Die Assembler-Referenz

Kodierung, Dekodierung und Referenz

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

Die Assembler-Referenz

Kodierung, Dekodierung und Referenz

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.

5 4 3 2 1

05 04 03 02

ISBN 3-8273-2015-1

Einbandgestaltung: Christine Rechl, München
Titelbild: Aspidium filix,Wurmfarn. © 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

Die Assembler-Referenz hat einen »Vorgänger«: *Das Assembler-Buch*. Dieses Buch ist parallel zu dem vorliegenden Werk in der fünften Auflage erschienen und seit 1993 erfolgreich auf dem Markt. 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 als die bis dahin realisierte aufweist: Die neue Auflage basiert auf dem derzeit aktuellen Intel-Pentium-4-Prozessor und seinen Möglichkeiten (mit ein wenig Abschweifen zum AMD-Athlon mit seinem 3DNow!-Instruktionssatz).

Die umfangreichen Änderungen an den modernen Prozessoren (SIMD: single instruction – multiple data) und damit einhergehend die gewaltigen Erweiterungen des Befehlssatzes (MMX, SSE, SSE2, 3D-Now!) haben es mit sich gebracht, dass das Assembler-Buch einen Umfang entwickelt hätte, der eine leichte Handhabung unmöglich gemacht hätte. Auch technisch hätten sich erhebliche Probleme ergeben: Bücher mit mehr als 1200 Seiten sind nur sehr schwer zu bändigen! Der Verlag und ich haben uns daher entschieden, Das Assembler-Buch in zwei Teile aufzuteilen: Das (neue) Assembler-Buch in der 5. Auflage (Addison-Wesley, 2002, ISBN 3-8273-1929-3) und die vorliegende Assembler-Referenz.

Das Assembler-Buch führt in bewährter Art und Weise in die Assemblerprogrammierung ein. Es erläutert die CPU- und FPU-Befehle und die neuen Multimedia-Befehle MMX, SSE, SSE2 und 3D-Now anhand von anschaulichen Beispielen! Darüber hinaus bespricht es die Assembler-Direktiven von Microsofts Makroassembler ML (früher: MASM) und Borlands Makroassembler TASM, zeigt anhand der auf der CD-ROM beiliegenden source files die Einbindung von Assemblermodulen in die Hochsprachen C++ (am Beispiel von Borlands C++-Builder) und

Delphi und geht auf die Nutzung der in diesen Sprachen integrierten Inline-Assembler ein. Die Assembler-Beispiele auf der CD-ROM zeigen nicht nur, wie man 16- und 32-Bit-Assemblerprogramme schreibt – und wie geradezu einfach das ist, nutzt man die fantastischen Möglichkeiten der modernen Assembler –, sondern auch, wie eine sinnvolle Zusammenarbeit zwischen Assemblermodulen und Hochsprachen samt moderner Exception-Verarbeitung erreicht werden kann und welche großartigen Möglichkeiten die Makroassembler mit den Makros zur Verfügung stellen.

Ein großes Kapitel in diesem Buch ist Hintergrundinformationen gewidmet, die für den Umgang mit dem Assembler, aber auch bei Programmierung in Hochsprachen wichtig und nützlich sind: Speicherverwaltung, Sicherheitskonzepte der modernen Betriebssysteme u.a. Ein Anhang geht auf Details wie die Darstellung von Zahlen im Integer- und Real-Format ein, schildert Unterschiede zu dem Pentium 4 vorangehenden Prozessoren und enthält auch eine Reihe von Tabellen, die die Dekodierung von Befehlssequenzen ermöglichen.

Die vorliegende Assembler-Referenz können Sie praktisch als Band II des Assembler-Buches auffassen. Sie enthält zu allen Befehlen die Informationen, die man benötigt, wenn man mit Assembler programmieren möchte. Es ist somit quasi die »Technische Referenz« zum Assembler-Buch mit sehr detaillierten Angaben, die bis hin zu einer Beschreibung für jeden Befehl geht, was im Inneren eines Prozessors abläuft, wenn er ihn ausführt.

Wir haben die Assembler-Referenz dennoch nicht »Das Assembler-Buch, Band II« genannt. Der Grund dafür ist, dass beide Bücher unabhängig voneinander genutzt werden können. So mag den Profi unter Ihnen nur eine Zusammenstellung der technischen Details der Befehle interessieren, da er bereits über gute Assemblerkenntnisse verfügt, die Zusammenhänge schon alle kennt und sich lediglich nicht durch didaktisch nicht sehr gut gemachte Original-Tabellen quälen möchte. Er benötigt daher vielleicht das Assembler-Buch nicht, obwohl ich auch ihm raten würde, es spaßeshalber einmal durchzublättern. Ich bin sicher, es sind auch für ihn einige wertvolle Informationen dabei. Auch für ihn ist dieses Buch als Nachschlagewerk interessant.

Der Einsteiger in die Assemblerprogrammierung dagegen mag mit der technischen Referenz noch überfordert sein – er benötigt zunächst einmal die Informationen, die er für den Einsatz des Assemblers benötigt. Die liefert ihm das Assembler-Buch. Mit zunehmender Erfahrung wird

er aber auch nicht an der Assembler-Referenz vorbeikommen, da sie die Informationen kompakt enthält, die man »täglich« benötigt.

Auch dieses Buch besitzt eine Begleit-CD-ROM. Auf ihr befindet sich ein Programm, mit dem Sie Befehlssequenzen kodieren und dekodieren können und die Wirkung der verschiedenen Teile einer Befehlssequenz auf das korrespondierende Mnemonic hat. Dieses Programm ist kein Assembler/Debugger! Es ist besser: Es zeigt Ihnen interaktiv, wie der Prozessor eine Befehlssequenz analysiert oder wie sie zusammengebaut wird.

Und nun: Viel Spaß mit der Assembler-Referenz – und mit dem Assembler-Buch!

München, Dezember 2001 Trutz Eyke Podschun

1 Befehlskodierung

Befehlssequenz

Der absolute GAU (größter anzunehmender Unmut) für den Prozessor (zumindest vom Chiphersteller Intel) in Fragen der Befehlskodierung ist in Abbildung 1.1 dargestellt: Es handelt sich um ein Ungetüm aus 16 Bytes Umfang, die alle dekodiert werden wollen. Der GAU für AMD-Prozessoren ist (zumindest theoretisch!) noch um ein Byte (den »suffix«, der in 3D-Now-Befehlen notwendig wird) größer, wie Abbildung 1.2 zeigt.

Abbildung 1.1: Allgemeine Form einer Befehlssequenz bestehend aus Präfixen, Opcode, ModR/M- und SIB-Byte, Adresse und Konstante

Abbildung 1.2: 3D-Now!-spezifische Befehlssequenz bei AMD-Prozessoren

Zugegeben: Dieser Fall wird bei AMDs Prozessoren wohl niemals eintreten, da ja bei den 3D-Now!-Befehlen lediglich ein Präfix, der address size override prefix, Sinn macht und wohl auch kein immediate folgen wird. Der GAU dürfte hier also »nur« 10 Bytes betragen. Aber: Wer weiß, was AMD und Intel sich so in Zukunft ausdenken werden? Und die Erfahrung zeigt: Existiert einmal ein feature (so z.B. ein suffix), wird es auch irgendwann – von beiden! – genutzt (z.B. die SSE-Erweiterungen von Intel auf AMD-Prozessoren unter dem Titel 3D-Now!-Profes-

sional und die extended feature flags von AMD bei Intel!). Wie dem auch sei.

Natürlich tritt dieser GAU nicht immer ein. So dürfte in der Regel nur ein Präfix erforderlich werden, wie wir bei der Besprechung der Präfixe weiter unten noch sehen werden. Auch dürfte die Kombination *address* und *immediate* eher selten sein, sodass 8-Byte-Sequenzen das weitaus häufigste Ende der Fahnenstange sind (ein Präfix, zwei Opcode-Bytes, ein ModR/M-Byte sowie eine 32-Bit-Adresse. In 16-Bit-Umgebungen gar fallen nochmals zwei Bytes der Adresse weg).

Um ehrlich zu sein: der dargestellte GAU ist ein theoretischer GAU. Denn Intel verbietet Instruktionen mit mehr als 15 Bytes Umfang und löst sogar eine Exception aus, falls dies nicht eingehalten wird. Das bedeutet also, dass es maximal drei Präfixe geben kann, wenn ein Zwei-Byte-Opcode zusammen mit ModR/M- und SIB-Byte erforderlich ist inklusive 4-Byte-Adresse und 4-Byte-Konstante!

Präfix Die Präfix-Bytes sind »Modifizierer«: Sie modifizieren einen folgenden Opcode. Generell gibt es verschiedene Arten, diesen Opcode zu modifizieren:

- Angabe der Größe der zum Einsatz kommenden Operanden (16/32 Bit),
- Angabe der Größe der zum Einsatz kommenden Adressen (16/32 Bit) sowie des Segmentes, an dem sie stehen, wenn auf den Speicher zugegriffen werden und nicht das Standard-Segment benutzt werden soll,
- Wiederholungen der folgenden Instruktion nach Art einer »Schleife«,
- Tipps über das wahrscheinlichste Ergebnis einer Bedingung im Rahmen von bedingten Sprungbefehlen,
- Unterscheidung von Befehlen in »Befehlsgruppen«.

Die Präfixe kodieren somit selbst keinerlei Befehl und entfalten ihre Funktion nur in Verbindung mit einem Befehl.

Dementsprechend werden sie in vier Gruppen eingeteilt. Da pro Gruppe maximal ein Präfix einem Befehl vorausgehen darf – sie schließen sich innerhalb einer Gruppe meistens gegenseitig aus! – können maxi-

mal vier Präfixe vor einem Befehl stehen. Bei den Gruppen handelt es sich um

Gruppe 1	Gruppe 2	Gruppe 3	Gruppe 4
Wiederholung; $F0 LOCK $F2 REPNE $F3 REP/REPE »3-Byte-Opcodes«: $F2 SSE $F3 SSE/SSE2	segment override: $2E CS: $3E DS: $26 ES: $64 FS: $65 GS: $36 SS: branch hints: $2E branch taken $3E branch not taken	$66 operand size override	$67 address size override

Opcode

Der wichtigste Teil einer Befehlssequenz ist der Opcode, in beiden Abbildungen auf Seite 11 als Op bezeichnet. Er kann maximal zwei Bytes besitzen, weshalb man auch von Ein- und Zwei-Byte-Opcodes spricht. In Zwei-Byte-Opcodes ist das erste Byte grundsätzlich das Opcode-Shift-Byte $0F. Eine Ausnahme hiervon bilden die FPU-Befehle, deren erstes Byte die »Escape«-Sequenz (ECS; die Bits 15 bis 11 des Bytes mit der Kodierung 11011_b) enthält. Somit liegt das erste Byte der Zwei-Byte-FPU-Befehle immer im Bereich $D8 bis $DF. Der »Wertebereich« von Ein-Byte-Opcodes ist also auf $00 bis $FF minus $0F minus $D8 bis $DF beschränkt.

Manchmal hört man auch von Drei-Byte-Opcodes. Dies ist jedoch nicht ganz richtig! Wahr ist, dass einige Befehle aus dem Befehlssatz mit zwei Bytes für den Opcode nicht auskommen. Dann wird jedoch nicht ein drittes Byte herangezogen, sondern eine Möglichkeit benutzt, die sich durch die Existenz des sog. ModR/M-Bytes ergibt, das in Abbildung 1.1 unmittelbar dem letzten Opcode-Byte folgen kann. Doch um das zu erklären, muss ein wenig ausgeholt werden.

Hat ein Befehl einen Operanden und handelt es sich dabei um ein Allzweckregister – und sonst nichts! –, so kann dieses Register als Teil des Opcodes kodiert werden. Dies erfolgt auch realiter, wie z.B. bei dem Befehl DEC zu sehen ist, bei dem der Registercode (0 bis 7 für die acht Allzweckregister) einfach zum »Basis-Opcode« $48 des Befehls addiert wird. So ist der Opcode für *DEC ECX* $49 (= $48 + 1).

Komplizierter wird das Ganze, wenn zwei Operanden existieren, etwa ein Ziel- und ein Quellregister. Dann ist diese einfache Möglichkeit der Befehlskodierung überfordert. In diesem Fall müssen die beiden Regis-

ter durch ein zusätzliches Byte kodiert werden. (Ansonsten blieben ja, 8 Allzweckregister jeweils als mögliche Quelle und Ziel, nur noch zwei Bit zur Kodierung des Befehls, wenn man jeweils drei Bits für die Kodierung der Register benötigt. Ein bisschen zu wenig, selbst bei RISC-Prozessoren!) Noch schwieriger wird es, wenn einer der beiden Operanden eine Speicherstelle sein kann. Dann nämlich muss zusätzlich berücksichtigt und kodiert werden, ob die Speicherstelle z.B. indirekt adressiert wird.

ModR/M-Byte

Langer Rede kurzer Sinn: In solchen Fällen gibt es ein ModR/M-Byte. In Abbildung 1.3 ist es links dargestellt. Das bedeutet: Befehle, die zwei Operanden erlauben oder erfordern und dem Programmierer die Möglichkeit geben, für beide Operanden ein Register oder eine Register-Speicherstelle-Kombination zu verwenden, haben zwangsläufig solch ein zusätzliches ModR/M-Byte.

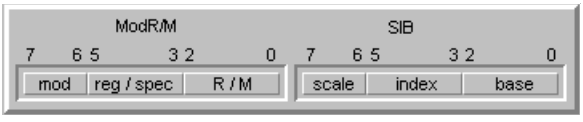


Abbildung 1.3: Speicherabbild des ModR/M- und SIB-Bytes, das Teil einer Befehlssequenz sein kann

Dieses Byte stellt drei Felder zur Verfügung: Das Modus-Feld *mod*, das den Modus der Operandenkodierung regelt, das Feld *reg*, das den Code für ein Register aufnehmen kann, und das Feld *R/M*, das seinerseits den Code für ein Register aufnehmen kann (R), oder aber die Art der Adressberechnung der Speicherstelle kodiert (M). Wir kommen auf alle drei Felder sofort zurück.

Doch zunächst zurück zu den angeblichen »Drei-Byte-Codes«. Befehle, die solche Codes verwenden, zeichnen sich durch eine Besonderheit aus: Sie haben nur maximal einen Register- oder Speicher-Operanden, wie z.B. die Form des ADD-Befehls, bei der eine Konstante zum Inhalt eines Registers oder einer Speicherstelle addiert wird: *ADD Reg, Const* oder *ADD Mem, Const*. In diesem Fall werden also nur die Felder *mod* und *R/M* des ModR/M-Bytes benötigt: mod, um zu signalisieren, dass entweder ein Register oder eine Speicherstelle betroffen ist, und R/M in Verbindung mit mod, um entweder das Register oder die Art der Adressierung des Speichers zu bestimmen. Das Feld *reg* ist frei verfügbar – und wird nun als *spec* (»special«) bezeichnet. Es kann dazu heran-

gezogen werden, zusätzlich zu den beiden Opcode-Bytes den gewünschten Befehl zu kodieren, wie am Opcode $80 gezeigt werden soll, der ein ModR/M-Byte verlangt. Ist spec = 000_b, liegt der ADD-Befehl vor, bei 001_b OR, bei 010_b ADC, dann SBB (011_b), AND (100_b), SUB (101_b), OR (110_b) und XOR (111_b). Sie sehen, es ist zwar nicht ganz falsch, von Drei-Byte-Codes zu reden, aber auch nicht richtig! Denn maximal wären es Zwei-Dreisiebentel-Byte-Codes. Und die Hauptaufgabe des ModR/M-Bytes bleibt immer noch die Kodierung der Operanden.

Verschiedene Befehle, die das spec-Feld des ModR/M-Bytes zur Kodierung heranziehen, teilen somit den gleichen Opcode, wie gesehen. Sie bilden damit eine Gruppe von Befehlen (Befehlsgruppe), die sich nur im spec-Feld des ModR/M-Bytes unterscheiden. Doch es gibt, wie oben bei den ESC-Sequenzen gesehen, auch andere Möglichkeiten, zu einer Befehlsgruppe zu gehören: Die FPU-Befehle teilen die ESC-Sequenz (Bit 15 bis 11) im ersten Opcode-Byte, die 3D-Now!-Befehle den Zwei-Byte-Opcode 0F 0F (und benötigen daher ein Suffix).

Bevor wir nun zur eingentlichen Aufgabe des ModR/M-Bytes kommen, noch eine kleine Anmerkung zum Thema »Drei-Byte-Opcodes«. Tatsächlich verwenden einige der neuen SIMD-Befehle neben einem Zwei-Byte-Opcode (und einem ModR/M-Byte, das hier jedoch tatsächlich zur Kodierung der Operanden benötigt wird) ein weiteres Byte. Sind das denn nun »echte« Drei-Byte-Opcode-Befehle? Nein! Denn was zur weiteren Unterscheidung der Befehle einer solchermaßen gebildeten Befehlsgruppe verwendet wird, ist ein Präfix, der bei anderen Befehlen anderes bewirkt. Diese Präfixe dienen somit als »Umschalter«. Und nachdem ja eine Kombination eines String-Befehls mit einem REP-Präfix auch nicht als »Zwei-Byte-Opcode-Stringbefehl« aufgefasst wird, ist eben ein SIMD-Befehl mit Präfix auch kein »Drei-Byte-Opcode-Befehl«.

Natürlich ist das Anschauungssache und man kann vermutlich Monate lang darüber philosophieren. Denn wenn man die eben genannten Kriterien ernsthaft anwenden würde, ist ja der »Umschalter« $0F, mit dem jeder »Zwei-Byte-Opcode« (außer den ESC-Sequenzen) beginnt, auch nur ein *Umschalter*, die Zwei-Byte-Opcodes demnach alle umgeschaltete Ein-Byte-Opcodes! Wer das so sehen will, kann das gerne tun. Übrigens auch der, der partout darauf besteht, dass es Drei-Byte-Opcodes gibt.

Nun zur eigentlichen Aufgabe des ModR/M-Bytes. Das Feld mod ermöglicht zusammen mit R/M die Kodierung von acht Registern oder 24 Arten der Adressierung der Speicherstelle:

mod = 00_b: Direkte Adressierung mit 32-Bit-Adresse, die dem ModR/M-Byte folgt (*EA = Adresse32*) oder indirekte Adressierung, bei der die Adresse in einem 32-Bit-Register steht: *EA = [Basisreg]*. Welches Register dies ist, zeigt R/M gemäß folgender Tabelle an:

Adress-Register	**R/M = 000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
32 Bit	EAX	ECX	EDX	EBX	SIB	adr	ESI	EDI

Zwei Ausnahmen: R/M = 100_b ist ein Code und zeigt an, dass für die indirekte Adressierung ein SIB-Byte folgen muss, in dem genauere Einzelheiten definiert werden. R/M = 101_b ist ebenfalls ein Code: Er kodiert die bereits erwähnte direkte Adressierung mit folgender 32-Bit-Adresse. Das bedeutet: EBP und ESP können für die indirekte Adressierung in diesem Modus nur in Verbindung mit einem SIB-Byte eingesetzt werden.

mod = 01_b: Indizierte indirekte Adressierung mit 8-Bit-Adresse als konstantem Teil der Adressberechnung. Im Register, das in der folgenden Tabelle aufgeführt wird, ist ein Wert verzeichnet, der als »Index« zu der Adresse addiert werden muss. Die Adresse selbst folgt dem ModR/M- bzw. SIB-Byte: *EA = Adresse8 + [Indexreg]*. Genauere Erläuterungen folgen in Kürze.

Adress-Register	**R/M = 000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
32 Bit	EAX	ECX	EDX	EBX	SIB	EBP	ESI	EDI

Auch hier eine Ausnahme: R/M = 100_b zeigt an, dass für die indirekte Adressierung ein SIB-Byte folgen muss, in dem genauere Einzelheiten definiert werden. *Adresse8* folgt dann diesem SIB-Byte unmittelbar. In diesem Modus kann zwar EBP, nicht aber ESP als Basisregister verwendet werden.

mod = 10_b: Indizierte indirekte Adressierung mit 32-Bit-Adresse, ansonsten wie mod = 01_b: *EA = Adresse32 + [Indexreg]*.

mod = 11_b. In R/M ist ein Register als zweiter Operand kodiert.

