– GROUP 632
– IDEAL 686
– IF 662
– IFcc 663
– INCLUDE 626
– INCLUDELIB 626
– INSTR 599
– INVOKE 618
– IRP 661
– IRPC 662
– JMP 621
– JUMPS 686
– LABEL 600
– LARGESTACK 652
– LOCAL 616, 659
– LOCALS 624

– MACRO 656
– MASKFLAG 690
– MASM 686
– MASM51 687
– MULTERRS 679
– NOEMUL 686
– NOJUMPS 686
– NOLOCALS 624
– NOMASM51 687
– NOMULTERRS 679
– NOSMART 687
– NOWARN 677
– OPTION 679
– ORG 606
– OWORD 567
– P186 673
– P286 673
– P286N 673
– P286P 673
– P287 673
– P386 674
– P386N 674
– P386P 674
– P387 674
– P486 674
– P486N 674
– P486P 674
– P586 674
– P586N 674
– P586P 674
– P8086 673
– P8087 673
– PAGE 668
– PNO87 674
– POPCONTEXT 688
– POPSTATE 688
– PROC 611
– PROCDESC 615
– PROCTYPE 595
– PROTO 614, 718
– PUBLIC 621, 717
– PUBLICDLL 626
– PURGE 658
– PUSHCONTEXT 688
– PUSHSTATE 688
– PWORD 569
– QUIRKS 687
– QWORD 567
– RADIX 686
– REAL10 568
– REAL4 568
– REAL8 568
– RECORD 589
– REPEAT 660
– REPT 660
– RETCODE 689
– RETF 689
– RETN 689
– SBYTE 568
– SDWORD 568
– SEGMENT 627
– SETFIELD 689
– SETFLAG 690
– SIZESTR 599
– SMALLSTACK 652
– SMART 687
– STRUC 570
– STRUCT 570
– SUBSTR 599
– SUBTITLE 667
– SWORD 568
– TABLE 597
– TBLINIT 597
– TBLINST 597
– TBLPTR 597
– TBYTE 568
– TESTFLAG 690
– TEXTEQU 599
– TITLE 667
– TYPEDEF 594
– UNION 584
– USES 617
– VERSION 686
– WARN 677
– WHILE 661
– WORD 567
displacement 770
DOS 397, 460, 766
DoubleQuadInt 806
DoubleQuadWord 806
DoubleReal 800
DoubleWord 808
DPL 470

E
effektive Adresse 435
EFlags-Register 38
Einsprungpunkte 479

Elementardaten 788
Epilog 750
Error Code 431, 489, 496
exception dispatcher 731
Exception Flags 195
Exception-Emulation 498
Exceptionklassen 500
Exception-Masken 191
Exceptions 120, 185, 270, 489, 777, 792
– 3DNow!-/3DNow!-X-~ 382
– Alignment Check (#AC) 187, 524
– benigne 501
– Bound Range Exceeded (#BR) 185, 506
– Break Point (#BP) 185, 504
– contributory 501
– Coprocessor Segment Overrun 510
– CPU-~ 183, 489, 499
– Debug (#DB) 185, 503
– Denormalized Operand (#D) 271, 536
– Device Not Available (#NM) 186, 508
– Divide By Zero (#Z) 271, 537
– Divide Error (#DE) 185, 502
– Double Fault (#DF) 186, 510
– Extension Fault (#XF) 187, 526
– FPU Error (#MF) 187, 521
– FPU-~ 268, 529
– General Protection (#GP) 186, 516
– Hardware-~ 489
– Inexact Result (#P) 271, 541
– Invalid Arithmetic Operand (#IA) 271, 533
– Invalid Opcode (#UD) 186, 506
– Invalid Operation (#I) 270, 532
– Invalid Stack Operand (#IS) 271, 533
– Invalid TSS (#TS) 186, 511
– Machine Check (#MC) 187, 526
– Maskierte ~ 269
– Math Fault (#MF) 187, 521
– MMX-~ 306
– No Math Coprocessor (#NM) 186, 508
– Non-maskable Interrupt (NMI) 504
– Numeric Overflow (#O) 271, 538
– Numeric Underflow (#U) 271, 540
– Overflow (#OF) 185, 505
– page fault 502
– Page Fault (#PF) 187, 445, 519
– page fault (#PF) 497
– Precision (#P) 271, 541
– Prioritäten 502, 531, 544
– Real Mode 552
– Segment Not Present (#NP) 186, 513
– SIMD Floating Point Error (#XF) 187, 526
– Software-~ 489
– SSE-/SSE2-~ 360
– Stack Segment Fault (#SS) 186, 515
– Undefined Opcode (#UD) 186, 506
– Unmaskierte ~ 363
– Virtual 8086 Mode ~ 553
Exception-Synchronisierung 530
Exceptiontypen 184, 270, 501
ExtendedReal 799