Operanden-Größe	R/M = 000	001	010	011	100	101	110	111
8 Bit	AL	CL	DL	BL	AH	CH	DH	BH
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
FPU	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
MMX	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

In *reg* ist immer ein Register kodiert – wie gesagt –, wenn tatsächlich ein weiteres Register als Operand benötigt wird. Die Kodierung dieses Registers erfolgt nach:

Operanden-Größe	reg = 000	001	010	011	100	101	110	111
8 Bit	AL	CL	DL	BL	AH	CH	DH	BH
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
FPU	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
MMX	MM0	MM1	MM2	MM3	MM4	MM5	MM&	MM7
XMM	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

In 16-Bit-Umgebungen, also wenn 16-Bit-Adressen verwendet werden, wird das ModR/M-Byte etwas anders interpretiert:

mod = 00_b: Direkte Adressierung mit 16-Bit-Adresse, die dem ModR/M-Byte folgt (*EA = Adresse16*), oder indirekte Adressierung, bei der die Adresse in einem 16-Bit-Register steht (*EA = [Basisreg]*) oder durch Addition zweier 16-Bit-Registerinhalte (Basis und Index) berechnet wird (*EA = [Basisreg] + [Indexreg]*). Als Register(-kombinationen) kommen nur wenige Möglichkeiten in Frage:

Adress-Register	R/M = 000	001	010	011	100	101	110	111
16 Bit	[BX] + [SI]	[BX] + [DI]	[BP] + [SI]	[BP] + [DI]	[SI]	[DI]	adr	[BX]

Ausnahme: R/M = 110_b (cave: nicht 101_b wie in 32-Bit-Umgebungen!) ist ein Code für die direkte Adressierung mit folgender 16-Bit-Adresse. In 16-Bit-Umgebungen kann kein SIB-Byte folgen!

mod = 01_b: Wie mod = 00_b, jedoch mit 8-Bit-Adresse. Diese folgt dem ModR/M-Byte und muss addiert werden: *EA = Adresse8 + [Basisreg]* bzw. *EA = Adresse8 + [Basisreg] + [Indexreg]*.

Adress-Register	R/M = 000	001	010	011	100	101	110	111
16 Bit	Addr + [BX] + [SI]	Addr + [BX] + [DI]	Addr + [BP] + [SI]	Addr + [BP] + [DI]	Addr + [[SI]	Addr + [DI]	Addr + [BP]	Addr + [BX]

mod = 10_b: Wie mod = 01_b, jedoch mit 16-Bit-Adresse: *EA = Adresse16 + [Basisreg]* bzw. *EA =Adresse16 + [Basisreg] + [Indexreg]*.

mod = 11_b. In R/M ist wie im Falle der 32-Bit-Adressierung ein Register als zweiter Operand kodiert.

Operanden-Größe	R/M = 000	001	010	011	100	101	110	111
8 Bit	AL	CL	DL	BL	AH	CH	DH	BH
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
FPU	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
MMX	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

Die Kodierung von reg bleibt wie gehabt:

Operanden-Größe	reg = 000	001	010	011	100	101	110	111
8 Bit	AL	CL	DL	BL	AH	CH	DH	BH
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
FPU	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
MMX	MM0	MM1	MM2	MM3	MM4	MM5	MM&	MM7
XMM	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

ModR/M-Byte: Sonderfälle

Wie bereits mehrfach geschildert, kann das reg-Feld des ModR/M-Bytes auch für die Kodierung von Sonderfällen herangezogen werden. Es heißt dann je nach Kontext spec-Feld (z.B. FPU-Instruktionen, einige CPU-»Gruppenbefehle«), sreg-Feld (Segment-Register) oder eee-Feld (»special purpose«).

spec-Feld

Die drei Bits des spec-Feldes werden zur Kodierung der Instruktion benutzt. So wird mit dem Opcode lediglich eine Befehlsgruppe spezifiziert, innerhalb derer die die Gruppe darstellenden Instruktionen mit dem spec-Feld identifiziert werden können. Da in diesem Fall das reg-Feld nicht für die Kodierung eines Registers zur Verfügung steht, können solche »spec-Instruktionen« *ausschließlich einen* Register- *oder* Speicher-Operanden haben, der über das R/M-Feld kodiert wird. Oder umgekehrt: Da bei Instruktionen, die nur einen Register-/Speicheroperanden haben, das reg-Feld frei bleiben müsste, fasst man solche Befehle zu Gruppen zusammen, bei denen die Befehlsunterscheidung dann über dieses spec-Feld erfolgt.

Von dieser Betrachtung ausgenommen sind Konstanten, die als zusätzliche Operanden auftreten können. Da deren Existenz nicht über das ModR/M-Byte signalisiert wird, sondern im Opcode selbst begründet ist (vgl. die verschiedenen Opcodes für ADD: $00 bis $03 ohne, $04 und $05 mit Konstante), können trotz Verwendung des spec-Feldes zur Instruktionskodierung auch Konstanten verwendet werden (ADD: $80, $81 und $83 verlangen ModR/M-Byte mit spec-Feld und Konstante).

Instruktionen, die sich durch diese Art der Kodierung auszeichnen, lassen sich relativ leicht erkennen: Sobald eine Instruktion die allgemeine Form

```
Befehl Registeroperand             ; z.B.: FADD ST(1)
Befehl Speicheroperand             ; z.B.: FLD  DWordVar
Befehl Registeroperand, Konstante  ; z.B.: ADD  AL,Const8
Befehl Speicheroperand, Konstante  ; z.B.: XOR  EAX, Const32
```

hat und es ggf. sogar verschiedene Befehle mit ähnlicher Struktur gibt (die eine Gruppe bilden könnten, so z.B. ADD, ADC, SUB, SBB), ist die Wahrscheinlichkeit sehr hoch, dass sie über eine *Opcode – »mod-spec-R/M«-Byte* – Folge kodiert werden.

sreg-Feld

Sobald bei einer Instruktion ein Segmentregister involviert ist, wird dieses Register (wie auch die Allzweckregister!) über das reg-Feld kodiert. In diesem Fall nennt man die Bits 5 bis 3 dann sreg-Feld des ModRM-Bytes. Die Zuordnung der Segmentregister erfolgt nach folgender Tabelle:

sreg =							
000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
ES	CS	SS	DS	FS	GS	reserviert	reserviert

Die reservierten Bitstellungen sollten nicht benutzt werden!

Es liegt auf der Hand, dass in diesem Fall ein eventuell vorhandener zweiter Operand über das R/M-Feld kodiert werden muss und daher nur ein Allzweckregister oder eine Speicherstelle in Frage kommt. Befehle, bei denen zwei Segmentregister als Operanden eingesetzt werden, gibt es daher nicht!

eee-Feld Auch die Kodierung der Control- und Debugregister erfolgt über die Bits 5 bis 3 des ModRM-Bytes, weshalb das reg-Feld in diesem Falle eee-Feld genannt wird und in den entsprechenden Instruktionen folgende Register kodiert:

eee =							
000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
CR0	reserviert	CR2	CR3	CR4	reserviert	reserviert	reserviert
DR0	DR1	DR2	DR3	reserviert	reserviert	DR6	DR7

Die reservierten Bitstellungen sollten nicht benutzt werden!

SIB-Byte Das SIB-Byte (scale, index, base) tritt also nur in Verbindung mit einem ModR/M-Byte auf und erweitert die durch jenes Byte gegebenen Möglichkeiten der indirekten Adressierung. Ein SIB-Byte kann nur in 32-Bit-Umgebungen auftreten, also wenn die verwendeten Adressen 32-bittig sind. Wie Sie im vorangehenden Abschnitt gesehen haben, fordert ein R/M-Feld mit dem Inhalt 100_b bei den Modi 00_b, 01_b und 10_b dieses zusätzliche Byte. In diesen Modi wird die Adresse indirekt berechnet, indem sie aus einem Register ausgelesen wird. Der Unterschied bei den Modi liegt lediglich darin, ob noch zusätzlich eine Konstante als direkter Anteil zur Adressberechnung herangezogen wird (mod = 01_b, 10_b) oder nicht (mod = 00_b) und, wenn ja, ob es sich um eine 8-Bit-Konstante (mod = 01_b) oder eine 32-Bit-Konstante (mod = 10_b) handelt.

An dieser Stelle ist noch nicht bekannt, welches Register die Adresse beherbergt. Daher gibt es im SIB-Byte das Feld *b* (»base«), das dieses sog. Basisregister festlegt. Zusätzlich kann ein Register definiert werden, das einen weiteren Index *i* aufnehmen kann, sodass folgende Adressberechnung möglich wird: *EA = [Base] + [Index] + Address*. Und

um das Ganze noch zu toppen, kann dieser Index sogar noch »skaliert« werden, was bedeutet, dass er mit einem Skalierungsfaktor *s* mit den Werten 2, 4 oder 8 multipliziert werden kann: *EA = [Base] + Scale * [Index] + Address*.

Warum gerade die Faktoren 1, 2, 4 und 8? Denken Sie an Tabellen. Dort können Bytes (Faktor = 1), Words (Faktor = 2), DoubleWords (Faktor = 4) und QuadWords (Faktor = 8) verzeichnet sein. Es ist also sehr hilfreich, mit einem Scale-Faktor zu arbeiten, wenn z.B. QuadWords in einer Schleife aus der Tabelle an Adresse Adr gelesen werden sollen:

```
        MOV     EBX, Adr                ; Basisadresse der Tabelle
        XOR     ECX, ECX                ; Index = 0
L1:     MOV     EAX, [EBX + 8*ECX]      ; Indirekte Adressierung
        :
        INC     ECX
        CMP     ECX, 00000008h
        JB      L1
        :
```

Versucht man nun, das alles in einen Code zu packen, der nicht mehr als 8 Bit umfassen darf, so kommt man auf eine Struktur, die der Mod-Reg-R/M-Struktur des ModR/M-Bytes zum Verwechseln ähnlich ist: das SIB-Byte mit den Feldern scale, index und base oder *s*, *i* und *b*.

$s = 00_b$: Keine Skalierung. Das Index-Register wird gemäß folgender Tabelle angegeben:

	index =							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
32 Bit	EAX	ECX	EDX	EBX	-	EBP	ESI	EDI

Ausnahmen: $i = 100_b$ ist ein Code dafür, dass kein Indexregister verwendet wird.

$s = 01_b$: Wie $s = 00_b$, jedoch wird der Inhalt des Indexregisters mit 2 multipliziert.

$s= 10_b$: Ebenfalls wie $s = 00_b$, jedoch wird der Inhalt des Indexregisters mit 4 multipliziert.

$s = 11_b$. Auch wie $s = 00_b$ mit einem Skalierungsfaktor von 8.

Das ggf. zum Einsatz kommende Basisregister wird »ganz normal« in b kodiert:

	base =							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
32 Bit	EAX	ECX	EDX	EBX	ESP	- / EBP	ESI	EDI

Ausnahme: b = 101_b. Wenn *mod* im ModR/M-Byte = 00_b ist, steht dieser Code dafür, dass keine Basis verwendet wird. Andernfalls kodiert es das EBP-Register als Basis.

Achtung: Luft holen! Das war sicherlich nicht einfach zu verstehen. Fühlen Sie sich in bester Gesellschaft, wenn Ihnen an dieser Stelle der Kopf schwirrt. Ich gebe es zu: Für mich war es ebenfalls nicht einfach, das Ganze so zu sortieren, dass man leicht durchblickt. Doch hier das Ergebnis meiner Hirnmarterei, ich hoffe es hilft Ihnen:

Gehen Sie von folgender allgemeinen Form der Berechnung einer effektiven Adresse aus (an dieser Stelle symbolisieren die geschweiften Klammern eine optionale Verwendung des betreffenden Teils und die eckigen, dass der Inhalt des entsprechenden Registers verwendet wird (indirekter Zugriff!):

```
EA = {Adresse} {+ [Basisreg]} {+ [{Faktor *} Indexreg]}
```

Das bedeutet, es gibt folgende Varianten der Adressierung:

- *EA = Adresse*; dies ist die direkte Adressierung, die mittels mod = 00_b und R/M = 101_b kodiert wird. Es folgt kein SIB-Byte, wohl aber eine 32-Bit-Adresse.
- *EA = [Basisreg]*; dies ist die »klassische« indirekte Adressierung, bei der die zu verwendende Adresse dem Register *Basisreg* entnommen wird. Dies wird mit mod = 00_b und R/M ≠ 100_b bzw. R/M ≠ 101_b kodiert, ein SIB-Byte ist nicht erforderlich, ebenso wenig wie eine folgende direkte Adresse (zu Ausnahmen kommen wir später).
- Nun wird es eine Stufe komplizierter, es wird eine Mischung aus einer direkten und einer indirekten Adressierung verwendet. Der direkte Teil spiegelt eine Adresse wider, zu der ein *Index* addiert werden muss, der einem Indexreg entnommen wird: *EA = Adresse + [Indexreg]*. Wichtig ist hierbei, dass kein Faktor zur Skalierung des Indexes erforderlich ist. Dann ist ebenfalls kein SIB-Byte nötig, da hier der »einfache« Index einem Register entnommen werden kann, und es kommt je nach Größe der Adresse mod = 01_b (8-Bit-Adresse) oder mod = 10_b (32-Bit-Adresse) zum Einsatz. Logisch, dass dann R/M nicht den Wert 100_b annehmen darf, da dies ja der Code für die Notwendigkeit zu einem SIB-Byte ist.

- Die nächste Stufe der Komplexizität der Adressen ist, dass der Index auch noch skaliert werden soll: *EA = Adresse + [Faktor * Indexreg]*. Dies ist nur mit einem SIB-Byte möglich (R/M = 100_b), da nur dort ein Skalierungsfaktor angegeben werden kann. Da bei dieser Art keine Basis verwendet wird, wohl aber eine direkt angegebene Adresse, ist mod = 00_b und b = 101_b. Der Index steht in i und der Skalierungsfaktor in s des SIB-Bytes. Diesem Byte folgt dann die direkt angegebene Adresse. ACHTUNG: Dies ist wirklich die einzige Art, diesen Fall zu realisieren. Zwar kann mit mod = 01_b oder mod = 10_b auch eine direkte Adresse angegeben und mit R/M = 100_b ein SIB-Byte angefordert werden. In diesem Fall steht jedoch b = 101_b *nicht* für »keine Basis«, was hier ja der Fall sein soll, sondern für »Basisreg = EBP«.
- Aber es geht noch komplizierter. Um sich das zu veranschaulichen, stellen Sie sich z.B. eine zweidimensionale Tabelle vor. Dann kann man die Adresse der Tabelle als direkte Adresse angeben und ein Element über einen Spalten- und einen Reihenindex selektionieren. Ausdrücken kann man dies dann so: *EA = Adresse + [Basisreg] + [Indexreg]*. Klar, dass so etwas nur mit SIB-Byte möglich ist, da hier ja zwei Register benötigt werden. In diesem Fall muss je nach Größe der Adresse mod = 01_b oder 10_b und R/M = 100_b sein, da ja ein direkter Adressanteil zum Einsatz kommt und der Sonderfall mod = 00_b, R/M = 100_b nicht greift (weil b nicht 101_b sein kann). Das Feld b des SIB-Bytes enthält hier das *Basisreg*, das Feld i das *Indexreg*. Das Feld s ist 0.
- Ende der Fahnenstange: *EA = Adresse + [Basisreg] + [Faktor * Indexreg]*. Das ist aber mit dem vorangehenden Fall identisch, wenn man das Feld s eben nicht mit dem Wert 0 kodiert.

Es gibt jedoch auch noch ein paar Möglichkeiten, die bislang noch nicht berücksichtigt wurden:

- *EA = Adresse + [Basisreg]*. Dies ist aber der gleiche Fall wie *EA = Adresse + [Indexreg]*, da nur ein Register für den indirekten Anteil zum Einsatz kommt und es schließlich egal ist, wie man dieses Register formal bezeichnet!
- *EA = [IndexReg]*. Aus den gleichen Gründen ist das das Gleiche wie *EA = [Basisreg]*.
- *EA = [Faktor * Indexreg]*. Dies ist ein interessanter Fall, da hier ein SIB-Byte erforderlich wird (wegen des Skalierungsfaktors), aber kein direkter Adressanteil vorliegt. Somit kommt weder mod = 01_b noch 10_b in Frage. Bliebe noch mod = 00_b und R/M = 100_b. Dann aber käme nur b = 101_b in Betracht, da ja kein Basisreg verwendet wird.

Diese Konstellation ist jedoch für den Fall *EA = Adresse + [Faktor * Indexreg]* reserviert. Ausweg aus dem Dilemma: Man setzt die Adresse einfach auf den Wert 0. Und so geschieht es auch: In dieser Konstellation wird eine Befehlssequenz erzeugt, in der die folgende direkte Adresse mit $0000_0000 angegeben wird.

- *EA =[Basisreg] + [Indexreg]* bzw. *EA = [Basisreg] + [Faktor * Indexreg].* Hierbei handelt es sich um eine vollständige indirekte Adressierung ohne direkten Anteil. Ermöglicht wird dies in der Konstellation mod = 00_b, R/M = 100_b (SIB-Byte!) und b ≠ 101_b. Je nachdem, ob s = 0 ist oder nicht, wird dann ein Faktor verwendet – oder eben nicht! Nachteil: Sie können EBP nicht als Basisreg verwenden. Lösung: Verwenden Sie es als Indexreg, sofern nicht skaliert werden muss. Richtig Pech haben Sie, wenn Indexreg und Basisreg = EBP sein soll. Das geht dann nicht. Aber warum sollten Sie für das Indexreg und das Basisreg das gleiche Register wollen? Ich wüsste schon warum – und werde Ihnen den Grund gleich als Tipp geben.
- *EA = [EBP]* und *EA = [ESP].* Dies sind interessante Konstellationen! Denn hier soll jeweils ein Register verwendet werden, das in dieser Form (kein direkter Adressanteil, daher mod = 00_b) nicht verwendet werden kann: Bei mod = 00_b sind die R/M-Werte, die EBP und ESP kodieren würden, für die Sonderfälle direkte Adressierung und SIB-Byte reserviert. Ausweg: Es wird im Falle von EBP mod = 01_b verwendet, wo EBP den R/M-Code 101_b hat. Konsequenz: Es muss ein direkter Anteil angegeben werden. Aber den kann man ja auf »0« setzen. So wird *MOV EAX, [EBP]* assembliert zu $8B $45 $00. Hier ist ModR/M = $45 und somit mod = 01_b, reg = 000_b (= EAX), R/M = 101_b und Adresse8 = $00. Etwas komplizierter wird es bei *EA = [ESP]*. Denn R/M = 100_b ist in allen Modi für »SIB-Byte erforderlich« reserviert. Also kommen wir um ein SIB-Byte nicht herum, da dies die einzige Möglichkeit ist, ESP z.B. als Basisregister einzusetzen. Und so erfolgt es auch: *MOV EAX, [ESP]* wird assembliert zu $8B $04 $24. Das ModR/M-Byte $04 wird zerlegt zu mod = 00_b, reg = 000_b und R/M = 100_b, was das SIB-Byte $24 anfordert. Dieses wiederum kann zerlegt werden in s = 00_b, i = 100_b und b = 100_b. Und das bedeutet laut obiger Tabellen: kein Skalierungsfaktor, kein Index aber Basis = ESP. Voilà!
- Analoges gilt natürlich, wenn ein direkter Anteil dazukommt: *EA = Adresse + [EBP]* bzw. *EA = Adresse + [ESP]*. Im Vergleich zu dem vorangehenden Fall ändert sich in beiden Fällen nur mod: Ist die Adresse 32-bittig, wird mod = 10_b verwendet, bei 8-Bit-Adressen mod = 01_b. Natürlich folgt in beiden Fällen eine Konstante, die dem Mod-Wert Rechnung trägt.

Es gibt auch ein paar »verbotene« Formen. So z.B. mod = 00_b, R/M = 100_b, i = 100_b und b = 101_b, s beliebig. Wenn Sie diese Werte nehmen und mit ihnen eine Befehlssequenz zusammenbauen, die Sie per DB-Direktive ins Codesegment einbauen, werden Sie Ihr blaues Wunder erleben! Warum? Wenn R/M = 100_b ist, heißt das: »SIB-Byte erforderlich«. Dann aber wird mit i = 100_b gesagt: »kein Index« und mit b = 101_b bei mod = 00_b: »keine Basis«. Warum dann ein SIB-Byte? Wegen des Skalierungsfaktors? Unsinn: »Faktor * kein Index« ist immer noch »kein Index«! Somit ist dieses ModR/M-Byte unzulässig. Das Problem: Der Assembler wird bei der Assemblierung keine Fehlermeldung ausgeben, da Sie ja DB-Direktiven eingestreut haben, die der Assembler nicht als Instruktion betrachtet. Also wird anstandslos assembliert. Doch die CPU mag später nicht mitspielen, wenn sie diese Codesequenz dann abarbeiten soll. Resultat: Sie steigt mit einer Exception aus.

Auch sollten Sie, wenn Sie keinen Index im SIB-Byte kodieren, das Feld s mit 0 kodieren. Denn aus gleichen Gründen akzeptiert die CPU ein SIB-Byte nicht, das widersprüchliche Angaben macht: Skalierungsfaktor ja, aber zu skalierender Index nein.

Ich hoffe, nun ist etwas klarer geworden, wie ModR/M- und SIB-Byte mit einer Adresse zusammenspielen, um die verschiedenen Möglichkeiten zu realisieren, die bei der indirekten Adressierung möglich sind.

Sie können den Skalierungsfaktor, den Sie im SIB-Byte angeben, noch ein wenig erweitern. Wie gesagt, er kann Werte von 1, 2, 4 und 8 annehmen. Wenn Sie nun für Index- und Basisreg das gleiche Register verwenden, heißt das, dass Sie so die »Faktoren« 3, 5 und 9 realisieren können:

EA = [1 * Indexreg]
EA = [2 * Indexreg]
EA = [2 * Indexreg] + [Basisreg] = [3 * Indexreg]
EA = [4 * Indexreg]
EA = [4 * Indexreg] + [Basisreg] = [5 * Indexreg]
EA = [8 * Indexreg]
EA = [8 * Indexreg] + [Basisreg] = [9 * Indexreg]

Leider geht das nicht mit 6 und 7!

Der Mensch ist ein optisches Wesen. Daher habe ich in Abbildung 1.4 ein Flussdiagramm realisiert, das den Weg der Kodierung eines ModR/M- und, falls erforderlich, eines SIB-Bytes aufzeigt.

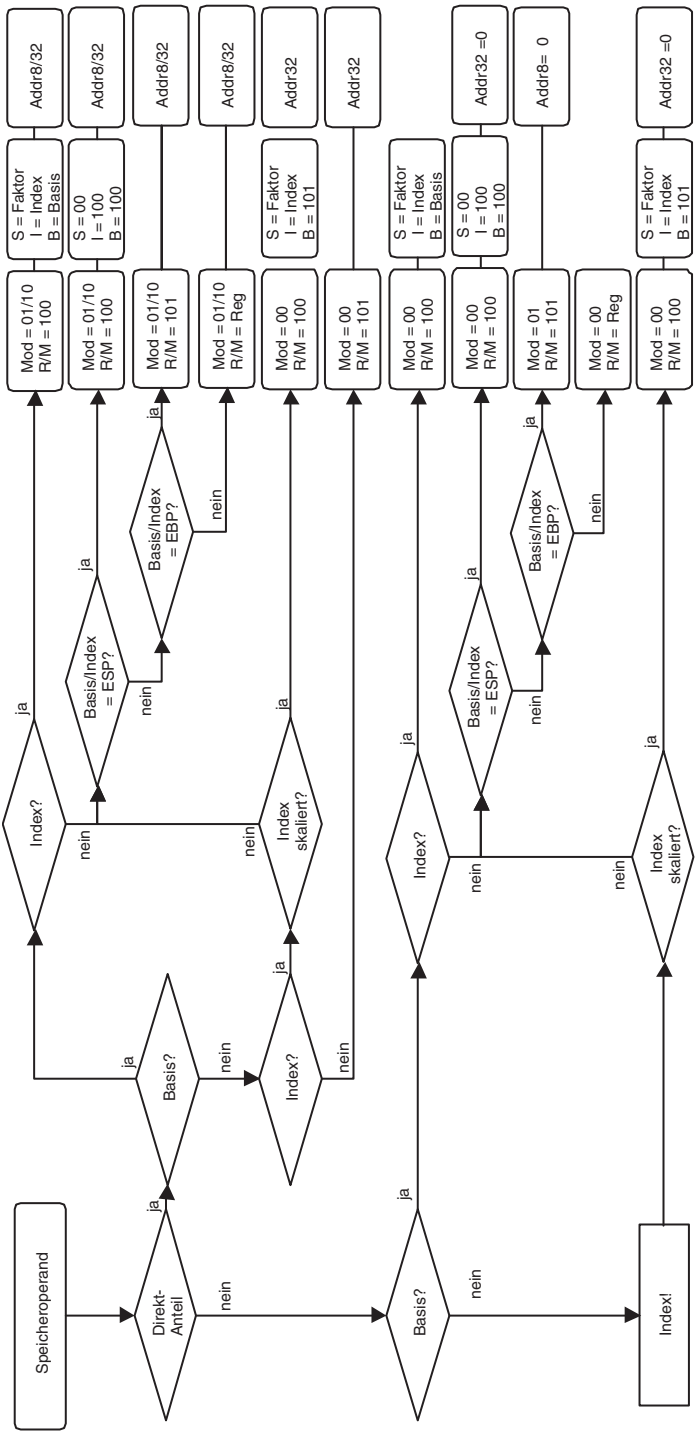


Abbildung 1.4: Flussdiagramm zur Kodierung des ModR/M- und ggf. SIB-Bytes einer Befehlssequenz, wenn einer der Operanden ein Speicheroperand ist

Voraussetzung ist, dass einer der Operanden des Befehls, zu dem diese Bytes gehören, ein Speicheroperand ist. Handelt es sich in beiden Fällen um einen Registeroperanden, ist der Fall einfach: mod ist dann 11b und reg und R/M kodieren die beiden Register.

Beachten Sie bitte, dass dieses Flussdiagramm die »verbotenen« Möglichkeiten nicht berücksichtigt. Aber es dürfte klar sein, wie und an welcher Stelle diese Fälle auf das Diagramm Auswirkungen hätten.

Adresse

Das Thema *Adresse* ist schnell abgehandelt. Wir haben im vorangehenden Abschnitt ja gesehen, wann und wie sie zum Einsatz kommt. Dennoch gibt es noch etwas anzumerken.

Falls einer der (oder *der*) Operand(en) eine Speicherstelle ist, benötigt der Assembler natürlich deren Adresse. Hierzu ist grundsätzlich eine logische Adresse erforderlich. Eine Logische Adresse besteht aus einem Selektor und einer effektiven Adresse, mit anderen Worten: dem Zeiger auf ein Segment (der in einem Segmentregister steht) und einem Offset in dieses Segment.

Den Segmentanteil können wir als abgehakt ansehen: Grundsätzlich beziehen sich alle Befehle, die eine Speicherstelle (als Datum!) adressieren, entweder auf das Stack- oder das Datensegment. Falls also im Rahmen der indirekten Adressierung das Register (E)SP oder (E)BP als Basisregister ins Spiel kommt, wird der in SS stehende Selektor bemüht, andernfalls der in DS stehende.

Daher benötigt man nur noch den Offset. Und dieser wird eben entweder indirekt in einem Register übergeben oder – was uns in diesem Abschnitt interessiert – als Konstante unmittelbar an das ModR/M- bzw. SIB-Byte anschließend. Diese Konstante kann zwei oder vier Bytes Umfang haben: In 16-Bit-Umgebungen wird sie durch ein Word angegeben, in 32-Bit-Umgebungen durch ein DoubleWord. Das war's eigentlich.

Was heißt das für die Länge unserer Befehlssequenz? Sobald ein SIB-Byte vorliegt, müssen auch ein ModR/M-Byte und eine Adresse vorhanden sein. Und da SIB nur in 32-Bit-Umgebungen möglich ist, heißt das: 1 (ModR/M) + 1 (SIB) + 4 (Adresse) Bytes zusätzlich zum Ein- oder Zwei-Byte-Opcode. Durchschnittliche Befehlssequenzen mit indirekter Adressierung und Angabe einer Speicherstelle haben in 32-Bit-Umgebungen also eine Größe von sieben bis acht Bytes. Wird dagegen direkt

adressiert, so benötigt man nur das ModR/M-Byte und die Adresse, also ein Byte weniger.

In 16-Bit-Umgebungen gibt es kein SIB-Byte und die Adressgröße ist auf zwei Bytes beschränkt. Das bedeutet, indirekte Adressierung mit Angabe einer Speicherstelle vorausgesetzt: 1 (ModR7M) + 2 (Adresse) = 3 Bytes zusätzlich zum Opcode, also durchschnittlich vier bis 5 Bytes für eine Befehlssequenz. Dies ist auch die Befehlslänge, wenn Sie direkt adressieren.

Konstante Bleibt noch das Feld, das in Abbildung 1.1 als *immediate* bezeichnet wurde. Dieses Feld wird erforderlich, wann immer dem Befehl eine Konstante als Operand übergeben wird, wie z.B. in *SHL EAX, 1*. Diese Konstante ist je nach Operandengröße 1 (Const8), zwei (Const16) oder vier (Const32) Bytes lang und folgt entweder der Adresse, die aufgrund der Einbindung eines Speicheroperanden erforderlich ist, oder direkt dem ModR/M- oder gar dem (letzten) Opcode-Byte. Im günstigsten Fall hat eine Befehlssequenz mit Konstante somit zwei Bytes Umfang (je ein Byte für Opcode und Konstante), wie z.B. in *ADD AL, 05h*: $04 $05.

Präfixe Wie Sie bereits wissen, dienen die Präfixe als »Umschalter« oder »Befehlsmodifizierer«. So schalten operand und address size override prefix wie eine Umschalt-Taste auf der Tastatur für den folgenden Befehl die Standardgröße von Operanden bzw. Adressen ab, während die segment override prefixes den Befehl so modifizieren, dass eine Adressberechnung nicht relativ zum Standardsegment erfolgt, sondern zum im prefix angegebenen. Die REPcc-Befehle modifizieren die String-Befehle so, dass diese bis zu einem (oder zwei) Abbruchkriterium wiederholt werden. Und die branch hints beeinflussen die prediction logic, die für die Vorhersage des wahrscheinlichsten Sprungziels bei einem bedingten Sprung verantwortlich ist, so, dass das angenommen wird, was der Programmierer für richtig hält.

Es ist klar, dass nicht jeder Präfix bei jedem Opcode Sinn macht: Ein segment override prefix macht bei Speicherzugriffen Sinn, nicht aber bei bedingten Sprüngen, die ja immer relativ zur aktuellen Adresse in aktuellen Codesegment sind. Und ein REP-Präfix in Verbindung mit MOV ist tatsächlich Schwachsinn. Daher erfolgt die Verwendung vieler Präfixe automatisch durch den Assembler oder dieser prüft die Rechtmäßigkeit der Verwendung und meckert ggf. Doch vor den Tücken der

»Handkodierung« durch den Programmierer ist er nicht gefeit. Was bedeutet, dass er nicht eingreifen kann, wenn Sie mittels DB-Direktiven Nonsense-Sequenzen erzeugen.

In der Regel werden niemals alle vier Präfixe zusammen eingesetzt werden, sondern einer: REP/REPNE bei String-Befehlen, oder hin und wieder mal ein operand size override prefix (z.B. bei SIMD-Befehlen oder wenn einmal ein Word-Register involviert sein sollte. Grund: Da wohl heutzutage die weitaus größte Zahl der Programme für 32-Bit-Betriebssysteme geschrieben werden, dürfte die Standard-Operandengröße 32 Bit und die Standard-Adressgröße ebenfalls 32 Bit betragen. Somit entfällt in der Regel die Notwendigkeit zur Verwendung des address size override prefix und meistens auch des operand size override prefix. Da darüber hinaus das Programmiermodell »flach« ist, besitzen DS, ES und SS sowieso den gleichen Inhalt und der Einsatz von segment override prefixes dürfte eher selten sein.

Suffix

Die Suffixe sind (noch) eine Besonderheit der AMD-Prozessoren. Wenn Sie so wollen, sind Befehle, die einen Suffix verwenden, tatsächlich Drei-Byte-Befehle, da der Opcode aus den beiden Bytes 0F 0F besteht und der Suffix die Unterscheidung der Befehle in der Befehlsgruppe vornimmt. Zurzeit ist das nur bei den 3D-Now!-Befehlen der Fall. In den folgenden Tabellen wird der Suffix zusätzlich zum Opcode angegeben, wo immer erforderlich.

Alle Bytes einer Befehlssequenz außer dem Opcode lassen sich durch die Art der verwendeten Operanden und dem eingesetzten Opcode herleiten. Sie sind auch unabhängig davon, ob ein CPU-, FPU- oder SIMD-Befehl zum Einsatz kommt. Daher ist wichtigster und für die Unterschiede verantwortlicher Teil einer Befehlssequenz der Opcode – und nichts anderes. Er wird daher in den folgenden Kapiteln als einziger Teil der Befehlssequenz für jeden Befehl angegeben. (Wie gesagt: mit der Ausnahme der 3D-Now!-Befehle!)

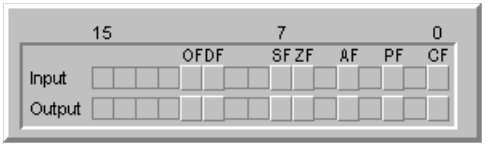
2 CPU-Instruktionen

Befehl(e) **eingeführt mit Prozessor**

Alle CPU-Instruktionen werden nach einem einheitlichen Schema dargestellt. Dieses Schema sieht wie folgt aus:

Kurze Beschreibung der Funktion des Befehls. *Funktion*

`Formale Darstellung der durchgeführten Operation in einer Pascal-ähnlichen Notation` *Operation*



Statusflags

Erläuterung der Symbole und ggf. Kommentar zu den Flags. Erhabene, nicht ausgefüllte Positionen signalisieren hierbei, dass die beim Eintritt in die Instruktionsbehandlung vorgefundenen Flags nicht verändert werden. Die Flagstellungen spiegeln somit den Zustand wider, den sie seit der letzten Veränderung erfahren haben.

Opcodes

Die definierten Opcodes werden in Form einer Tabelle dargestellt. Hierbei finden sich in der ersten Spalte die Opcodes für den Fall, dass 8-Bit-Operanden verwendet werden, in Spalte 2 die Bytesequenzen für die »Standardumgebung«. Bei 32-Bit-Betriebssystemen sind das 32-Bit-Operanden. Sollen in diesem Falle 16-Bit-Operanden verwendet werden, muss dem Opcode ein (nicht dargestelltes) operand size override prefix ($66) vorangestellt werden. In 16-Bit-Umgebungen kommen standardmäßig 16-Bit-Operanden zum Einsatz, die Verwendung von 32-Bit-Operanden erfordert dann wiederum den operand size override prefix. Alle in Spalte 1 und 2 angegebenen zweistelligen Zahlen sind Hexadezimalzahlen.

In den Tabellen wird auch ein ModR/M-Byte ausgewiesen, falls eines erforderlich ist. Seine Notwendigkeit wird ggf. durch »/r« angegeben. Grund: einige Befehle verwenden das »reg«-Feld dieses Bytes (Bit 5-3) als »spec«-Feld und damit für die Erweiterung des Opcodes. Dies wird signalisiert, indem dem Schrägstrich eine Ziffer (0 – 7) folgt: »/5«.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
04	05	ADD acc, c	AL / AX / EAX	Const

In der Tabelle bedeuten in der Opcode-Spalte

/r Der Opcode besitzt ein *ModR/M*-Byte, da als Operanden sowohl ein Register als auch ein Register bzw. eine Speicherstelle angegeben werden müssen, die mittels der Felder *reg* und *r/m* kodiert werden.

/0 ... /7 Der Opcode besitzt ein *ModR/M*-Byte mit modifizierter Bedeutung: Das *reg*-Feld im *ModR/M*-Byte kann nur den spezifizierten Wert annehmen (z.B. 010_b bei /2) und dient der Erweiterung des Opcodes. Zur Kodierung von Operanden kann somit nur das *r/m*-Feld herangezogen werden. Das bedeutet entweder, dass es nur einen Operanden gibt, oder dass ein evtl. möglicher zweiter Operand nur eine Konstante sein kann, die nicht via *reg* kodiert werden muss.

+r Zu dem Byte des Opcodes wird ein Wert zwischen 0 und 7 addiert, der das Register kodiert, das im Befehl eine Rolle spielt:

Operandengröße	r =							
	0	1	2	3	4	5	6	7
8 Bit	AL	CL	DL	BL	AH	CH	DH	BH
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Ferner bedeuten in der Mnemonic-Spalte:

acc Akkumulator, also je nach Operandengröße das Register AL, AX oder EAX

c[x] Konstante der entsprechenden Operandengröße (8, 16 oder 32 Bit). Wird eine bestimmte Größe der Konstanten vorausgesetzt, wird sie als x angegeben, z.B. in c8.

creg Kontrollregister (control register)

dreg Debugregister

sreg 16-Bit-Segmentregister

m[x] Speicherstelle, die ein Datum der entsprechenden Operandengröße (8, 16 oder 32 Bit) aufnimmt. Wird eine bestimmte Operandengröße vorausgesetzt, wird sie als x angegeben, z.B. in m8.

r[x] Register der entsprechenden Operandengröße (8, 16 oder 32 Bit). Wird eine bestimmte Größe des Registers vorausgesetzt, wird sie als x angegeben, z.B. in r8.

r/m Der Operand kann entweder ein Register (siehe /r) oder eine Speicherstelle (siehe /m) sein.

pf Far Pointer: Der Operand ist ein direkt als Konstante angegebener 32- oder 48-Bit-Pointer, der zur absoluten Adressierung verwendet wird. Er besteht aus einem 16-Bit-Selektor und je nach Umgebung (Segmentattribut und address size override prefix) einem 16- oder 32-Bit-Offset. Dieser Pointer wird direkt als 32- bzw. 48-Bit-Konstante der Form 16:16 bzw. 16:32 dem Befehl übergeben. ACHTUNG: Aufgrund der Intel-Konvention heißt das für die Befehlssequenz, dass sich der 16- bzw. 32-Bit-Offset unmittelbar an den Opcode anschließt, dann erst folgt der 16-Bit-Selektor: `Opcode Offset Segment`

pn Near Pointer: Der Operand ist ein direkt als Konstante angegebener, je nach Umgebung (Segmentattribut und address size override prefix) 16 oder 32 Bit breiter Offset, der zur relativen Adressierung von der aktuellen Position verwendet wird. Er folgt unmittelbar dem Opcode.

ps Short Pointer: Der Operand ist ein direkt als Konstante angegebener 8-Bit-Offset, der zur relativen Adressierung von der aktuellen Position verwendet wird. Er folgt unmittelbar dem Opcode.

ps/n Je nach Opcode ist der Operand ein Near oder Short Pointer (pn oder ps).

Schließlich bedeuten in der Operanden-Spalte analog zur Mnemonic-Spalte:

Const[x] Konstante der entsprechenden Operandengröße (8, 16 oder 32 Bit). Wird eine bestimmte Größe der Konstanten vorausgesetzt, wird sie als x angegeben, z.B. in Const8.

CReg Kontrollregister (control register) mit den Mnemonics CR0, CR2, CR3 und CR4. CR1 ist zwar implementiert, ist aber nicht adressierbar.

DReg Debugregister mit den Mnemonics DR0 bis DR7

SReg 16-Bit-Segmentregister mit den Mnemonics DS, ES, FS, GS und SS

Mem[x] Speicherstelle, die ein Datum der entsprechenden Operandengröße (8, 16 oder 32 Bit) aufnimmt. Wird eine bestimmte Operandengröße vorausgesetzt, wird sie als x angegeben, z.B. in Mem8, womit ein Byte adressiert wird. In der Befehlssequenz stellt Mem eine effektive Adresse, also einen Offset dar, der als Konstante hinter den Opcode geschrieben wird. Die Größe dieses Offsets ist abhängig vom aktuellen Betriebsmodus, der aktuellen Umgebung und dem address size override prefix. Sie kann 16 oder 32 Bit betragen, sodass dem Opcode je nach Situation zwei oder vier Adressenbytes folgen. Falls sich der Offset auf ein anderes Segment bezieht, kann der Einsatz eines segment override prefix erforderlich werden.

Reg[x] Register der entsprechenden Operandengröße (8, 16 oder 32 Bit). Wird eine bestimmte Größe des Registers vorausgesetzt, wird sie als x angegeben, z.B. in Reg8.

Reg/Mem Der Operand kann entweder ein Register (siehe Reg) oder eine Speicherstelle (siehe Mem) sein.