F
Faults 500
Fehlercode 489, 496
Flags
– adjust flag 41
– auxiliary carry flag 41
– busy flag 195
– carry flag 40
– DAZ 362
– direction flag 131
– FZ 362
– granularity flag 397
– ID-Flag 40
– Kontrollflag 131
– OSXMMEXCEPT 363
– overflow flag 41
– parity flag 40
– sign flag 41
– zero flag 41
Flat Model 394
FPU-Stack 200
Funktionen 777

G
Gate-Deskriptoren 422
Gates 421

GDT siehe global descriptor table
global descriptor table 417, 427
global unwind 732

H
Hardware-Interrupts 487
Heap 388
Hexadezimalsystem 782
Hexadezimal-Ziffern 786

I
I/O permission bit map 485
I/O privileg level 483
Image 442
immediate 770
indefinite 792, 803
indirekte Adressierung 818
Infinite 791
Infinity Control 194
Inline 749
Inline-Assembler 557, 745
instruction pointer 37
Instruktionen
– AAA 57
– AAD 57
– AAM 57
– AAS 57
– ADC 53
– ADD 45
– ADDPD 345
– ADDPS 319
– ADDSD 345
– ADDSS 319
– AND 64
– ANDNPD 345
– ANDNPS 323
– ANDPD 345
– ANDPS 323
– ARPL 174
– BOUND 124
– BSF 85
– BSR 85
– BSWAP 90
– BT 82
– BTC 82
– BTR 82
– BTS 82
– CALL 107
– CBW 99
– CDQ 99
– CLC 127
– CLD 127
– CLFLUSH 358
– CLI 127
– CLTS 178
– CMC 127
– CMOVcc 117
– CMP 69
– CMPPD 346
– CMPPS 325
– CMPS 132
– CMPSB 132
– CMPSD 132, 346
– CMPSS 325
– CMPSW 132
– CMPXCHG 92
– CMPXCHG8B 92
– COMISD 346
– COMISS 328
– CPUID 143
– CVTDQ2PD 349
– CVTDQ2PS 349
– CVTPD2DQ 349
– CVTPD2PI 347
– CVTPD2PS 350
– CVTPI2PD 347
– CVTPI2PS 335
– CVTPS2DQ 349
– CVTPS2PD 350
– CVTPS2PI 335
– CVTSD2SI 347
– CVTSD2SS 350
– CVTSI2SD 347
– CVTSI2SS 335
– CVTSS2SD 350
– CVTSS2SI 335
– CVTTPD2DQ 351
– CVTTPD2PI 351
– CVTTPS2DQ 351
– CVTTPS2PI 351
– CVTTSD2SI 351
– CVTTSS2SI 351
– CWD 99
– CWDE 99
– DAA 62
– DAS 62
– DEC 55
– DIV 51