Exceptions Die bei dem entsprechenden Befehl möglichen Gründe werden tabellarisch in Abhängigkeit des Betriebsmodus des Prozessors angegeben:

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Bemerkungen Hier werden verschiedene Bemerkungen oder Hinweise gegeben. Dieser Abschnitt ist optional und nicht bei allen Befehlen vorhanden.

2.1 Arithmetische Instruktionen

ADD	**8086**
ADC	**8086**
SUB	**8086**
SBB	**8086**

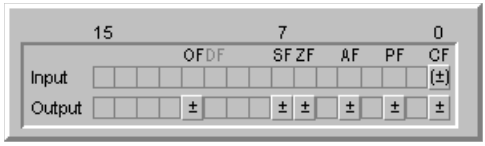
Funktion

Addiere (»add«, »add with carry«) bzw. subtrahiere (»subtract«, »subtract with borrow«) zwei Integers.

Operation

```
Temp := Dest           ; Temp doppelte Größe von Dest
IF (Instruction = ADD) THEN
   Temp := Temp + Src
ELSEIF (Instruction = ADC) THEN
   Temp := Temp + (Src + CF)
ELSEIF (Instruction = SUB) THEN
   Temp := Temp - Src
ELSE                   ; Instruction = SBB
   Temp := Temp – (Src + CF)
; Flags setzen:
IF (High(Temp) ≠ 0) THEN
   CF := 1             ; (Temp > Max) oder (Temp < Min)
ELSE
   CF := 0
IF (Temp[MSB] = Dest[MSB]) THEN
   OF := 0             ; kein Übertrag aus dem /in das MSB
ELSE
   OF := 1             ; Übertrag aus dem / in das MSB!
IF (Temp[4] = Dest[4]) THEN
   AF := 0             ; kein Übertrag aus dem /in das Nibble
ELSE
   AF := 1             ; Übertrag aus dem / in das Nibble!
IF (Temp = 0) THEN
   ZF := 1
ELSE
   ZF := 0
PF := 1                ; Annahme: Null = gerade Parität
FOR (I := 0) TO (I = OperandSize - 1) DO
   PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
                       ; Ergebnis setzen
Dest := Low(Temp)      ; untere Hälfte von Temp
SF := Dest[MSB]        ; SF = Kopie vom MSB des Zieloperanden
```

Statusflags



Input: nur bei ADC und SBB.

Output: ±: gesetzt oder gelöscht gemäß Ergebnis.

Opcodes Bei allen Befehlen handelt es sich um einen Ein-Byte-Opcode. Ist der erste Operand impliziert und stellt den Akkumulator dar, so bleibt es bei diesem Byte. Andernfalls folgt ein ModR/M-Byte, das für die Kodierung des ersten und zweiten Operanden sorgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
04	05	ADD acc, c	AL/AX/EAX	Const
00 /r	01 /r	ADD r/m, r	Reg / Mem	Reg
02 /r	03 /r	ADD r, r/m	Reg	Reg / Mem
80 /0	81 /0	ADD r/m, c	Reg / Mem	Const
-	83 /08	ADD r/m, c8	Reg / Mem	Const8
14	15	ADC acc, c	AL/AX/EAX r	Const
10 /r	11 /r	ADC r/m, r	Reg / Mem	Reg
12 /r	13 /r	ADC r, r/m	Reg	Reg / Mem
80 /2	81 /2	ADC r/m, c	Reg / Mem	Const
-	83 /2	ADC r/m, c8	Reg / Mem	Const8
1C	1D	SBB acc, c	AL/AX/EAX	Const
18 /r	19 /r	SBB r/m, r	Reg / Mem	Reg
1A /r	1B /r	SBB r, r/m	Reg	Reg / Mem
80 /3	81 /3	SBB r/m, c	Reg / Mem	Const
-	83 /3	SBB r/m, c8	Reg / Mem	Const8
2C	2D	SUB acc, c	AL/AX/EAX	Const
28 /r	29 /r	SUB r/m, r	Reg / Mem	Reg
2A /r	2B /r	SUB r, r/m	Reg	Reg / Mem
80 /5	81 /5	SUB r/m, c	Reg / Mem	Const
-	83 /5	SUB r/m, c8	Reg / Mem	Const8

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

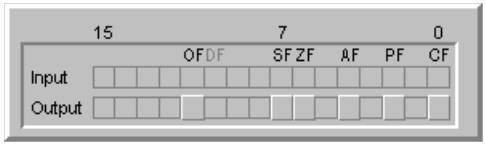
DIV 8086
IDIV 8086

Führe eine Division (»divide«, »integer divide«) mit zwei Integers durch. *Funktion*

Operation

```
IF Src = 0 THEN
   #DE
ELSE
   IF (Operandengröße = 8) THEN
      Temp := AX / Src *)
      IF (Temp > $7F) OR (Temp < $80) THEN
         #DE
      ELSE
         AH := AX MOD Src *)
         AL := Temp
   ELSEIF (Operandengröße = 16) THEN
      Temp := DX:AX / Src )
      IF (Temp > $7FFF) OR (Temp < $8000) THEN
         #DE
      ELSE
         DX := DX:AX MOD Src *)
         AX := Temp
   ELSE                ; Operandengröße = 32
      Temp := EDX:EAX / Src *)
      IF (Temp > $7FFFFFFF) OR (Temp < $80000000) THEN
         #DE
      ELSE
         EDX := EDX:EAX MOD Src *)
         EAX := Temp

*) bei DIV: unsigned division und modulus, bei IDIV: signed division
und modulus.
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Zieloperand (Dest) ist implizit vorgegeben, es handelt sich bei 8-Bit-Operanden um das AX-Register, bei 16-Bit-Operanden um die Registerkombination DX:AX und bei 32-Bit-Operanden um die Registerkombination EDX:EAX. Der Quelloperand (Src) ist explizit angegeben und bestimmt die Operandengröße.

Opcode		Mnemonic	Operanden		
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src1	Src2
F6 /6	F7 /7	DIV r/m	AL/AH	AX	Reg /
F6 /7	F7 /7	IDIV r/m	AX/DX	DX:AX	Mem
			EAX/EDX	EDX:EAX	

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine devide error exception #DE, eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#DE	2, 3	-	2, 3	-	2, 3
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

MUL 8086
IMUL 8086

Multipliziere (»multiply«, »integer multiply«) zwei Integers miteinander. *Funktion*

Operation

```
MUL:
IF (Operandengröße = 8) THEN
   AX := AL * Src
   IF (AH = $00) THEN
      CF := 0
      OF := 0
   ELSE
      CF := 1
      OF := 1
ELSEIF (Operandengröße = 16) THEN
   DX:AX := AX * Src
   IF (DX = $0000) THEN
      CF := 0
      OF := 0
   ELSE
      CF := 1
      OF := 1
ELSE           ; Operandengröße = 32
   EDX:EAX := EAX * SRC
   IF (EDX = $00000000) THEN
      CF := 0
      OF := 0
   ELSE
      CF := 1
      OF := 1

IMUL (Ein-Operanden-Form)
IF (Operandengröße = 8) THEN
   AX := AL * Src
   IF (AH = $00) OR (AH = $FF) THEN
      CF := 0
      OF := 0
   ELSE
      CF := 1
      OF := 1
```

```
ELSEIF (Operandengröße = 16) THEN
   DX:AX := AX * Src
   IF (DX = $0000) OR (DX = $FFFF) THEN
      CF := 0
      OF := 0
   ELSE
      CF := 1
      OF := 1
ELSE           ; Operandengröße = 32
   EDX:EAX := EAX * SRC
   IF (EDX = $00000000) OR (EDX = $FFFFFFFF) THEN
      CF := 0
      OF := 0
   ELSE
      CF := 1
      OF := 1

IMUL (Zwei-Operanden-Form)
Temp := Dest * Src     ; Temp doppelte Größe von Operanden
Dest := Low(Temp)      ; Low = untere Hälfte von Temp
IF (High(Temp) > 0) THEN
   CF := 0
   OF := 0
ELSE
   CF := 1
   OF := 1

IMUL (Drei-Operanden-Form)
Temp := Src1 * Src2    ; Temp doppelte Größe von Operanden
Dest := Low(Temp)      ; Low = untere Hälfte von Temp
IF (High(Temp) > 0) THEN
   CF := 0
   OF := 0
ELSE
   CF := 1
   OF := 1
```

Statusflags

Input: keiner.

Output: Anhand des Ergebnisses werden nur OF und CF gesetzt, alle anderen Flags gelten als undefiniert.

Opcodes

Der Zieloperand (Dest) ist bei MUL und der Ein-Operanden-Form von IMUL implizit vorgegeben, es handelt sich bei 8-Bit-Operanden um das AX-Register, bei 16-Bit-Operanden um die Registerkombination DX:AX und bei 32-Bit-Operanden um die Registerkombination EDX:EAX. Quelloperand 1, der Multiplikand, ist ebenfalls implizit vorgegeben, es handelt sich je nach Operandengröße um AL, AX bzw. EAX. Der zweite Quelloperand, der Multiplikator, ist entweder ein Register oder eine Speicherstelle der gleichen Größe.

In der Zwei-Operandenform von IMUL ist der Zieloperand und Quelle für den Multiplikanden ein 16- oder 32-Bit-Register, als Quelle für den Multiplikatoren kommen ein Register, eine Speicherstelle oder eine Konstante der gleichen Größe bzw. eine auf die entsprechende Größe vorzeichenerweiterte 8-Bit-Konstante in Betracht. Die Drei-Operanden-Form des IMUL-Befehls weicht davon in der Weise ab, dass nun das Ziel ein 16- oder 32-Bit-Register ist, der Multiplikand als zweiter Parameter durch ein gleich großes Register oder eine Speicherstelle der gleiche Größe übergeben wird und Multiplikator eine Konstante mit ebenfalls der gleichen Größe ist.

Opcode			Operanden		
8-Bit-Operanden	**32(16)-Bit-Op.**	**Mnemonic**	**Dest**	**Src1**	**Src2**
`F6 /4` `F6 /5`	`F7 /4` `F7 /5`	MUL r/m IMUL r/m	AX, DX:AX, EDX:EAX	AL, AX, EAX	Reg / Mem
	`0F AF /r`	IMUL r, r/m	Reg		Reg / Mem
	`69 /r`	IMUL r, c	Reg		Const
	`6B /r`	IMUL r, c8	Reg		Const8
	`69 /r`	IMUL r, r/m, c	Reg	Reg/Mem	Const
	`6B /r`	IMUL r, r/m, c8	Reg	Reg/Mem	Const8

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

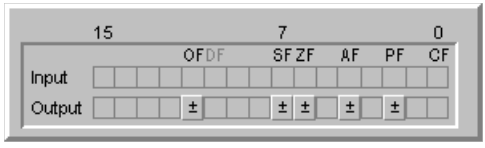
DEC 8086
INC 8086

Funktion Erhöhe (»increment«) oder erniedrige (»decrement«) den Wert einer Zahl um 1.

Operation

```
IF (Instruction = DEC) THEN
   Dest := Dest - 1
ELSE                    ; (Instruction = INC)
   Dest := Dest + 1
```

Statusflags



Input: keiner

Output: Bis auf das carry flag werden alle Statusflags anhand des Ergebnisses gesetzt.

Opcodes DEC und INC gibt es in Form eines Ein-Byte-Opcodes mit und ohne ModR/M-Byte. Der Ein-Byte-Opcode ohne ModR/M-Byte ist sehr effektiv, lässt aber nur die Wahl eines Registers als Quell- und Zieloperanden zu. Der Opcode mit ModR/M-Byte ermöglicht auch Speicherstellen als Operanden.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	(48+r)	DEC r	Reg	
FE /1	FF /1	DEC r/m	Reg / Mem	
	(40+r)	INC r	Reg	
FE /0	FF /0	INC r/m	Reg / Mem	

Im Falle des Ein-Byte-Codes sind die Register, die der Befehl adressieren kann, wie folgt kodiert:

Operandengröße	r =							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

NEG — 8086

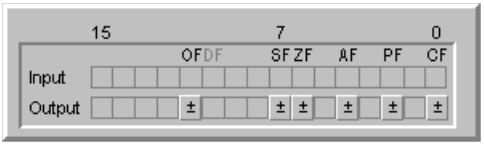
Funktion

Bilde das 2er-Komplement (»negate«) einer Zahl (»Vorzeichenwechsel«).

Operation

```
IF (Dest = 0) THEN
   CF := 0
ELSE
   CF := 1
Dest := -Dest
```

Statusflags



Input: keiner

Output: Alle Statusflags werden anhand des Ergebnisses gesetzt.

Opcodes

NEG hat nur einen Operanden, der gleichzeitig Quell- und Zieloperand ist.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
F6 /3	F7 /3	NEG r/m	Reg / Mem	

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS)

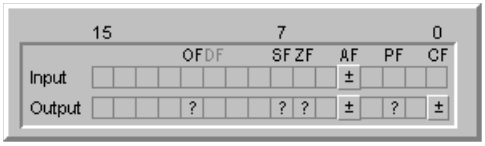
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

AAA 8086
AAS 8086

Funktion Korrigiere das Ergebnis einer Addition (»ASCII adjust after addition«) oder Subtraktion (»ASCII adjust after subtraction«) bei ungepackten BCDs (binary coded decimals).

Operation

```
AAA:
IF (AL > $0F) OR (AF = 1) THEN
   AL := AL + $06
   AH := AH + 1
   AF := 1
   CF := 1
ELSE
   AF := 0
   CF := 0
AL := AL AND $0F

AAS:
IF (AL > $0F) OR (AF = 1) THEN
   AL := AL - $06
   AH := AH - 1
   AF := 1
   CF := 1
ELSE
   AF := 0
   CF := 0
AL := AL AND $0F
```



Statusflags

Input: AF aus vorangehender Operation
Output: ±: gesetzt oder gelöscht gemäß Ergebnis,
?: undefiniert.

Opcodes

Diese beiden Befehle haben nur einen implizierten Operanden, der gleichzeitig Ziel und Quelle der Operation ist: der 8-Bit-Akkumulator AL, der jeweils ein Nibble enthält. Sollte ein Übertrag erforderlich sein, erfolgt er in das AH-Register, nicht etwa in das zweite Nibble von AL!

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
37		AAA	AL (AH)	
3F		AAS	AL (AH)	

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Diese Befehle lösen keine Exceptions aus.

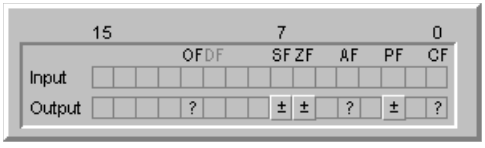
AAD 8086
AAM 8086

Funktion

Korrigiere das Ergebnis einer Multiplikation (»ASCII adjust after multiplication«) zweier ungepackten BCDs (binary coded decimals) bzw. bereite deren Division (»ASCII adjust before division«) vor.

Operation

```
AAD:
AL := (AL + (AH * Src)) AND $FF
AH := 0

AAM:
AH := AL DIV Src
AL := AL MOD Src
```



Statusflags

Input: keiner.
Output: ±: gesetzt oder gelöscht gemäß Ergebnis,
?: undefiniert.

Opcodes

AAM und AAD sind die Mnemonics für die Ein-Byte-Opcodes $D4 und $D5. Bei ihnen sind die Konstanten, die der Opcode $D4 bzw. $D5 benötigt, als Konstante $0A bereits vorgegeben. Daher sind in diesem Fall beide Operanden impliziert: Die 8-Bit-Register-Kombination

AH:AL fungiert sowohl als Ziel- als auch als erster Quelloperand, die Konstante $0A ist zweiter Quelloperand. In einer zweiten Version kann die Konstante jeweils frei gewählt werden. Allerdings gibt es für diesen Fall kein Mnemonic, weshalb dieser Opcode »von Hand« programmiert werden muss.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
D4 0A		AAM	AH:AL	$0A
D4		-	AH:AL	
D5 0A		AAD	AH:AL	$0A
D5		-	AH:AL	

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions AAD löst keine Exceptions aus, bei AAM kommt nur eine divide error exception #DE in Frage.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#DE	15, 23, 24, 26, 35	-	1	-	1

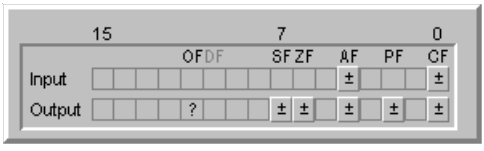
DAA 8086
DAS 8086

Funktion Korrigiere das Ergebnis einer Addition (»decimal adjust after addition«) oder Subtraktion (»decimal adjust after subtraction«) bei gepackten BCDs (binary coded decimals).

Operation

```
DAA:
IF ((AL AND $0F) > $0F) OR (AF = 1) THEN
   AL := AL + $06
   IF Überlauf THEN    ; Ergebnis > $FF?
      TempCarry := 1
   ELSE
      TempCarry := 0
   AF := 1
   CF := CF OR TempCarry
ELSE
   AF := 0
IF ((AL AND $F0 > $90) OR (CF = 1) THEN
   AL := AL + $60
   CF := 1
```

```
ELSE
   CF := 0

DAS:
IF ((AL AND $0F) > $09) OR (AF = 1) THEN
   AL := AL - $06
   IF Unterlauf THEN    ; Ergebnis < $00?
      TempCarry := 1
   ELSE
      TempCarry := 0
   AF := 1
   CF := CF OR TempCarry
ELSE
   AF := 0
IF (AL AND $F0) > $90) OR (CF = 1) THEN
   AL := AL - $60
   CF := 1
ELSE
   CF := 0
```



Statusflags

Input: CF und AF aus vorangehender Operation.
Output:±: gesetzt oder gelöscht gemäß Ergebnis,
?: undefiniert.

Opcodes

Die beiden Befehle DAA und DAS haben nur einen implizierten Operanden, der gleichzeitig Ziel und Quelle der Operation ist: den 8-Bit-Akkumulator AL, der jeweils zwei Nibbles enthält. Ein Übertrag in AH wie im Falle AAA und AAS erfolgt nicht!

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
27		DAA	AL	
2F		DAS	AL	

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Diese Befehle lösen keine Exceptions aus.

2.2 Logische Instruktionen

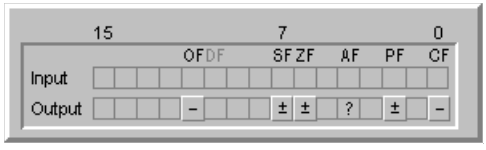
AND **8086**
OR **8086**
XOR **8086**

Funktion Führe eine logische UND-, ODER- bzw. Exklusiv-ODER-Verknüpfung zweier im Integer-Format dargestellten Werte durch.

Operation

```
For (I = 0) TO (I = OperandSize – 1) DO
  IF (Instruction = AND) THEN
      Dest[I] := Dest[I] AND Src[I]
   ELSEIF (Instruction = OR) THEN
      Dest[I] := Dest[I] OR Src[I]
   ELSE                ; (Instruction = XOR)
      Dest[I] := Dest[I] XOR Src[I]
OF := 0
CF := 0
IF (Dest = 0) THEN
   ZF := 1
ELSE
   ZF := 0
SF := Dest[MSB]
PF := 1                ; Annahme: Null = gerade Parität
FOR (I := 0) TO (I = OperandSize - 1) DO
   PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
```

Statusflags



Input: keiner

Output: ±: gesetzt oder gelöscht gemäß Ergebnis,
-: gelöscht, ?: undefiniert.

Opcodes Bei allen Befehlen gibt es eine Ein-Byte-Opcode-Version ohne ModR/M-Byte. Der erste Operand ist hier impliziert und stellt den Akkumulator dar. Daneben gibt es eine Opcode-Form mit ModR/M-Byte, das für die Kodierung des ersten und zweiten Operanden sorgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
24	25	AND acc, c	AL/AX/EAX	Const
20 /r	21 /r	AND r/m, r	Reg / Mem	Reg
22 /r	23 /r	AND r, r/m	Reg	Reg / Mem

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
80 /4	81 /4	AND r/m, c	Reg / Mem	Const
-	83 /4	AND r/m, c8	Reg / Mem	Const8
0C	0D	OR acc, c	AL/AX/EAX	Const
08 /r	09 /r	OR r/m, r	Reg / Mem	Reg
0A /r	0B /r	OR r, r/m	Reg	Reg / Mem
80 /1	81 /1	SBB r/m, c	Reg / Mem	Const
-	83 /1	SBB r/m, c8	Reg / Mem	Const8
34	35	XOR acc, c	AL/AX/EAX	Const
30 /r	31 /r	XOR r/m, r	Reg / Mem	Reg
32 /r	33 /r	XOR r, r/m	Reg	Reg / Mem
80 /6	81 /6	XOR r/m, c	Reg / Mem	Const
-	83 /6	XOR r/m, c8	Reg / Mem	Const8

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

NOT — 8086

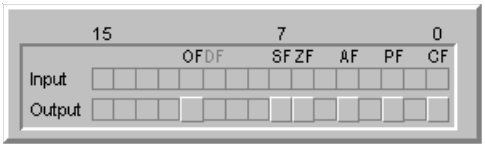
Funktion

Bilde das 1er-Komplement (»NOT«) einer Zahl (logische »Negierung«).

Operation

```
For (I = 0) TO (I = OperandSize – 1) DO
   Dest[I] := NOT Dest[I]
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Für NOT gibt es im Gegensatz zu den anderen logischen Instruktionen die Ein-Byte-Opcode-Form, in der ein ModR/M-Byte eingesetzt wird, das für die Kodierung des einzigen Operanden sorgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
F6 /2	F7 /2	NOT r/m	Reg / Mem	

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

2.3 Instruktionen zum Datenvergleich

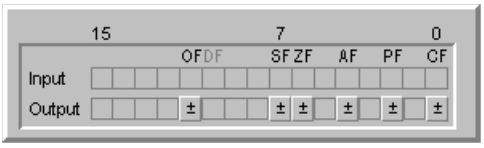
CMP 8086

Funktion Vergleiche (»compare«) zwei Integers arithmetisch miteinander.