– DIVPD 345
– DIVPS 320
– DIVSD 345
– DIVSS 320
– EMMS 298
– ENTER 158
– F2XM1 228
– FABS 213
– FADD 205
– FADDP 210
– FBLD 245
– FBSTP 245
– FCHS 213
– FCLEX 256
– FCMOVcc 248
– FCOM 230
– FCOMI 234
– FCOMIP 234
– FCOMP 230
– FCOMPP 230
– FCOS 219
– FDECSTP 257
– FDISI 267
– FDIV 205
– FDIVP 210
– FDIVR 211
– FDIVRP 211
– FEMMS 378
– FENI 267
– FFREE 258
– FIADD 208
– FICOM 232
– FICOMP 232
– FIDIV 208
– FIDIVR 211
– FILD 243
– FIMUL 208
– FINCSTP 257
– FINIT 253
– FIST 243
– FISTP 243
– FISUB 208
– FISUBR 211
– FLD 238
– FLD1 240
– FLDCW 254
– FLDENV 264, 873
– FLDL2E 240
– FLDL2T 240
– FLDLG2 240
– FLDLN2 240
– FLDPI 240
– FLDZ 240
– FMUL 205
– FMULP 210
– FNCLEX 256
– FNDISI 267
– FNENI 267
– FNINIT 253
– FNOP 266
– FNSAVE 259, 870
– FNSTCW 254
– FNSTENV 264, 873
– FNSTSW 255
– FPATAN 222
– FPREM 214
– FPREM1 214
– FPTAN 221
– FRNDINT 250
– FRSTOR 259, 870
– FSAVE 259, 870
– FSCALE 251
– FSETPM 267
– FSIN 219
– FSINCOS 220
– FSQRT 212
– FST 241
– FSTCW 254
– FSTENV 264, 873
– FSTP 241
– FSTSW 255
– FSUB 205
– FSUBP 210
– FSUBR 211
– FSUBRP 211
– FTST 236
– FUCOM 230
– FUCOMI 234
– FUCOMIP 234
– FUCOMP 230
– FUCOMPP 230
– FWAIT 267
– FXAM 238
– FXCH 247
– FXRSTOR 261, 299, 868
– FXSAVE 261, 299, 868
– FXTRACT 251
– FYL2X 224

– FYL2XP1 224
– HLT 179
– IDIV 51
– IMUL 48
– IN 98
– INC 55
– INS 133
– INSB 133
– INSD 133
– INSW 133
– INT 122
– INT3 123
– INTO 123
– INVD 179
– INVLPG 180
– IRET 122
– IRETD 122
– Jcc 103
– JCXZ 103
– JECXZ 103
– JMP 101
– LAHF 126
– LAR 176
– LDMXCSR 339
– LDS 142
– LEA 142, 821
– LEAVE 158
– LES 142
– LFENCE 358
– LFS 142
– LGDT 166
– LGS 142
– LIDT 166
– LLDT 168
– LMSW 181
– LOCK siehe Präfixe
– LODS 131
– LODSB 131
– LODSD 131
– LODSW 131
– LOOP 106
– LOOPcc 106
– LSL 176
– LSS 142
– LTR 169
– MASKMOVDQU 356
– MASKMOVQ 340
– MAXPD 345
– MAXPS 323
– MAXSD 345
– MAXSS 323
– MFENCE 358
– MINPD 345
– MINPS 323
– MINSD 345
– MINSS 323
– MOV 86, 170
– MOVAPD 346
– MOVAPS 330
– MOVD 296
– MOVDQ2Q 353
– MOVDQA 353
– MOVDQU 353
– MOVHLPS 332
– MOVHPD 346
– MOVHPS 331
– MOVLHPS 332
– MOVLPD 346
– MOVLPS 331
– MOVMSKPD 347
– MOVMSKPS 332
– MOVNTDQ 356
– MOVNTI 356
– MOVNTPD 346, 356
– MOVNTPS 340
– MOVNTQ 340
– MOVQ 296
– MOVQ2DQ 353
– MOVS 132
– MOVSB 132
– MOVSD 132, 346
– MOVSS 330
– MOVSW 132
– MOVSX 88
– MOVUPD 346
– MOVUPS 330
– MOVZX 88
– MUL 48
– MULPD 345
– MULPS 320
– MULSD 345
– MULSS 320
– NEG 57
– NOP 165
– NOT 64
– OR 64
– ORPD 346
– ORPS 323

– OUT 98
– OUTS 133
– OUTSB 133
– OUTSD 133
– OUTSW 133
– PACKSSDW 287
– PACKSSWB 287
– PACKUSWB 287
– PADDB 282
– PADDD 282
– PADDQ 354
– PADDSB 282
– PADDSW 282
– PADDUSB 282
– PADDUSW 282
– PADDW 282
– PAND 295
– PANDN 295
– PAUSE 358
– PAVGB 311
– PAVGUSB 377
– PAVGW 311
– PCMPEQB 286
– PCMPEQD 286
– PCMPEQW 286
– PCMPGTB 286
– PCMPGTD 286
– PCMPGTW 286
– PEXTRW 312
– PF2ID 376
– PF2IW 380
– PFACC 375
– PFADD 370
– PFCMPEQ 375
– PFCMPGE 375
– PFCMPGT 375
– PFMAX 375
– PFMIN 375
– PFMUL 370
– PFNACC 380
– PFPNACC 380
– PFRCP 370
– PFRCPIT1 370
– PFRCPIT2 370
– PFSQIT1 372
– PFSQRT 372
– PFSUB 370
– PFSUBR 370
– PI2FD 376
– PI2FW 380
– PINSRW 312
– PMADDWD 285
– PMAXSW 312
– PMAXUB 312
– PMINSW 312
– PMINUB 312
– PMOVMSKB 313
– PMULHRW 378
– PMULHUW 314
– PMULHW 284
– PMULLW 284
– PMULUDQ 354
– POP 94
– POPA 97
– POPAD 97
– POPF 125
– POPFD 125
– POR 295
– PREFETCH 379
– PREFETCHW 379
– PREFETCHx 341
– PSADBW 315
– PSHUFD 355
– PSHUFHW 355
– PSHUFLW 355
– PSHUFW 316
– PSLLD 293
– PSLLDQ 356
– PSLLQ 294
– PSLLW 293
– PSRAD 293
– PSRAW 293
– PSRLD 293
– PSRLQ 294
– PSRLW 293
– PSUBB 282
– PSUBD 282
– PSUBQ 354
– PSUBSB 282
– PSUBSW 282
– PSUBUSB 282
– PSUBUSW 282
– PSUBW 282
– PSWAPD 381
– PUNPCKHBW 289
– PUNPCKHDQ 289
– PUNPCKHWD 289
– PUNPCKLBW 290

– PUNPCKLDQ 290
– PUNPCKLWD 290
– PUSH 94
– PUSHA 97
– PUSHAD 97
– PUSHF 125
– PUSHFD 125
– PXOR 295
– RCL 78
– RCPPS 321
– RCPSS 321
– RCR 78
– RDMSR 171
– RDPMC 172
– RDTSC 172
– REP siehe Präfixe
– REPcc siehe Präfixe
– RET 107
– ROL 77
– ROR 77
– RSM 180
– RSQRTPS 322
– RSQRTSS 322
– SAHF 126
– SAL 76
– SAR 76
– SBB 53
– SCAS 133
– SCASB 133
– SCASD 133
– SCASW 133
– SETcc 119
– SFENCE 342
– SGDT 166
– SHL 75
– SHLD 76
– SHR 75
– SHRD 76
– SHUFPD 347
– SHUFPS 333
– SIDT 166
– SLDT 168
– SMSW 181
– SQRTPD 345
– SQRTPS 320
– SQRTSD 345
– SQRTSS 320
– STC 127
– STD 127
– STI 127
– STMXCSR 339
– STOS 131
– STOSB 131
– STOSD 131
– STOSW 131
– STR 169
– SUB 45
– SUBPD 345
– SUBPS 320
– SUBSD 345
– SUBSS 320
– SYSCALL 114
– SYSENTER 111
– SYSEXIT 111
– SYSRET 114
– TEST 72
– UCOMISD 346
– UCOMISS 328
– UD2 166
– UNPCKHPD 347
– UNPCKHPS 334
– UNPCKLPD 347
– UNPCKLPS 334
– VERR 177
– VERW 177
– WAIT 165
– WBINVD 179
– WRMSR 171
– XADD 92
– XCHG 89
– XLAT 91
– XLATB 91
– XOR 64
– XORPD 346
– XORPS 323
Instruktionspräfix siehe Präfixe
Intel 144
Intel Architecture 782
Intel-Codierung 768
Interpretation 35
Interprivileg-CALLs 479
Interrupt Descriptor Table 428, 492
interrupt gate 424, 493
interrupt request 529
interrupt vector table 490
Interrupt-Behandlung 489
Interrupts 120, 424, 486
– Real Mode 552

INTR 529
IOPL 483
IOPL-sensibel 483

J
J-Bit 789, 794

K
Kernelmodus 469

L
LDT siehe local descriptor table
Limit Checking 471
linearer Adressraum 394
Little-Endian-Format 781
local descriptor table 417, 428
local unwind 732
logische Adresse 403, 436
LongInt 808

M
Mantisse 789
Maskierte Exceptions 365
M-Bit 791
Microcode 771
millenium edition 458
MMX 274, 793
MMX und die Floating-Point Unit 303
MMX-Befehle 280
MMX-Befehlssatz 352
MMX-Datenformate 274
MMX-Emulation 306
MMX-Erweiterungen 377, 381
– unter SSE 311
MMX-Register 275
Mnemonics 43, 768
ModR/M-Byte 770
Modul 772
Multitasking 462
MXCSR 361