Operation

```
Temp := Src1 – Src2     ; Temp hat doppelte Größe wie Operanden
IF (High(Temp) ≠ 0) THEN
   CF := 1              ; (Temp > Max) oder (Temp < Min)
ELSE
   CF := 0
IF (Temp[MSB] = Src1[MSB]) THEN
   OF := 0              ; kein Übertrag aus dem /in das MSB
ELSE
   OF := 1              ; Übertrag aus dem / in das MSB!
IF (Temp[4] = Src1[4]) THEN
   AF := 0              ; kein Übertrag aus dem /in das Nibble
ELSE
   AF := 1              ; Übertrag aus dem / in das Nibble!
IF (Temp = 0) THEN
   ZF := 1
ELSE
   ZF := 0
```

```
PF := 1                 ; Annahme: Null = gerade Parität
FOR (I := 0) TO (I = OperandSize - 1) DO
   PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
SF := Dest[MSB]         ; SF = Kopie vom MSB des Zieloperanden
```



Statusflags

Input: keiner.
Output: ±: gesetzt oder gelöscht gemäß Ergebnis des Vergleichs.

Opcodes

Es gibt eine Ein-Byte-Opcode-Version, der eine Konstante als zweiter Operand folgt. Der erste Operand ist hier impliziert und stellt den Akkumulator dar. Daneben gibt es eine Ein-Byte-Opcode-Form, in der ein ModR/M-Byte eingesetzt wird, das für die Kodierung des ersten und zweiten Operanden sorgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Src1**	**Src2**
3C	3D	CMP acc, c	AL/AX/EAX	Const
38 /r	39 /r	CMP r/m, r	Reg / Mem	Reg
3A /r	3B /r	CMP r, r/m	Reg	Reg / Mem
80 /7	81 /7	CMP r/m, c	Reg / Mem	Const
-	83 /7	CMP r/m, c8	Reg / Mem	Const8

Exceptions

Die bei diesem Befehl möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

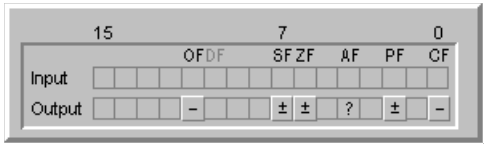
TEST 8086

Funktion Vergleiche (»test«) zwei als Integer dargestellte Werte logisch miteinander.

Operation

```
Temp := Src1 AND Src2
CF := 0
OF := 0
IF (Temp = 0) THEN
   ZF := 1
ELSE
   ZF := 0
PF := 1                ; Annahme: Null = gerade Parität
FOR (I := 0) TO (I = OperandSize - 1) DO
   PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
SF := Temp[MSB]        ; SF = Kopie vom MSB des Zieloperanden
```

Statusflags



Input: keiner.
Output: ±: gesetzt oder gelöscht gemäß Ergebnis des Vergleichs. CF und OF werden explizit gelöscht.

Opcodes Es gibt eine Ein-Byte-Opcode-Version, der eine Konstante als zweiter Operand folgt. Der erste Operand ist hier impliziert und stellt den Akkumulator dar. Daneben gibt es eine Ein-Byte-Opcode-Form, in der ein ModR/M-Byte eingesetzt wird, das für die Kodierung des ersten und zweiten Operanden sorgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Src1	Src2
A8	A9	TEST acc, c	AL/AX/EAX	Const
84 /r	85 /r	TEST r/m, r	Reg / Mem	Reg
F6 /0	F7 /0	TEST r/m, c	Reg / Mem	Const

Exceptions Die bei diesem Befehl möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

2.4 Bit-orientierte Instruktionen

SAL **8086**
SAR **8086**
SHL **8086**
SHR **8086**

Funktion

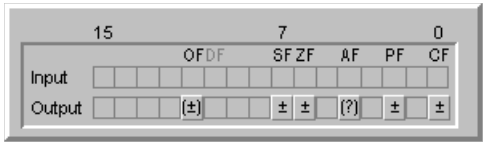
Verschiebe bitweise eine als Integer dargestellte Zahl arithmetisch (= Multiplikation bzw. Division mit bzw. durch eine Potenz von 2; »shift arithmetically left or right«) oder logisch (»shift [logically] left or right«) um eine bestimmte Anzahl von Stellen nach links oder rechts.

Operation

```
IF (Src = 0) THEN
   EXIT                  ; Keine Aktion, Flags nicht verändert
Count := Src AND $1F
WHILE (Count ≠ 0) DO
   IF (Instruction = SAL) OR (Instruction = SHL) THEN
      CF := Dest[MSB]
      Dest := Dest * 2
   ELSEIF (Instruction = SAR) THEN
      CF := Dest[LSB]
      Dest := Dest DIV 2 ; signed division, rounding → -∞
   ELSE                  ; (Instruction = SHR)
      CF := Dest[LSB]
      Dest := Dest DIV 2 ; unsigned division
   Count := Count – 1
IF (Src = 1) THEN
   IF (Instruction = SAL) OR (Instruction = SHL) THEN
      OF := Dest[MSB] XOR CF
   ELSEIF (Instruction = SAR) THEN
      OF := 0
   ELSE                  ; (Instruction = SHR)
      OF := Dest[MSB]
ELSE                     ; (Src > 1)
   OF := undefiniert
```

```
IF (Dest = 0) THEN
   ZF := 1
ELSE
   ZF := 0
PF := 1                 ; Annahme: Null = gerade Parität
FOR (I := 0) TO (I = OperandSize - 1) DO
   PF := PF XOR Dest[I]; wenn Bit gesetzt, Parität ändern
SF := Dest[MSB]         ; SF = Kopie vom MSB des Zieloperanden
AF := undefiniert
```

Statusflags



Input: keiner.
Output: Wenn kein shift erfolgt, sind alle Flags unverändert, andernfalls siehe Angaben unter »Operation«.

Opcodes Bei allen Befehlen gibt es zwei Ein-Byte-Opcode-Versionen mit und ohne ModR/M-Byte. In der MODR/M-losen Form sind beide Operanden impliziert und stellen den Akkumulator und die Konstante 1 bzw. das CL-Register dar. In der zweiten Form wird ein ModR/M-Byte eingesetzt, das für die Kodierung des ersten Operanden sorgt, der zweite Operand ist eine Konstante.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
D0 /7	D1 /7	SAR r/m, 1	Reg /Mem	1
D2 /7	D3 /7	SAR r/m, CL	Reg / Mem	CL
C0 /7	C1 /7	SAR r/m, c8	Reg / Mem	Const8
D0 /4	D1 /4	SHL r/m, 1	Reg /Mem	1
D2 /4	D3 /4	SHL r/m, CL	Reg / Mem	CL
C0 /4	C1 /4	SHL r/m, c8	Reg / Mem	Const8
D0 /5	D1 /5	SHR r/m, 1	Reg /Mem	1
D2 /5	D3 /5	SHR r/m, CL	Reg / Mem	CL
C0 /5	C1 /5	SHR r/m, c8	Reg / Mem	Const8

Die grau unterlegten Operanden sind implizit vorgegeben, müssen aber im Mnemonic angegeben werden.

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

SHLD 80386
SHRD 80386

Funktion

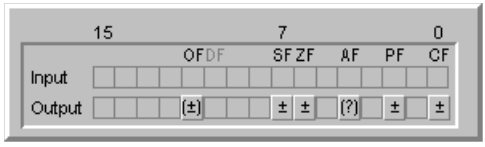
Verschiebe eine als Integer dargestellte Zahl logisch um eine bestimmte Anzahl von Positionen nach links oder rechts (»double-precision shift [logically] left or right«) und fülle die entstehenden Freistellen aus einer anderen Integer auf.

Operation

```
Count := Src2 AND $1F
Size := OperandSize
IF (Count = 0) THEN
   EXIT                  ; Keine Aktion, Flags nicht verändert
IF (Count > OperandSize) THEN
   Dest := undefiniert
   CF = OF = SF = AF = PF := undefiniert
   EXIT
IF (Instruction = SHLD) THEN
   CF := Dest[Size – Count]
   FOR (I = Size – 1) DOWNTO (I = COUNT) DO
      Dest[I] := Dest[I – Count]
   FOR (I = Count – 1) DOWNTO (I = 0) DO
      Dest[I] := Src1[I + Size - Count]
ELSE                     ; (Instruction = SHRD)
   CF := Dest[Count – 1)
   FOR (I = 0) TO (I = Size – 1 – Count) DO
      Dest[I] := Dest[I + Count]
   FOR (I = Size – Count) TO (I = Size – 1) DO
      DEST[I] := Src[I + Count – Size]
IF (Src2 = 1) THEN
   OF := Dest[MSB] XOR CF
ELSE                     ; (Src2 > 1)
   OF := undefiniert
```

```
IF (Dest = 0) THEN
   ZF := 1
ELSE
   ZF := 0
PF := 1                 ; Annahme: Null = gerade Parität
FOR (I := 0) TO (I = OperandSize - 1) DO
   PF := PF XOR Dest[I]; wenn Bit gesetzt, Parität ändern
SF := Dest[MSB]         ; SF = Kopie vom MSB des Zieloperanden
AF := undefiniert
```

Statusflags



Input: keiner.
Output: Wenn kein shift erfolgt, sind alle Flags unverändert, andernfalls siehe Angaben unter »Operation«.

Opcodes Bei den Befehlen handelt es sich um Zwei-Byte-Opcodes mit ModR/M-Byte; Byte 1 ist immer das Opcode-Shift-Byte $0F. Die Befehle erwarten drei Operanden, wobei Operand 1 das Ziel der Operation ist und Operand 2 und 3 die beiden Quelloperanden darstellen. Operand 3 ist immer eine Konstante, die entweder im Register CL angegeben wird oder als Konstante dem Opcode folgt.

Opcode		Mnemonic	Operanden		
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src1	Src2
-	0F A5 /r	SHLD r/m, r, CL	Reg / Mem	Reg	CL
-	0F A4 /r	SHLD r/m, r, c8	Reg / Mem	Reg	Const8
-	0F AD /r	SHRD r/m, r, CL	Reg / Mem	Reg	CL
-	0F AC /r	SHRD r/m, r, c8	Reg / Mem	Reg	Const8

Die grau unterlegten Operanden sind implizit vorgegeben, müssen aber im Mnemonic angegeben werden.

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

RCL 8086
RCR 8086
ROL 8086
ROR 8086

Funktion

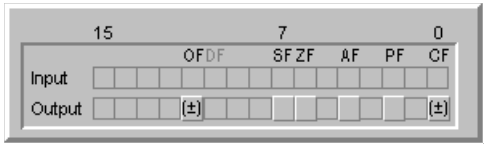
Verschiebe die Bits eines als Integer dargestellten Wertes mit oder ohne Beteiligung des carry flags zyklisch um eine bestimmte Anzahl von Stellen nach links oder rechts (»rotate with carry left or right«, »rotate [without carry] left or right«).

Operation

```
IF (Instruction = RCL) OR (Instruction = RCR) THEN
   IF (OperandSize = 8) THEN
      Count := (Src AND $1F) MOD 9
   ELSEIF (OperandSize = 16) THEN
      Count := (Src AND $1F) MOD 17
   ELSE                    ; (OperandSize = 32)
      Count := Src AND $1F
ELSE                       ; (Instruction = ROL oder ROR)
   IF (OperandSize = 8) THEN
      Count := (Src MOD 8)
   ELSEIF (OperandSize = 16) THEN
      Count := (Src MOD 16)
   ELSE                    ; (OperandSize = 32)
      Count := Src AND $1F
IF (Instruction = RCL) THEN
   WHILE (Count ≠ 0) DO
      TempCarry := Dest[MSB]
      Dest := (Dest * 2) + CF
      CF := TempCarry
      Count := Count – 1
   IF (Src = 1) THEN
      OF := Dest[MSB] XOR CF
   ELSE
      OF := undefiniert
```

```
ELSEIF (Instruction = RCR) THEN
   IF (Src = 1) THEN
      OF := Dest[MSB] XOR CF
   ELSE
      OF := undefiniert
   WHILE (Count ≠ 0) DO
      TempCarry := Dest[LSB]
      Dest := (Dest / 2) + (CF * 2^OperandSize)
      CF := TempCarry
      Count := Count – 1
ELSEIF (Instruction = ROL) THEN
   WHILE (Count ≠ 0) DO
      TempBit := Dest[MSB]
      Dest := (Dest * 2) + TempBit
      Count := Count – 1
   CF := Dest[LSB]
   IF (Src = 1) THEN
      OF := Dest[MSB] XOR CF
   ELSE
      OF := undefiniert
ELSE                    ; (Instruction = ROR)
   WHILE (Count ≠ 0) DO
      TempBit := Dest[LSB]
      Dest := (Dest * 2) + (TempBit * 2^OperandSize)
      Count := Count – 1
   CF := Dest[MSB]
   IF (Src = 1) THEN
      OF := Dest[MSB] XOR Dest[MSB-1]
   ELSE
      OF := undefiniert
```

Statusflags



Input: keiner.
Output: Zu OF und CF siehe Angaben unter »Operation«; SF, ZF, AF und PF werden nicht verändert.

Opcodes Bei allen Befehlen gibt es zwei Ein-Byte-Opcode-Versionen ohne ModR/M-Byte. Beide Operanden sind hier impliziert und stellen den Akkumulator und die Konstante 1 bzw. das CL-Register dar. Daneben gibt es eine Opcode-Form, in der ein ModR/M-Byte eingesetzt wird, das für die Kodierung des ersten Operanden sorgt, der zweite Operand ist eine Konstante.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
D0 /2	D1 /2	RCL r/m, 1	Reg /Mem	1
D2 /2	D3 /2	RCL r/m, CL	Reg / Mem	CL
C0 /2	C1 /2	RCL r/m, c8	Reg / Mem	Const8
D0 /3	D1 /3	RCR r/m, 1	Reg /Mem	1
D2 /3	D3 /3	RCR r/m, CL	Reg / Mem	CL
C0 /3	C1 /3	RCR r/m, c8	Reg / Mem	Const8
D0 /0	D1 /0	ROL r/m, 1	Reg /Mem	1
D2 /0	D3 /0	ROL r/m, CL	Reg / Mem	CL
C0 /0	C1 /0	ROL r/m, c8	Reg / Mem	Const8
D0 /1	D1 /1	ROR r/m, 1	Reg /Mem	1
D2 /1	D3 /1	ROR r/m, CL	Reg / Mem	CL
C0 /1	C1 /1	ROR r/m, c8	Reg / Mem	Const8

Die grau unterlegten Operanden sind implizit vorgegeben, müssen aber im Mnemonic angegeben werden.

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

BSF 80386
BSR 80386

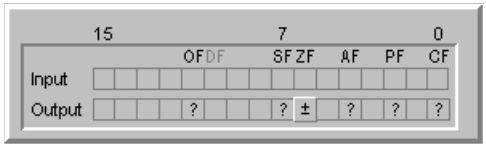
Funktion

Suche das erste gesetzte Bit in einem als Integer dargestellten Wert unter Berücksichtigung der Suchrichtung (»bit scan forward or reverse«).

Operation

```
IF (Src = 0) THEN      ; kein Bit gesetzt
   ZF := 1
   Dest := undefiniert
ELSE                   ; mindestens ein Bit gesetzt
   ZF := 0
   IF (Instruction = BSF) THEN
      Dest := 0
      WHILE (Dest < OperandSize) AND (Src[Dest] = 0) DO
         Dest := Dest + 1
   ELSE                ; (Instruction = BSR)
      Dest := OperandSize – 1
      WHILE (Dest > 0) AND (Src[Pos] = 0) DO
         Dest := Dest - 1
```

Statusflags



Wenn ein Bit gefunden wurde, ist ZF gesetzt, andernfalls gelöscht. Alle anderen Flags sind undefiniert.

Opcodes Bei den Befehlen handelt es sich um Zwei-Byte-Opcodes mit ModR/M-Byte; Byte 1 ist immer das Opcode-Shift-Byte $0F. Die Befehle erwarten zwei Operanden, wobei Operand 1 das Ziel der Operation ist und Operand 2 den Quelloperanden darstellt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
-	0F BC /r	BSF r, r/m	Reg	Reg / Mem
-	0F BD /r	BSR r, r/m	Reg	Reg / Mem

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

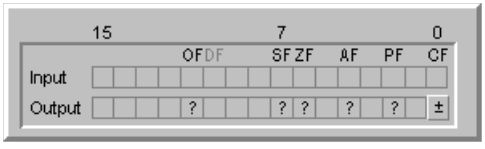
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

BT **80386**
BTC **80386**
BTR **80386**
BTS **80386**

Prüfe den Zustand eines Bits (»bit test«) in einem als Integer dargestellten Wert und verändere dann ggf. seinen Zustand (»complement, reset, set«). *Funktion*

Operation

```
CF := Dest[Src]
IF (Instruction = BT) THEN
   Nothing
ELSEIF (Instruction = BTS) THEN
   Dest[Src] := 1
ELSEIF (Instruction = BTC) THEN
   Dest[Src] := 0
ELSE                    ; (Instruction = BTC)
   Dest[Src] := NOT Dest[Src]
```



Statusflags

Das CF erhält den Zustand des geprüften Bits. Alle anderen Flags sind undefiniert.

Opcodes

Bei den Befehlen handelt es sich um Zwei-Byte-Opcodes mit ModR/M-Byte; Byte 1 ist immer das Opcode-Shift-Byte $0F. Die Befehle erwarten zwei Operanden, wobei Operand 1 das Ziel der Operation ist und Operand 2 den Quelloperanden darstellt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F A3/r	BT r/m, r	Reg /Mem	Reg
	0F BA /4	BT r/m, c8	Reg / Mem	Const8
	0F BB /r	BTC r/m, r	Reg /Mem	Reg
	0F BA /7	BTC r/m, c8	Reg / Mem	Const8
	0F B3/r	BTR r/m, r	Reg /Mem	Reg
	0F BA /6	BTR r/m, c8	Reg / Mem	Const8
	0F AB /r	BTS r/m, r	Reg /Mem	Reg
	0F BA /5	BTS r/m, c8	Reg / Mem	Const8

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, (20)	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

Da BT keinerlei schreibenden Zugriff auf eine Speicherstelle vornimmt, entfällt hier der Grund 20 für eine #GP im protected mode.

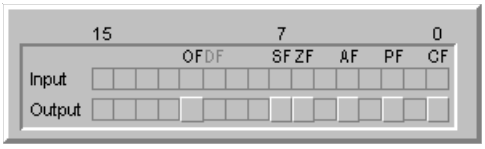
2.5 Instruktionen zum Datenaustausch

MOV **8086**
MOVSX **80386**
MOVZX **80386**

Funktion Kopiere den Inhalt einer Quelle in ein Ziel (»move«) und erweitere das Datum hierbei ggf. vorzeichenlos oder vorzeichenerweitert (»zero or sign extend«).