N
Name Mangling 721
NaN 792
– quiet NaN 792
– signalling NaN 792
Nibble 36, 787
NMI 504, 529
Nomenklatur 779
Non-numeric Exceptions 268, 306, 360
Normale 790
Not a Number 792
Null 791
Null-Exponent 793
Nullselektor 430, 474
Numeric Exceptions 269, 306, 360
numerische Makros 710

O
Objekt-File 715
OctelInt 806
OctelWord 806
Oktalzahlen 785
Opcode 768
Operand Size 766
Operand Size Override 137
Operandengröße 765, 818
Operatoren 690, 692
– ! 704, 706, 707
– != 705
– " 693
– % 707
– & 705, 706
– && 705
– ' 693
– ( ) 690
– * 691
– + 691
– . 691
– .TYPE 703
– - 691
– / 691
– ;; 708
– < 705
– <= 705
– < > 707
– == 705
– > 705
– >= 705
– ? 690
– || 705
– ADDR 693
– AND 694
– CARRY? 706
– CODEPTR 694
– DATAPTR 694

– DUP 694
– EQ 694
– GE 695
– GT 695
– HIGH 695
– HIGHWORD 695
– LARGE 696
– LE 696
– LENGTH 696
– LENGTHOF 697
– LOW 697
– LOWWORD 697
– LROFFSET 698
– LT 698
– MASK 698
– MOD 698
– NE 698
– NOT 698
– OFFSET 699
– OPATTR 699
– OR 699
– OVERFLOW? 706
– PARITY? 706
– PROC 699
– PTR 700
– SEG 700
– SHL 701
– SHR 701
– SIGN? 706
– SIZE 701
– SIZEOF 701
– SMALL 702
– SYMTYPE 702
– THIS 702
– TYPE 704
– WIDTH 704
– XOR 704
– ZERO? 706

P

PackedByte 813
PackedDouble 814
PackedDouble-Word 813
PackedIntegers 813
PackedLongInt 813
PackedQuadInt 814
PackedQuad-Word 814
PackedReals 814
PackedShortInt 813
PackedSingle 814
PackedSmallInt 813
PackedWord 813
PAE-Modus 443, 449
Page Attribute Table 457
Page Directory 442
page directory entry
– PDBR 444
page directory pointer table 449
Page Table 441
page table entry 446
Pages 441
Paging 441, 481
Paging-Mechanismus 445, 468
Physikalische Adresse 436, 441
Port 827
Präfixe 134, 769
– address size override 137, 765
– branch hints 140
– branch not taken 358
– branch taken 358
– LOCK 139
– operand size override 137, 765
– REP 137
– REPcc 137
– segment override 135
Precision Control 192
prefix siehe Präfixe
Privilege Level Checking 474
Privilegierte Instruktionen 481
Privilegstufe 407, 469
Professional-Windows 458, 469
Programm 772
Prolog 750
Protected Mode 399, 438, 492
Prozeduren 777
Prozess 773
Prüfung der Privilegien 481
Prüfung des Page-Typs 482
PSE-36-Modus 443, 449
PSE-Modus 447

Q

qNaN 792
QuadInt 807
QuadWord 807
Qualifizierte Adresse 436
Quasi-Parallelität 463
quiet NaN 792

R
Real Mode 400, 440, 490, 766
Real Mode Exceptions 553
Rechenregister 199
Register
– CPU-Basis-Register 31
– Debugregister 856
– FPU-Register 188
– GDTR 432
– IDTR 432
– Kontrollregister 856
– LDTR 432
– modellspezifische Register 856
– PDBR 444
– PDPTR 450
– Segmentregister 433
– Systemregister 431
– TR 433, 464, 473
– XMM-~ 309, 343
requestor privileg level 471
Rounding Control 193
Routinen 777
RPL 471, 776

S
Saturation 278
ScalarDouble 815
ScalarSingle 815
Schiefe 790
Schiefer Exponent 789
Schlüsselworte
– Assembler 749
– Inline 749
Schutzkonzepte 405, 465
Schutzmechanismen 467
segment limit 471
Segment Override 135
segment override prefix 37
Segmentattribute 396
Segmentdeskriptor 396
Segmente 394
Segmented Model 394
Segmentgrenzen 403
Segment-Register 37, 433
Segment-Selektor 429, 473
Selektor 429, 776
Selektor-Index 776
self modifying code 473
ShortInt 809
ShortPacked-Byte 812
ShortPacked-DoubleWord 812
ShortPackedIntegers 812
ShortPacked-LongInt 812
ShortPackedReals 812
ShortPacked-ShortInt 812
ShortPacked-SmallInt 812
ShortPacked-Word 812
SIB-Byte 770
sign extension 802
signalling NaN 792
SIMD-Realzahl-Exceptions 542
Single Instruction _ Multiple Data 272
SingleReal 800
Skalare XMM-Daten 318
SmallInt 808
sNaN 792
Software-Interrupts 488
Speicheradressierung 816
Speichermodell 715
Speicherorganisation 394
Speichersegmentierung 468, 481
Speicherverwaltung 394
SSE 308
SSE2-Befehle 344
SSE-Befehle 310
SSE-Datenformat 309
Stack 385
Stack Frames 389
Stack Operand Size 767
Stack Pointer register 33
Stack Switching 393
Stack-Rahmen 158, 389
Stack-Segment 386, 394, 415, 477, 767
Stalagmiten 389
Stalagtiten 389
Stand-Alone-Assembler 557, 745
Standard-Adressgröße 765
Standard-Datengröße 765
Status Register 194
Status Word 191
Statusflags 40
STDCALL 718, 720
Strings 127
strukturierte Ausnahme-behandlung 729, 760
Supervisor mode 481
Symbol 558

Symbole 709
– $ 710
– ??date 709
– ??filename 710
– ??time 710
– ??Version 713
– @32Bit 711
– @CatStr 713
– @code 709
– @CodeSize 711
– @CPU 711
– @curseg 709
– @data 709
– @DataSize 711
– @Date 709
– @Environ 709
– @fardata 710
– @fardata? 710
– @FileCur 710
– @FileName 710
– @InStr 713
– @Interface 711
– @Line 712
– @Model 712
– @Object 714
– @SizeStr 713
– @stack 710
– @Startup 712
– @SubStr 714
– @Table 714
– @TableAddr 714
– @Time 710
– @Version 712
– @WordSize 713
System Management Mode 407, 487
System-Register 431
Systemsegment 395, 417
Systemsegment-Deskriptoren 417

T
Tag Register 190
Task 418, 464, 775
task execution space 464
task gate 426, 494
Task Selector 464
task siehe Task State Segments
Task State Segment 418, 464, 485
task switch 418, 465
Task Switching 463
Task-Zustandssegmente siehe Task State Segment
TempReals 779
Textmakros 709
Thread 774
Top Of Stack (TOS) 198
trap 730
trap gate 425, 493
Traphandler 730
Traps 501
Tropfsteinhöhlen 389
Type Checking 473

U
Übergabekonventionen 725
Unendlichkeit 791, 796
Unmaskierte Exceptions 269
UPN 199
User mode 481
Usermodus 469

V
Virtual 8086 Mode 406
Virtuelle Adresse 438
vorzeichenbehaftete Integer 70
vorzeichenbehaftete Zahlen 801
vorzeichenlose Integer 70
vorzeichenlose Zahlen 803

W
Win 2000 459
Win 98 458
Windows 458
Windows 2000 458
Windows 95 458
Windows 98 458
Windows ME 458
Windows NT 458
Windows XP 458
Word 808
Wrap-around 278

X
XMM-Befehlssatz 345

Z
Zweierkomplement 802

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt. Dieses eBook stellen wir lediglich als persönliche Einzelplatz-Lizenz zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich

- der Reproduktion,
- der Weitergabe,
- des Weitervertriebs,
- der Platzierung im Internet, in Intranets, in Extranets,
- der Veränderung,
- des Weiterverkaufs
- und der Veröffentlichung

bedarf der schriftlichen Genehmigung des Verlags.

Insbesondere ist die Entfernung oder Änderung des vom Verlag vergebenen Passwortschutzes ausdrücklich untersagt!

Bei Fragen zu diesem Thema wenden Sie sich bitte an: info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf unseren Websites ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und viele weitere eBooks können Sie rund um die Uhr und legal auf unserer Website

http://www.informit.de

herunterladen