Operation
```
IF (Instruction = MOV) AND (Dest ≠ SegmentRegister) THEN
   Dest := Src
ELSEIF (Instruction = MOVSX) THEN
   Dest := SignExtend(Src)
ELSEIF (Instruction = MOVZX) THEN
   Dest := ZeroExtend(Src)
ELSE                    ; MOV mit SegmentRegister als Ziel
   SegmentSelector := Src
   IF (Dest = SS) THEN
      IF (SegmentSelector = NULL) THEN
         #GP(0)
      IF (SegmentSelector außerhalb DescriptorTable)
      OR (RPL ≠ CPL)
      OR (Segment kein beschreibbares Datensegment)
      OR (DPL ≠ CPL) THEN
         #GP(Selektor)
      IF (Segment nicht »present« markiert) THEN
         #SS(Selektor)
```

```
      Dest[visible] := SegmentSelektor
      Dest[hidden] := SegmentDescriptor
   IF (Dest = DS, ES, FS, GS) THEN
      IF (SegmentSelector = NULL) THEN
         Dest[visible] := NullSelector
         Dest[hidden] := NullDescriptor
      ELSE
         IF (SegmentSelector außerhalb DescriptorTable)
         OR ((Segment ist Daten- oder Nonconforming Codeseg.)
            AND ((RPL > DPL) OR (CPL > DPL)))
         OR (Segment kein Daten- oder lesbares Codeseg.) THEN
            #GP(Selektor)
         IF (Segment nicht »present« markiert) THEN
            #NP(Selektor)
      Dest[visible] := SegmentSelektor
      Dest[hidden] := SegmentDescriptor
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Bei allen Befehlen gibt es eine Ein-Byte-Opcode-Version, der eine Konstante als zweiter Operand folgt. Der erste Operand ist hier impliziert und stellt den Akkumulator dar. In einer weiteren Form kann das Register im Opcode kodiert werden, wenn der zweite Operand eine Konstante ist. Daneben gibt es eine Ein-Byte-Opcode-Form, in der ein ModR/M-Byte eingesetzt wird, das für die Kodierung des ersten und zweiten Operanden sorgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
A0	A1	MOV acc, m	AL/AX/EAX	Mem
A2	A3	MOV m, acc	Mem	AL/AX/EAX
88 /r	89 /r	MOV r/m, r	Reg / Mem	Reg
8A /r	8B /r	MOV r, r/m	Reg	Reg / Mem
(B0+r)	(B8+r)	MOV r, c	Reg	Const
C6 /0	C7 /0	MOV r/m, c	Reg / Mem	Const
	8C /r	MOV r/m, sreg	Reg16 / Mem16	SReg
	8E /r	MOV sreg, r/m	SReg	Reg16 / Mem16

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F BE /r	MOVSX r, r/m	Reg	Reg8 / Mem8
	0F BF /r	MOVSX r, r/m	Reg	Reg16 / Mem16
	0F B6 /r	MOVZX r, r/m	Reg	Reg8 / Mem8
	0F B7 /r	MOVZX r, r/m	Reg	Reg16 / Mem16
	0F 20 /r	MOV r32, creg	Reg32	CReg
	0F 22 /r	MOV creg, r32	CReg	Reg32
	0F 21 /r	MOV r32, dreg	Reg32	DReg
	0F 23 /r	MOV dreg, r32	DReg	Reg32

Die grau unterlegten Opcodes/Mnemonics sind nur im Kernelmodus (Privilegstufe 0) verfügbar.

Im Falle des Ein-Byte-Codes mit Konstante als zweitem Operanden sind die Register, die der Befehl adressieren kann, wie folgt kodiert:

Operandengröße	r =							
	0	1	2	3	4	5	6	7
8 Bit	AL	CL	DL	BL	AH	CH	DH	BH
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS). Bei der #GP kann entweder der Wert 0 als ErrorCode übergeben werden oder der Selektor, der zur Exception führte. Analog kann auch über der #SS der ErrorCode 0 übergeben werden oder der auslösende Selektor.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 6, 8, 20, 32, 49, 50, 51	0	2, 8, 56	0	8, 50
	14, 21, 22, 41, 46	Selektor	-	Selektor	./.
#NP	4	Selektor	4	Selektor	./.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#PF	1	fault code	1	fault code	./.
#SS	1 7	0 Selektor	1 -	0 Selektor	1 ./.
#UD	8, 9	-	8, 9	-	8, 9

MOVNTI SSE2

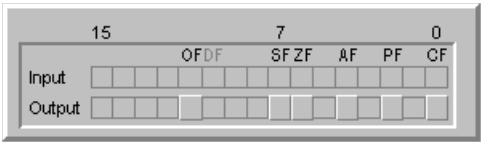
Funktion

Schreibe ein DoubleWord aus einem Allzweck-Register unter Umgehung des Cache in den Speicher (»move using non-temporal hint integer«)

Operation

```
Dest := Src
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Als Quelle kommt bei diesem Befehl lediglich ein Allzweckregister infrage, Ziel kann nur eine Speicherstelle sein.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F C3 /r	MOVNTI m, r	Mem32	Reg32

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

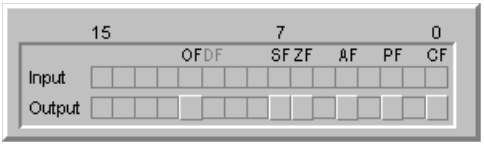
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1
#UD	14	-	14	-	14

XCHG 8086

Funktion Vertausche den Inhalt von Quelle und Ziel (»exchange«).

Operation
```
Temp := Dest
Dest := Src
Src := Temp
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes XCHG kann mit einem Ein-Byte-Code mit und ohne ModR/M-Byte kodiert werden. Beim Opcode ohne ModR/M-Byte ist nur der Austausch zwischen einem beliebigen Register und dem Akkumulator möglich. Soll eine Speicherstelle involviert werden oder ist nicht einer der Operanden der Akkumulator, so ist die Form mit ModR/M-Byte erforderlich. ACHTUNG: Die Unterscheidung zwischen den Mnemonics `XCHG acc,r` und `XCHG r,acc` bzw. `XCHG r/m,r` und `XCHG r,r/m` ist rein formaler Art und dient nur der Anwenderfreundlichkeit. Rein faktisch sind natürlich jeweils beide Formen identisch, weshalb auch die Opcodes jeweils identisch sind.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	(90+r)	XCHG acc, r	AL/AX/EAX	Reg
	(90+r)	XCHG r, acc	Reg	AL/AX/EAX
86 /r	87 /r	XCHG r/m, r	Reg / Mem	Reg
86 /r	87 /r	XCHG r, r/m	Reg	Reg / Mem

Im Falle des Ein-Byte-Codes ohne ModR/M-Byte sind die Register, die der Befehl adressieren kann, wie folgt kodiert:

Operandengröße	r =							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

BSWAP 80486

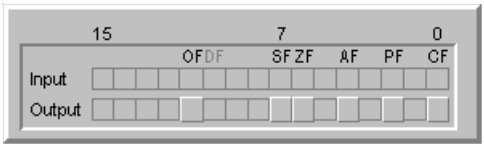
Funktion

Vertausche die Reihenfolge der Bytes eines DoubleWords (»byte swap«).

Operation

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

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

BSWAP kann nur mit 32-Bit-Registern in einer Zwei-Byte-Opcode-Form verwendet werden.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F (C8+r)	BSWAP r32	Reg32	

Die Register, die der Befehl adressieren kann, sind wie folgt kodiert:

Operandengröße	r =							
	0	1	2	3	4	5	6	7
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions BSWAP löst keine Exceptions aus.

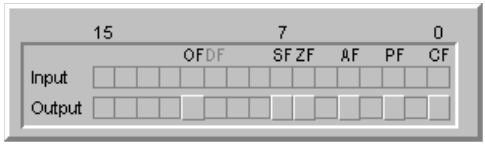
XLAT 8086
XLATB 8086

Funktion »Übersetze« (= ersetze) einen Index in eine Byte-Tabelle mit dem Tabellenwert, auf den er zeigt (»translate«).

Operation

```
IF (AddressSize = 16) THEN
  AL := Mem[DS:(BX + ZeroExtend(AL))]
ELSE                    ; (AddressSize = 32)
  AL := Mem[DS:(EBX + ZeroExtend(AL))]
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes XLAT ist eigentlich XLATB und hat nur zwei implizite Operanden: Einen Zeiger auf eine Speicherstelle in DS:(E)BX und ein Tabellenindex in AL, der auf den AL-ten Eintrag in der an DS:(E)BX stehenden Tabelle zeigt. Dennoch existiert auch die »parametrische« Form XLAT, bei der ein explizites Symbol, ein Dummy-Parameter eingesetzt wird, der keinerlei Auswirkungen hat. Beide Formen sind im Opcode absolut identisch.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
D7		XLAT m8	AL	Mem
D7		XLAT	AL	Mem

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS). *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

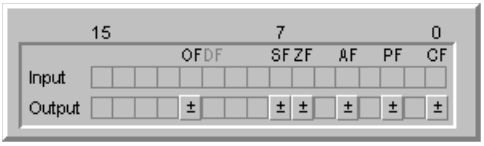
XADD 80486

Addiere zwei Integers und vertausche anschließend Ziel und Quelle (»exchange and add«). *Funktion*

Operation

```
Temp := Src + Dest      ; Temp hat doppelte Größe der Operanden
Src := Dest
IF (High(Temp) ≠ 0) THEN
   CF := 1              ; (Temp > Max) oder (Temp < Min)
ELSE
   CF := 0
IF (Temp[MSB] = Dest[MSB]) THEN
   OF := 0              ; kein Übertrag aus dem /in das MSB
ELSE
   OF := 1              ; Übertrag aus dem / in das MSB!
IF (Temp[4] = Dest[4]) THEN
   AF := 0              ; kein Übertrag aus dem /in das Nibble
ELSE
   AF := 1              ; Übertrag aus dem / in das Nibble!
IF (Temp = 0) THEN
   ZF := 1
ELSE
   ZF := 0
PF := 1                 ; Annahme: Null = gerade Parität
FOR (I := 0) TO (I = OperandSize - 1) DO
   PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
Dest := Low(Temp)       ; untere Hälfte von Temp
SF := Dest[MSB]         ; SF = Kopie vom MSB des Zieloperanden
```

Statusflags



Die Flags werden nach dem Ergebnis der Addition analog zu ADD gesetzt.

Opcodes XADD verwendet einen Zwei-Byte-Opcode mit ModR/M-Byte, bei dem Byte 1 das Opcode-Shift-Byte $0F ist.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
`0F C0 /r`	`0F C1 /r`	XADD r/m, r	Reg / Mem	Reg

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

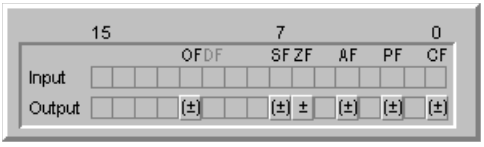
CMPXCHG 80486
CMPXCHG8B Pentium

Funktion Vergleiche zwei Daten miteinander und vertausche sie abhängig vom Ergebnis des Vergleichs (»compare and exchange«).

Operation

```
IF (Accumulator = Dest) THEN
   ZF := 1
   Dest := Src
ELSE                    ; (Accumulator ≠ Dest)
   ZF := 0
   Accumulator := Src
IF (Instruction = CMPXCHG) THEN
   IF (Temp[MSB] = Dest[MSB]) THEN
      OF := 0           ; kein Übertrag aus dem /in das MSB
   ELSE
      OF := 1           ; Übertrag aus dem / in das MSB!
   IF (Temp[4] = Dest[4]) THEN
      AF := 0           ; kein Übertrag aus dem /in das Nibble
```

```
   ELSE
      AF := 1          ; Übertrag aus dem / in das Nibble!
   PF := 1             ; Annahme: Null = gerade Parität
   FOR (I := 0) TO (I = OperandSize - 1) DO
      PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
   Dest := Low(Temp)   ; untere Hälfte von Temp
   SF := Dest[MSB]     ; SF = Kopie vom MSB des Zieloperanden
ELSE                   ; (Instruction = CMPXCHG8B)
   CF = PF = AF = SF = OF = unverändert
```



Statusflags

CMPXCHG: Alle Flags anhand des Vergleichs gesetzt; CMPXCHG8B: ZF gesetzt, alle anderen Flags unverändert.

Opcodes

Beide Befehle verwenden einen Zwei-Byte-Opcode mit ModR/M-Byte, bei dem Byte 1 das Opcode-Shift-Byte $0F ist. Bei CMPXCHG kann das Ziel angegeben werden, bei CMPXCHG8B ist es impliziert die Registerkombination EDX:EAX.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
`0F B0 /r`	`0F B1 /r`	CMPXCHG r/m, r	Reg / Mem	Reg
	`0F C7 /1`	CMPXCHG8B m64	EDX:EAX	Mem64

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

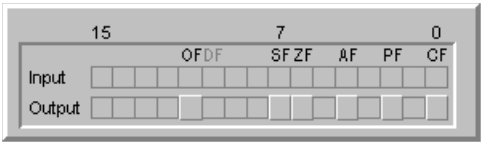
POP 8086

Funktion Hole ein Datum vom Stack (»pop stack«).

Operation

```
IF (Dest ≠ SReg) THEN
   IF (StackOperandSize = 32) THEN
      IF (OperandSize = 32) THEN
         Dest := DWORD[SS:ESP]
         ESP := ESP + 4
      ELSE             ; (OperandSize = 16)
         Dest := WORD[SS:ESP]
         ESP := ESP + 2
   ELSE                ; (StackOperandSize = 16)
      IF (OperandSize = 32) THEN
         Dest := DWORD[SS:SP]
         SP := SP + 4
      ELSE             ; (OperandSize = 16)
         Dest := WORD[SS:SP]
         SP := SP + 2
ELSE                   ; (Dest = SReg)
   SegmentSelector := Src
   IF (Dest = SS) THEN
      IF (SegmentSelector = NULL) THEN
         #GP(0)
      IF (SegmentSelector außerhalb DescriptorTable)
      OR (RPL ≠ CPL)
      OR (Segment kein beschreibbares Datensegment)
      OR (DPL ≠ CPL) THEN
         #GP(Selektor)
      IF (Segment nicht »present« markiert) THEN
         #SS(Selektor)
      Dest[visible] := SegmentSelektor
      Dest[hidden] := SegmentDescriptor
   IF (Dest = DS, ES, FS, GS) THEN
      IF (SegmentSelector = NULL) THEN
         Dest[visible] := SegmentSelector
         Dest[hidden] := NullDescriptor
      ELSE
         IF (SegmentSelector außerhalb DescriptorTable)
         OR ((Segment ist Daten- oder Nonconforming Codeseg.)
            AND ((RPL > DPL) OR (CPL > DPL)))
         OR (Segment kein Daten- oder lesbares Codeseg.) THEN
            #GP(Selektor)
         IF (Segment nicht »present« markiert) THEN
            #NP(Selektor)
```

```
Dest[visible] := SegmentSelektor
Dest[hidden] := SegmentDescriptor
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Bis auf den Fall POP FS und POP GS handelt es sich bei allen POP-Befehlen um Ein-Byte-Opcodes, denen nur im Falle einer Speicherbeteiligung ein ModR/M-Byte folgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	8F /0	POP m	Mem	SS:(E)SP
	(58+r)	POP r	Reg	SS:(E)SP
	1F	POP DS	SReg DS	SS:(E)SP
	07	POP ES	SReg ES	SS:(E)SP
	17	POP SS	SReg SS	SS:(E)SP
	0F A1	POP FS	SReg FS	SS:(E)SP
	0F A9	POP GS	SReg GS	SS:(E)SP

Die grau unterlegten Operanden sind implizit vorgegeben, müssen aber im Falle des Ziels explizit angegeben werden.

Im Falle des Ein-Byte-Codes mit einem Register als Ziel ist dieses wie folgt kodiert:

Operandengröße	r =							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS). Bei der #GP kann entweder der Wert 0 als ErrorCode übergeben werden oder

der Selektor, der zur Exception führte. Analog kann auch über der #SS der ErrorCode 0 übergeben werden oder der auslösende Selektor.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 6, 8, 20	0	8	0	8
	14, 46	Selektor	-	Selektor	./.
#NP	4	Selektor	4	Selektor	./.
#PF	1	fault code	1	fault code	./.
#SS	1, 9	0	-	0	-
	7	Selektor	-	Selektor	./.

Bemerkungen

POP kann einen Wert vom Stack direkt in eine Speicherstelle transferieren. In diesem Falle ist natürlich auch eine indirekte Adressierung möglich. Sollte dazu das ESP-Register als Basisregister verwendet werden, berechnet POP die effektive Adresse, *nachdem* der Wert in ESP durch POP inkrementiert wurde. Dies ist sinnvoll, da ja an SS:ESP vor der Inkrementation der zu POPpende Wert steht, die Adresse also »darunter« (d. h. bei höheren Adressen!) stehen muss. Cave: Dies ist nur mit ESP, nicht aber mit SP möglich, da im ModR/M-Byte als Basisregister nur BX, BP, SI und DI, nicht aber SP in Frage kommen und im SIB-Byte nur die 32-Bit-Register und somit zwar ESP, nicht aber SP angegeben werden können!

Wird ESP als Ziel des POP-Befehls angegeben, wird ESP inkrementiert, *bevor* das Datum vom alten Top of Stack (also dem vor dem Inkrementieren) in ESP geschrieben wird. (Andernfalls würde ja nichts passieren: der TOS würde mit dem Wert im TOS überschrieben!)

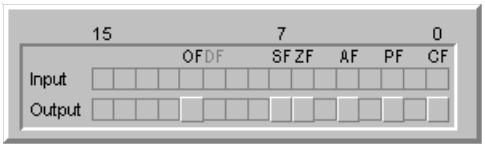
Bei einem POP mit SS als Ziel werden alle Interrupts und NMIs ausgesetzt, bis die auf POP SS folgende Instruktion ausgeführt wurde. Dies erfolgt, um z.B. via `POP SS, MOV ESP, EBP` eine vollständige logische Adresse in die Registerkombination SS:(E)SP zu schreiben, ohne im Falle eines Interrupts Gefahr laufen zu müssen, dass nur der Selektor, nicht aber der Offset des Stacks gültig ist.

PUSH 8086

Kopiere ein Datum auf den Stack (»push stack«). *Funktion*

Operation

```
IF (StackOperandSize = 32) THEN
   IF (OperandSize = 32) THEN
      ESP := ESP - 4
      DWORD[SS:ESP] := Src
   ELSE                ; (OperandSize = 16)
      ESP := ESP - 2
      WORD[SS:ESP] := Src
ELSE                   ; (StackOperandSize = 16)
   IF (OperandSize = 32) THEN
      SP := SP - 4
      DWORD[SS:SP] := Src
   ELSE                ; (OperandSize = 16)
      SP := SP - 2
      WORD[SS:SP] := Src
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Bis auf den Fall PUSH FS und PUSH GS handelt es sich bei allen POP-Befehlen um Ein-Byte-Opcodes, denen nur im Falle einer Speicherbeteiligung ein ModR/M-Byte folgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	FF /6	PUSH m	SS:(E)SP	Mem
	(50+r)	PUSH r	SS:(E)SP	Reg
6A	68	PUSH c	SS:(E)SP	Const
	0E	PUSH CS	SS:(E)SP	SReg CS
	16	PUSH SS	SS:(E)SP	SReg SS
	1E	PUSH DS	SS:(E)SP	SReg DS
	0F A0	PUSH FS	SS:(E)SP	SReg FS
	0F A8	PUSH GS	SS:(E)SP	SReg GS

Die grau unterlegten Operanden sind implizit vorgegeben, müssen aber im Falle der Quelle explizit angegeben werden.

Im Falle des Ein-Byte-Codes mit einem Register als Quelle ist dieses wie folgt kodiert:

Operandengröße	r = 0	1	2	3	4	5	6	7
16 Bit	AX	CX	DX	BX	SP	BP	SI	DI
32 Bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 6, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	1, 2

Bemerkungen PUSH kann einen Wert aus einer Speicherstelle direkt auf den Stack transferieren. In diesem Falle ist natürlich auch eine indirekte Adressierung möglich. Sollte dazu das ESP-Register als Basisregister verwendet werden, berechnet PUSH die effektive Adresse, *bevor* der Wert in ESP durch POP dekrementiert wurde. Dies ist notwendig, da ja die Adresse im »alten« TOS (also vor der Dekrementierung) steht, nicht aber im neuen. Cave: Dies ist nur mit ESP, nicht aber mit SP möglich, da im ModR/M-Byte als Basisregister nur BX, BP, SI und DI, nicht aber SP in Frage kommen und im SIB-Byte nur die 32-Bit-Register und somit zwar ESP, nicht aber SP angegeben werden können!

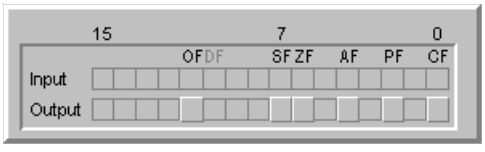
Im real mode wird der Prozessor heruntergefahren, wenn in (E)SP der Wert 1 steht und ein PUSH erfolgen soll, der Platz auf dem Stack aber nicht mehr ausreicht. In diesem Falle wird keine Exception ausgelöst, das Herunterfahren erfolgt ohne Vorwarnung!

POPA **80186**
POPAD **80386**
(POPAW **80386)**

Restauriere den Inhalt aller Allzweckregister vom Stack (»pop all«). *Funktion*

Operation

```
IF (OperandSize = 32) THEN ; ≡ POPAD
   EDI := PopStack
   ESI := PopStack
   EBP := PopStack
   ESP := ESP + 4
   EBX := PopStack
   EDX := PopStack
   ECX := PopStack
   EAX := PopStack
ELSE                       ; ≡ POPA, POPAW
   DI := PopStack
   SI := PopStack
   BP := PopStack
   SP := SP + 2
   BX := PopStack
   DX := PopStack
   CX := PopStack
   AX := PopStack
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

POPA und POPAD haben den identischen Opcode. Die Operanden sind alle impliziert:

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	61	POPA POPAD (POPAW)	Allzweck-register	SS:E(SP)

Die grau unterlegten Operanden sind implizit an- und vorgegeben.

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#PF	1	fault code	1	fault code	./.
#SS	10	0	10	0	10
#AC	1	0	1	0	./:

Bemerkungen Nicht alle Assembler implementieren das Mnemonic POPAW, sondern setzen es mit POPA gleich. In diesem Fall steht POPA für die 16-Bit-Operandenform, POPAD für die 32-Bit-Operandenform und es liegt in der Verantwortung des Programmierers, die korrekten Mnemonics zu verwenden. Sollen Assemblermodule für beide Fälle geschrieben werden, ist dann eine Fallunterscheidung mit Hilfe der bedingten Assemblierung erforderlich.

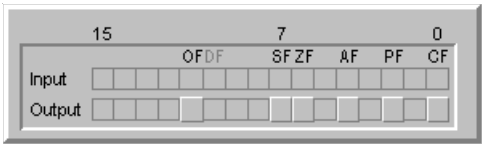
MASM und TASM dagegen sind Assembler, die POPAW implementieren. Es erzwingt die Nutzung von 16-Bit-Operanden auch in 32-Bit-Umgebungen und setzt dort ein operand size override prefix vor den Opcode. Analog erzwingt POPAD die Verwendung von 32-Bit-Operanden und kodiert in 16-Bit-Umgebungen ebenfalls ein operand size override prefix. POPA repräsentiert den Opcode $61 ohne prefix und ist somit das Chamäleon unter den Befehlen: In 16-Bit-Umgebungen werden dadurch 16-Bit-Operanden verwendet, in 32-Bit-Umgebungen 32-Bit-Operanden.

PUSHA 80186
PUSHAD 80386
(PUSHAW 80386)

Funktion Kopiere den Inhalt aller Allzweckregister auf den Stack (»push all«).

Operation
```
IF (OperandSize = 32) THEN ; ≡ PUSHAD
   Temp := ESP
   PushStack(EAX)
   PushStack(ECX)
   PushStack(EDX)
   PushStack(EBX)
```

```
   PushStack(Temp)
   PushStack(EBP)
   PushStack(ESI)
   PushStack(EDI)
ELSE                          ; ≡ PUSHA, PUSHAW
   Temp := SP
   PushStack(AX)
   PushStack(CX)
   PushStack(DX)
   PushStack(BX)
   PushStack(Temp)
   PushStack(BP)
   PushStack(SI)
   PushStack(DI)
```



Input: keiner

Output: Status-Flags werden nicht verändert.

Statusflags

PUSHA und PUSHAD haben den identischen Opcode. Die Operanden sind alle impliziert:

Opcodes

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	60	PUSHA PUSHAD (PUSHAW)	SS:(E)SP	Allzweck-register

Die grau unterlegten Operanden sind implizit an- und vorgegeben.

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception #GP, eine page fault exception (#PF) und eine stack fault exception (#SS).

Exceptions

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	-	0	58	0	58
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-

Bemerkungen Nicht alle Assembler implementieren das Mnemonic PUSHAW, sondern setzen es mit PUSHA gleich. In diesem Fall steht PUSHA für die 16-Bit-Operandenform, PUSHAD für die 32-Bit-Operandenform und es liegt in der Verantwortung des Programmierers, die korrekten Mnemonics zu verwenden. Sollen Assemblermodule für beide Fälle geschrieben werden, ist dann eine Fallunterscheidung mit Hilfe der bedingten Assemblierung erforderlich.

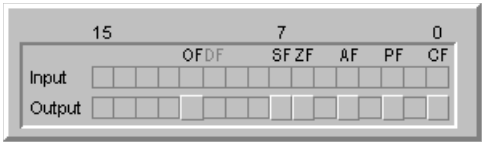
MASM und TASM dagegen sind Assembler, die PUSHAW implementieren. Es erzwingt die Nutzung von 16-Bit-Operanden auch in 32-Bit-Umgebungen und setzt dort ein operand size override prefix vor den Opcode. Analog erzwingt PUSHAD die Verwendung von 32-Bit-Operanden und kodiert in 16-Bit-Umgebungen ebenfalls ein operand size override prefix. PUSHA repräsentiert den Opcode $61 ohne prefix und ist somit das Chamäleon unter den Befehlen: In 16-Bit-Umgebungen werden dadurch 16-Bit-Operanden verwendet, in 32-Bit-Umgebungen 32-Bit-Operanden.

IN 8086
OUT 8086

Funktion Gib ein Datum an ein Port aus oder hole es von dort ab (»input«, »output«).

Operation

```
PortSize := OperandSize
IF (Instruction = IN) THEN
   Port := Src
ELSE                    ; (Instruction = OUT)
   Port := Dest
; Protected Mode mit CPL > IOPL oder Virtual 8086 Mode
IF (PE = 1) AND ((CPL > IOPL) OR (VM = 1)) THEN
   FOR (I = 0) TO (I = PortSize – 1) DO
      IF IOPermissionTable[Port + I] = 1 THEN
         #GP(0)
      ELSE
         IF (Instruction = IN) THEN
            Dest := [Port]
         ELSE           ; (Instruction = OUT)
            [Port] := Src
```



Input: keiner

Output: Status-Flags werden nicht verändert.

Statusflags

Alle Mnemonics werden in Ein-Byte-Opcodes übersetzt. Die zum Einsatz kommenden Operanden sind bis auf die 8-Bit-Konstanten, mit denen die I/O-Port-Nummer direkt angegeben werden kann, implizit vorgegeben.

Opcodes

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
E4	E5	IN acc, c	AL / AX / EAX	Port[Const8]
EC	ED	IN acc, r	AL / AX / EAX	Port[DX]
E6	E7	OUT c, acc	Port[Const8]	AL / AX / EAX
EE	EF	OUT r, acc	Port[DX]	AL / AX / EAX

Die grau unterlegten Operanden sind implizit vorgegeben, müssen aber explizit angegeben werden.

Bei diesen Befehlen kommt als Exceptionquelle nur eine general protection exception #GP in Frage.

Exceptions

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#GP	34	0	48	0	-

2.6 Instruktionen zur Datenkonvertierung

CBW **8086**
CWD **8086**
CWDE **80386**
CDQ **80386**

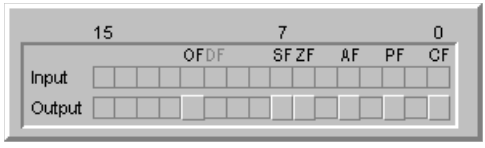
Konvertiere eine ShortInt in eine SmallInt (»convert byte to word«), eine SmallInt in eine LongInt (»convert word to doubleword«, »convert word to doubleword in extended register«) oder eine LongInt in eine QuadInt (»convert doubleword to quadword«).

Funktion

Operation

```
IF (Size(Dest) = 16) THEN
   IF (Size(Src) = 8) THEN   ; ≡ CBW
      AX := SignExtend(AL)
   ELSE                      ; ≡ CWD, (Size(Src) = 16)
      DX: AX := SignExtend(AX)
ELSE                         ; (Size(Dest) = 32)
   IF (Size(Src) = 16)       ; ≡ CWDE
      EAX := SignExtend(AX)
   ELSE                      ; ≡ CDQ; (Size(Src) = 32)
      EDX:EAX := SignExtend(EAX)
```

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert.

Opcodes Bei allen Befehlen handelt es sich um einen Ein-Byte-Opcode. Ist der erste Operand impliziert und stellt den Akkumulator dar, so bleibt es bei diesem Byte. Andernfalls folgt ein ModR/M-Byte, das für die Kodierung des ersten und zweiten Operanden sorgt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	98	CBW	AX	AL
		CWDE	EAX	AX
	99	CWD	DX:AX	AX
		CDQ	EDX:EAX	EAX

Die grau unterlegten Operanden sind implizit an- und vorgegeben.

Exceptions Diese Befehle lösen keine Exceptions aus.

Bemerkungen Bitte beachten Sie, dass CWD und CWDE jeweils eine SmallInt in eine LongInt konvertieren. Der Unterschied liegt in der Repräsentation der Zahl in den Allzweckregistern: Bei CDW wird die LongInt in zwei 16-Bit-Teile aufgeteilt, die in der Registerkombination DX:AX liegen. Bei CWDE wird die resultierende LongInt »im Ganzen« im 32-Bit-Register EAX abgelegt.

Manche Assembler benutzen CBW und CWDE bzw. CDW und CDQ als Synonyme und kodieren mit dem Opcode $98 in 16-Bit-Umgebungen die Konvertierung einer ShortInt in eine SmallInt und in 32-Bit-

Umgebungen einer SmallInt in eine LongInt bzw. mit dem Opcode $99 in 16-Bit-Umgebungen die Konversion einer SmallInt in eine LongInt und in 32-Bit-Umgebungen die einer LongInt in eine QuadInt.

MASM und TASM tun dies nicht! Die Verwendung von CBW erzeugt in 16-Bit-Umgebungen den Opcode $98 ohne Präfix und in 32-Bit-Umgebungen mit operand size override prefix. Dadurch wird sichergestellt, dass jeweils eine ShortInt in eine SmallInt konvertiert wird, unabhängig davon, welche Umgebung herrscht. Analog erzeugt CWDE in 16-Bit-Umgebungen den Opcode $98 mit operand size override prefix, in 32-Bit-Umgebungen ohne Präfix, was die umgebungsunabhängige Konversion einer SmallInt in eine LongInt zur Folge hat. Ebenso wird mit den Mnemonics CWD / CDQ und dem Opcode $99 zur Konversion von SmallInts in LongInts und LongInts in QuadInts verfahren.

2.7 Verzweigungen im Programmablauf: Sprungbefehle

JMP 8086

Führe einen Sprung zu einer neuen Befehlszeile durch (»jump«).

Funktion

Operation

```
IF short/near jump THEN ; Short/Near- oder Intrasegmentsprung
   ; Der NEAR-Pointer hat die Form Offset32, Offset16 bzw.
   ; Offset8 und wird über den Operanden Dest entweder direkt
   ; als 8-, 16- bzw. 32-Bit-Konstante dem Opcode folgend oder
   ; indirekt über ein Word/DoubleWord in einem Register oder
   ; einer Speicherstelle angegeben
   IF relative THEN    ; relative jumps
      Offset := EIP + Dest
   ELSE                ; absolute jumps
      Offset := Dest
   IF (Offset > CodeSegmentLimit) THEN
      #GP(0)
   ELSE
      IF (OperandSize = 32) THEN
         EIP := Offset
      ELSE             ; (OperandSize = 16)
         EIP := ZeroExtend(Offset)

ELSE                   ; Far- oder Intersegment-Sprung
   ; Der FAR-Pointer hat die Form Selector16:Offset32 bzw.
   ; Selector16:Offset16 und wird über den Operanden Dest
```

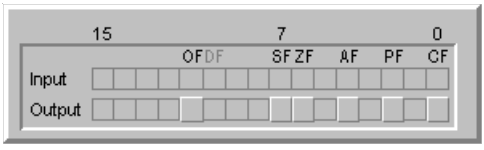
```
; entweder als 48- bzw. 32-Bit-Konstante dem Opcode folgend
; direkt angegeben oder indirekt über eine 48- bzw. 32-Bit-
; Speicherstelle.
IF (OperandSize = 32) THEN
   Selector := Dest[47..32]
   Offset := Dest[31..00]
ELSE                ; (OperandSize = 16)
   Selector := Dest[31..16]
   Offset := Dest[15..00]

; real mode oder virtual 8086 mode
IF (PE = 0 OR (PE = 1) AND (VM = 1)) THEN
   IF (Offset > NewCodeSegmentLimit) THEN
      #GP(0)
   ELSE
      CS := Selector
      IF (OperandSize = 32) THEN
         EIP := Offset
      ELSE
         EIP := ZeroExtend(Offset)

; protected mode
IF (PE = 1) AND (VM = 0) THEN
   IF (Selector = Nullselektor)
   OR ((Indirekte Adressierung) AND
      (Aktueller Selektor illegal)) THEN
      #GP(0)
   IF (Selector > DescriptorTableLimit) THEN
      #GP(Selector)
   Descriptor := DescriptorTable[Selector]
   NewType := Descriptor[TypeFiled]
   NewDPL := Descriptor[DPLField]
   IF (NewType ≠ Conforming CodeSegment)
   OR (NewType ≠ NonConforming CodeSegment)
   OR (NewType ≠ Call Gate)
   OR (NewType ≠ Task Gate)
   OR (NewType ≠ TSS) THEN
      #GP(Selector)

   ; Ziel: conforming code segment
   IF (NewType = Conforming CodeSegment) THEN
      IF (NewDPL > CurrentCPL) THEN
         #GP(Selector)
```

```
      IF (Segment nicht »present« markiert) THEN
         #NP(Selektor)
      IF (Offset > NewCodeSegmentLimit) THEN
         #GP(0)
      CS[visible] := Selector
      CS[hidden] := Descriptor
      CS[RPL] := CPL
      EIP := Offset

   ; Ziel: non-conforming code segment
   IF (NewType = Non-Conforming CodeSegment) THEN
      IF (SelectorRPL > CurrentCPL)
      OR (NewDPL ≠ CurrentCPL) THEN
         #GP(Selector)
      IF (Segment nicht »present« markiert) THEN
         #NP(Selektor)
      IF (Offset > NewCodeSegmentLimit) THEN
         #GP(0)
      CS[visible] := Selector
      CS[hidden] := Descriptor
      CS[RPL] := CPL
      EIP := Offset

   ; Ziel: call gate. Das bedeutet, dass Selector noch auf
   ; den Deskriptor für das Call Gate zeigt. Dieses ent-
   ; hält jedoch einen Selektor und einen Offset in das
   ; dazugehörige Codesegment. Daher werden Selector und
   ; Offset neu definiert!
   IF (NewType = Call Gate) THEN
      IF (NewDPL < CurrentCPL)
      OR (NewDPL < SelectorRPL) THEN
         #GP(Selector)
      IF (Gall Gate nicht »present« markiert) THEN
         #NP(Selector)
      ; ACHTUNG: Hier werden Selector und Offset neu
      ; definiert! Der mit dem JMP-Befehl übergebene
      ; Offset ist somit ein Dummy-Offset!
      Selector := Descriptor[SelectorField]
      Offset := Descriptor[OffsetField]
      ;  nun aktueller Descriptor = CodeSegmentDescriptor
      ; Offset aus Gall Gate, segment aus
      Descriptor := DescriptorTable[Selector]
      NewType := Descriptor[TypeFiled]
      NewDPL := Descriptor[DPLField]
```

```
   IF (Selector = NullSelector) THEN
      #GP(0)
   IF (Selector > DescriptorTableLimit) THEN
      #GP(Selector)
   IF (NewType ≠ CodeSegment)
   OR ((NewType = Conforming Codesegment) AND
      (NewDPL > CurrentCPL))
   OR ((SegmentType = NonConforming CodeSegment) AND
      (NewDPL ≠ CurrentCPL)) THEN
      #GP(Selector)
   IF (Segment nicht »present« markiert) THEN
      #NP(Selector)
   IF (Offset > NewCodeSegmentLimit) THEN
      #GP(0)
   CS[visible] := Selector
   CS[hidden] := Descriptor
   CS[RPL] := CPL
   EIP := Offset

; Ziel: task gate. Das bedeutet, dass Selector noch auf
; den Deskriptor für das Task Gate zeigt. Dieses ent-
; hält einen Selektor für ein Task State Segment. Und
; in diesem wiederum sind Selector und Offset (die
; Felder CS und EIP) der Instruktion verzeichnet, die
; nach dem zu erfolgenden task switch als Nächstes aus-
; geführt wird. Der mit dem JMP-Befehl übergebene Offset
; ist somit ein Dummy-Offset, benötigt wurde nur der
; Selector des FAR-Pointers als Selector für das Task
; Gate
IF (NewType = Task Gate) THEN
   IF (NewDPL < CurrentCPL)
   OR (NewDPL < SelectorRPL) THEN
      #GP(Selector)
   IF (Task Gate nicht »present« markiert) THEN
      #NP(Selector)
   ; ACHTUNG: Hier wird der Selector neu definiert
   ; Er zeigt dann auf einen TSS-Deskriptoren
   Selector := Descriptor[SelectorField]
   Descriptor := DescriptorTable[Selector]
   Global := Descriptor[G-Field]
   Busy := Descriptor[B-Field]
   IF (NOT Global)
   OR Busy
   OR (Selector > DescriptorTableLimit) THEN
      #GP(Selector)
```

```
      IF (TSS nicht »present« markiert) THEN
         #NP(Selector)
      TaskSwitch(Selector)
      IF (EIP > SegmentLimit) THEN
         #GP(0)

   ; Ziel: task state segment. Das bedeutet, dass Selector
   ; auf den Deskriptor für das TSS zeigt. Hier sind
   ; Selector und Offset (die Felder CS und EIP) der
   ; Instruktion verzeichnet, die nach dem zu erfolgenden
   ; task switch als Nächstes ausgeführt wird. Der mit dem
   ; JMP-Befehl übergebene Offset ist somit ein Dummy-
   ; Offset, benötigt wurde nur der Selector des FAR-
   ; Pointers als Selector für das TSS
   IF (NewType = TSS) THEN
      IF (NewDPL < CurrentCPL)
      OR (NewDPL < SelectorRPL) THEN
         #GP(Selector)
      IF (TSS nicht »present« markiert) THEN
         #NP(Selector)
      TaskSwitch(Selector)
      IF (EIP > SegmentLimit) THEN
         #GP(0)
```

(Ziemlich kompliziert, dieser JMP-Befehl, oder?)



Statusflags

Input: keiner
Output: Statusflags werden nicht verändert, es sei denn, es erfolgt ein task switch. Dann sind alle Flags betroffen.

Opcodes

Relative Sprünge sind nur innerhalb des Segments möglich (short oder near jumps). In diesem Fall folgt dem Ein-Byte-Opcode eine vorzeichenbehaftete Konstante, die zum aktuellen EIP-Inhalt addiert wird. Alternativ kann ein absoluter Intrasegmentsprung durchgeführt werden, indem das Sprungziel als Offset zum Segmentbeginn in einem Register oder einer Speicherstelle übergeben wird. Intersegmentsprünge sind grundsätzlich absolute far jumps, bei denen ein Zeiger bestehend aus 16-Bit-Selektor und 16-/32-Bit-Offset übergeben wird. Dies kann durch Angabe einer 32-/48-Bit-Konstanten erfolgen, was einem direkten Sprung entspricht, oder indirekt über eine 32-/48-Bit-Speicherstelle.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Src	Bemerkung
EB	E9	JMP ps/n	Const	short/near; direct; relative
	FF /4	JMP r/m	Reg / Mem	near; indirect; absolute
	EA	JMP pf	Const	far; direct, absolute
	FF /5	JMP m	Mem	far; indirect; absolute

Exceptions Die bei diesem Befehl möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine segment not present exception #NP, eine page fault exception (#PF) und eine stack fault exception (#SS).

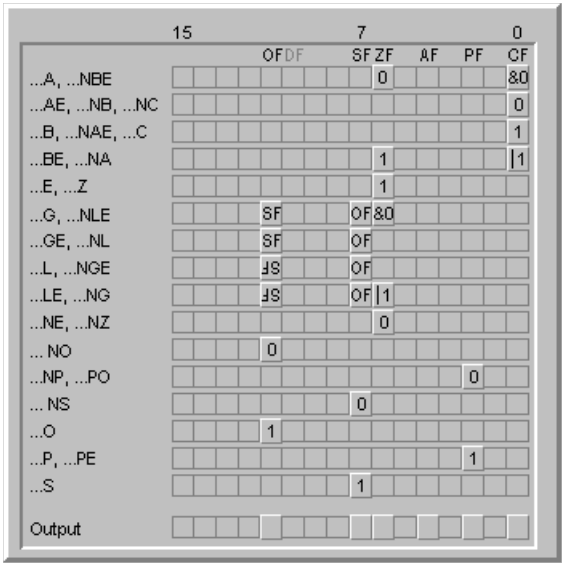
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 4, 8, 10 15, 23, 24, 26, 35, 36, 40, 43, 47, 64	0 selector	8, 10 -	0 -	8 ./.
#NP	1	selector	-	-	./.
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

Jcc 8086

Funktion Verzweige je nach Erfüllung der Bedingung zu einer anderen Adresse im Codesegment (»jump on condition«).

Operation

```
; Bedingte Sprünge sind IMMER short jumps, sodass das Sprung-
; ziel relativ als vorzeichenbehaftete Distanz von der aktu-
; ellen Position aus angegeben wird.
IF Condition THEN      ; wenn die Bedingung erfüllt ist
   EIP := EIP + SignExtend(Dest)
   IF (OperandSize = 16) THEN
      EIP := EIP AND $0000FFFF
```



Statusflags

Input: Je nach Sprungbefehl werden einzelne Flags oder Flag-Kombinationen ausgewertet. »&« steht dafür, dass beide Bedingungen zusammen erfüllt sein müssen, »|« dafür, dass eine Bedingung erfüllt sein muss. Auf dem Kopf stehende Beschriftungen zeigen umgekehrte Flag-Stellung.

Output: Die Flags bleiben unverändert.

Opcodes

ACHTUNG: Die bedingten Sprünge mit einem near pointer als Operanden gibt es erst ab dem 80386! Bedingte Sprünge sind immer relative Sprünge, das heißt, der Operand wird als vorzeichenbehaftete Konstante aufgefasst und zum Inhalt des EIP-Registers addiert.

Opcode			Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**	**Mnemonic**	**Src**	**Bemerkung**
77	0F 87	JA ps/n	Const	short/near; direct; relative
		JNBE ps/n	Const	short/near; direct; relative
73	0F 83	JAE ps/n	Const	short/near; direct; relative
		JNB ps/n	Const	short/near; direct; relative
		JNC ps/n	Const	short/near; direct; relative
72	0F 82	JB ps/n	Const	short/near; direct; relative
		JNAE ps/n	Const	short/near; direct; relative
		JC ps/n	Const	short/near; direct; relative
76	0F 86	JBE ps/n	Const	short/near; direct; relative
		JNA ps/n	Const	short/near; direct; relative
E3		J(E)CXZ ps	Const8	short; direct; relative

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Src	Bemerkung
74	0F 84	JE ps/n	Const	short/near; direct; relative
		JZ ps/n	Const	short/near; direct; relative
7F	0F 8F	JG ps/n	Const	short/near; direct; relative
		JNLE ps/n	Const	short/near; direct; relative
7D	0F 8D	JGE ps/n	Const	short/near; direct; relative
		JNL ps/n	Const	short/near; direct; relative
7C	0F 8C	JL ps/n	Const	short/near; direct; relative
		JNGE ps/n	Const	short/near; direct; relative
7E	0F 8E	JLE ps/n	Const	short/near; direct; relative
		JNG ps/n	Const	short/near; direct; relative
75	0F 85	JNE ps/n	Const	short/near; direct; relative
		JNZ ps/n	Const	short/near; direct; relative
71	0F 81	JNO ps/n	Const	short/near; direct; relative
7B	0F 8B	JNP ps/n	Const	short/near; direct; relative
		JPO ps/n	Const	short/near; direct; relative
79	0F 89	JNS ps/n	Const	short/near; direct; relative
70	0F 80	JO ps/n	Const	short/near; direct; relative
7A	0F 8A	JP ps/n	Const	short/near; direct; relative
		JPE ps/n	Const	short/near; direct; relative
78	0F 88	JS ps/n	Const	short/near; direct; relative

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind auf eine general protection exception (#GP) beschränkt.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	10	0	11	0	11

CALL 8086

Rufe eine Routine auf (»call«). *Funktion*

Operation

```
IF near call THEN      ; Near- oder Intrasegment-Call
   ; Der NEAR-Pointer hat die Form Offset32 und wird über den
   ; Operanden Dest entweder direkt als 32-Bit-Konstante dem
   ; Opcode folgend oder indirekt über ein DoubleWord in einem
   ; Register oder einer Speicherstelle angegeben
   IF relative THEN
      IF (OperandSize = 32) THEN
         IF (FreeStackSpace < 4) THEN
            #SS(0)
         PushStack(EIP)
         EIP := EIP + Dest
      ELSE             ; (OperandSize = 16)
         IF (FreeStackSpace < 2) THEN
            #SS(0)
         PushStack(IP)
         EIP := (EIP AND $0000FFFF) + Dest
   ELSE                ; absolute calls
      IF (Dest > SegmentLimit) THEN
         #GP(0)
      IF (OperandSize = 32) THEN
         IF (FreeStackSpace < 4) THEN
            #SS(0)
         PushStack(EIP)
         EIP := Dest
      ELSE             ; (OperandSize = 16)
         IF (FreeStackSpace < 2) THEN
            #SS(0)
         PushStack(IP)
         EIP := ZeroExtend(Dest)

ELSE                   ; Far- oder Intersegment-Call
   ; Der FAR-Pointer hat die Form Selector16:Offset32 bzw.
   ; Selector16:Offset16 und wird über den Operanden Dest
   ; entweder als 48- bzw. 32-Bit-Konstante dem Opcode folgend
   ; direkt angegeben oder indirekt über eine 48- bzw. 32-Bit-
   ; Speicherstelle.
   IF (OperandSize = 32) THEN
      Selector := Dest[47..32]
      Offset := Dest[31..00]
   ELSE                ; (OperandSize = 16)
      Selector := Dest[31..16]
      Offset := Dest[15..00]
```

```
; real mode oder virtual 8086 mode
IF (PE = 0 OR (PE = 1) AND (VM = 1)) THEN
   IF (OperandSize = 32) THEN
      IF (FreeStackSpace < 6) THEN
         #SS(0)
      IF (Offset > NewCodeSegmentLimit) THEN
         #GP(0)
      PushStack(CS)
      PushStack(EIP)
      CS := Selector
      EIP := Offset
   ELSE             ; (OperandSize = 16)
      IF (FreeStackSpace < 4) THEN
         #SS(0)
      IF (Offset > NewCodeSegmentLimit) THEN
         #GP(0)
      PushStack(CS)
      PushStack(IP)
      CS := Selector
      EIP := ZeroExtend(Offset)

; protected mode
IF (PE = 1) AND (VM = 0) THEN
   IF (Selector = Nullselektor)
   OR ((Indirekte Adressierung) AND
      (Aktueller Selektor illegal)) THEN
      #GP(0)
   IF (Selector > DescriptorTableLimit) THEN
      #GP(Selector)
   Descriptor := DescriptorTable[Selector]
   NewType := Descriptor[TypeFiled]
   NewDPL := Descriptor[DPLField]
   IF (NewType ≠ Conforming CodeSegment)
   OR (NewType ≠ NonConforming CodeSegment)
   OR (NewType ≠ Call Gate)
   OR (NewType ≠ Task Gate)
   OR (NewType ≠ TSS) THEN
      #GP(Selector)

   ; Ziel: conforming code segment
   IF (NewType = Conforming CodeSegment) THEN
      ; aktueller Descriptor = CodeSegmentDescriptor
      IF (NewDPL > CurrentCPL) THEN
         #GP(Selector)
```

```
      IF (Segment nicht »present« markiert) THEN
         #NP(Selektor)
      IF (Offset > NewCodeSegmentLimit) THEN
         #GP(0)
      IF (OperandSize = 32) THEN
         IF (FreeStackSpace < 6) THEN
            #SS(0)
         PushStack(CS)
         PushStack(EIP)
         CS[visible] := Selector
         CS[hidden] := Descriptor[Selector]
         CS[RPL] := CPL
         EIP := Offset
      ELSE          ; (OperandSize = 16)
         IF (FreeStackSpace < 4) THEN
            #SS(0)
         PushStack(CS)
         PushStack(IP)
         CS[visible] := Selector
         CS[hidden] := Descriptor[Selector]
         CS[RPL] := CPL
         EIP := ZeroExtend(Offset)

   ; Ziel: non-conforming code segment
   IF (NewType = Non-Conforming CodeSegment) THEN
      ; aktueller Descriptor = CodeSegmentDescriptor
      IF (SelectorRPL > CurrentCPL)
      OR (NewDPL ≠ CurrentCPL) THEN
         #GP(Selector)
      IF (Segment nicht »present« markiert) THEN
         #NP(Selektor)
      IF (Offset > NewCodeSegmentLimit) THEN
         #GP(0)
      IF (OperandSize = 32) THEN
         IF (FreeStackSpace < 6) THEN
            #SS(0)
         PushStack(CS)
         PushStack(EIP)
         CS[visible] := Selector
         CS[hidden] := Descriptor[Selector]
         CS[RPL] := CPL
         EIP := Offset
      ELSE          ; (OperandSize = 16)
         IF (FreeStackSpace < 4) THEN
            #SS(0)
```

```
      PushStack(CS)
      PushStack(IP)
      CS[visible] := Selector
      CS[hidden] := Descriptor[Selector]
      CS[RPL] := CPL
      EIP := ZeroExtend(Offset)

; Ziel: call gate. Das bedeutet, dass Selector noch auf
; den Deskriptor für das Call Gate zeigt. Dieses ent-
; hält jedoch einen Selektor und einen Offset in das
; dazugehörige Codesegment. Daher werden Selector und
; Offset neu definiert!
IF (NewType = Call Gate) THEN
   ; aktueller Descriptor = CallGateDescriptor
   IF (NewDPL < CurrentCPL)
   OR (NewDPL < SelectorRPL) THEN
      #GP(Selector)
   IF (Gall Gate nicht »present« markiert) THEN
      #NP(Selector)
   IF (Selector = NullSelector) THEN
      #GP(0)
   IF (Selector > DescriptorTableLimit) THEN
      #GP(Selector)
   ; Hier werden der Selektor und der Descriptor
   ; des Codesegments ausgelesen, auf das das
   ; Call Gate verweist
   Selector := Descriptor[SelectorField]
   Offset := Descriptor[OffsetField]
   Count := Descriptor[ParamCountField]
   ;  nun aktueller Descriptor = CodeSegmentDescriptor
   ; Offset aus Gall Gate, segment aus
   Descriptor := DescriptorTable[Selector]
   NewType := Descriptor[TypeFiled]
   NewDPL := Descriptor[DPLField]
   IF (NewType ≠ CodeSegment)
   OR (NewDPL > CurrentCPL) THEN
      #GP(selector)
   IF (Segment nicht »present« markiert) THEN
      #NP(Selector)
   IF ((NewType = NonConforming Codesegment) AND
      (NewDPL < CurrentCPL)) THEN

      ; CALL einer höher privilegierten Routine
      ; -> Stack der entsprechenden Privilegstufe
      ; verwenden
```

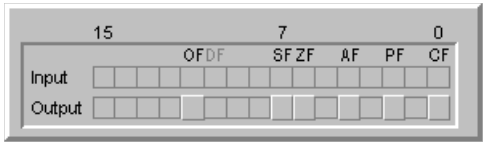
```
IF (aktueller TSS = 32-Bit-TSS) THEN
   ; Suche im TSS nach dem zu verwendenden Stack
   ; Privilegstufe 0 (DPL = 0) -> TSS-Offset  4
   ; Privilegstufe 1 (DPL = 1) -> TSS-Offset 12
   ; Privilegstufe 2 (DPL = 2) -> TSS-Offset 20
   FirstByte := (NewDPL * 8) + 4
   NewSS := WORD(TSS[FirstByte + 4])
   NewESP := DWORD(TSS[FirstByte])
ELSE       ; (aktueller TSS = 16-Bit-TSS
   ; Suche im TSS nach dem zu verwendenden Stack
   ; Privilegstufe 0 (DPL = 0) -> TSS-Offset  2
   ; Privilegstufe 1 (DPL = 1) -> TSS-Offset 6
   ; Privilegstufe 2 (DPL = 2) -> TSS-Offset 10
   FirstByte := (NewDPL * 4) + 2
   NewSS := WORD(TSS[FirstByte + 2])
   NewESP := ZeroExtend(WORD(TSS[FirstByte]))
IF (NewSS = Nullselektor) THEN
   #TS(NewSS)
IF (NewSS > DescriptorTableLimit) THEN
   #TS(NewSS)
; Lesen des Deskriptors, auf den NewSS zeigt
;  -> Neues Stacksegment
SSDescriptor := DescriptorTable[Selector]
SSRPL := NewSS[RPLField]
SSDPL := SSDescriptor[DPLField]
SSType := SSDescriptor[TypeField]
IF (SSRPL ≠ NewDPL)
OR (SSDPL ≠ NewDPL)
OR (SSType ≠ writeable data segment) THEN
   #TS(NewSS)
IF (NewSS nicht »present« markiert) THEN
   #SS(NewSS)
IF (Offset > NewCodeSegmentLimit) THEN
   #GP(0)
IF (CallGateSize = 32) THEN
   IF (FreeStackSpace < 16 + Parameter) THEN
      #SS(NewSS)
ELSE       ; (CallGateSize = 16)
   IF (FreeStackSpace < 8 + Parameter) THEN
      #SS(NewSS)
OldSS := SS
OldESP := ESP
SS[visible] := NewSS
SS[hidden] := SSDescriptor
ESP := NewESP
```

```
            CS[visible] := Selector
            CS[hidden] := Descriptor
            EIP := Offset
            PushStack(OldSS)
            PushStack(OldESP)
            ; Parameter vom alten auf den neuen Stack
            ; kopieren
            FOR (I = 0) TO (I = Count – 1) DO
                PushStack(GetOldStack(I))
            PushStack(OldCS)
            PushStack(OldEIP)
         ELSE

            ; CALL einer gleich privilegierten Routine
            IF (CallGateSize = 32) THEN
               IF (FreeStackSpace < 8) THEN
                  #SS(NewSS)
               IF (EIP > NewCodeSegmentLimit) THEN
                  #GP(0)
            ELSE       ; (CallGateSize = 16)
               IF (FreeStackSpace < 4) THEN
                  #SS(NewSS)
               IF (IP > NewCodeSegmentLimit) THEN
                  #GP(0)
            CS[visible] := Selector
            CS[hidden] := Descriptor
            EIP := Offset
            PushStack(OldCS)
            PushStack(OldEIP)

      ; Ziel: task gate. Das bedeutet, dass Selector noch auf
      ; den Deskriptor für das Task Gate zeigt. Dieser ent-
      ; hält einen Selektor für ein Task State Segment. Und
      ; in diesem wiederum sind Selector und Offset (die
      ; Felder CS und EIP) der Instruktion verzeichnet, die
      ; nach dem zu erfolgenden task switch als Nächstes aus-
      ; geführt wird. Der mit dem JMP-Befehl übergebene Offset
      ; ist somit ein Dummy-Offset, benötigt wurde nur der
      ; Selector des FAR-Pointers als Selector für das Task
      ; Gate
      IF (NewType = Task Gate) THEN
         IF (NewDPL < CurrentCPL)
         OR (NewDPL < SelectorRPL) THEN
            #GP(Selector)
         IF (Task Gate nicht »present« markiert) THEN
            #NP(Selector)
```

```
      ; ACHTUNG: Hier wird der Selector neu definiert
      ; Er zeigt dann auf einen TSS-Deskriptor
      Selector := Descriptor[SelectorField]
      Descriptor := DescriptorTable[Selector]
      Global := Descriptor[G-Field]
      Busy := Descriptor[B-Field]
      IF (NOT Global)
      OR Busy
      OR (Selector > DescriptorTableLimit) THEN
         #GP(Selector)
      IF (TSS nicht »present« markiert) THEN
         #NP(Selector)
      ; Der NestedTaskSwitch bei CALL unterscheidet sich
      ; vom TaskSwitch bei JMP dadurch, dass in das PTL-
      ; Feld (previous task link) des TSS der Selector des
      ; TSS des aktuellen tasks eingetragen wird. Dies er-
      ; möglicht den Rücksprung in den rufenden task.
      NestedTaskSwitch(Selector)
      IF (EIP > SegmentLimit) THEN
         #GP(0)

   ; Ziel: task state segment. Das bedeutet, dass Selector
   ; auf den Deskriptor für das TSS zeigt. Hier sind
   ; Selector und Offset (die Felder CS und EIP) der
   ; Instruktion verzeichnet, die nach dem zu erfolgenden
   ; task switch als Nächstes ausgeführt wird. Der mit dem
   ; JMP-Befehl übergebene Offset ist somit ein Dummy-
   ; Offset, benötigt wurde nur der Selector des FAR-
   ; Pointers als Selector für das TSS
   IF (NewType = TSS) THEN
      IF (NewDPL < CurrentCPL)
      OR (NewDPL < SelectorRPL) THEN
         #GP(Selector)
      IF (TSS nicht »present« markiert) THEN
         #NP(Selector)
      ; Der NestedTaskSwitch bei CALL unterscheidet sich
      ; vom TaskSwitch bei JMP dadurch, dass in das PTL-
      ; Feld (previous task link) des TSS der Selector des
      ; TSS des aktuellen tasks eingetragen wird. Dies er-
      ; möglicht den Rücksprung in den rufenden task.
      NestedTaskSwitch(Selector)
      IF (EIP > SegmentLimit) THEN
         #GP(0)
```

(Noch viel komplizierter als der JMP-Befehl!)

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert, es sei denn, es erfolgt ein task switch. Dann sind alle Flags betroffen.

Opcodes Relative Calls sind nur innerhalb des Segments möglich (near calls). In diesem Fall folgt dem Ein-Byte-Opcode eine vorzeichenbehaftete Konstante, die zum aktuellen EIP-Inhalt addiert wird. Alternativ kann ein absoluter Intrasegment-CALL durchgeführt werden, indem das Sprungziel als Offset zum Segmentbeginn in einem Register oder einer Speicherstelle übergeben wird. Intersegment-Calls sind grundsätzlich absolute far calls, bei denen ein Zeiger bestehend aus 16-Bit-Selektor und 16-/32-Bit-Offset übergeben wird. Dies kann durch Angabe einer 32-/48-Bit-Konstanten erfolgen, was einem direkten Sprung entspricht, oder indirekt über eine 32-/48-Bit-Speicherstelle.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Src	Bemerkung
	E8	CALL pn	Const	near; direct; relative
	FF /2	CALL r/m	Reg / Mem	near; indirect; absolute
	9A	CALL pf	Const	far; direct, absolute
	FF /3	CALL m	Mem	far; indirect; absolute

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine segment not present exception #NP, eine page fault exception (#PF) , eine stack fault exception (#SS) sowie eine invalid TSS exception (#TS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 4, 5, 8, 10 12, 23, 24, 35, 36, 38, 40, 47, 64	0 selector	8, 10 -	0 -	8, 10 ./.
#NP	1	selector	-	-	./.
#PF	1	fault code	1	fault code	./.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#SS	1, 5	0	-	0	-
	5, 6, 8	selector	-	selector	-
#TS	1, 2, 3	selector	-	selector	./.

RET — 8086

Kehre aus einer Routine zurück (»return«).

Funktion

Operation

```
IF near return THEN      ; Rückkehr von Near Call
   IF (OperandSize = 32) THEN
      IF ((ESP – 12) > StackSegmentLimit) THEN
         #SS(0)
      EIP := PopStack
   ELSE             ; (OperandSize = 16)
      IF ((ESP – 6) > StackSegmentLimit)) THEN
         #SS(0)
      EIP := (PopStack AND $0000FFFF)
      IF (EIP > CodeSegmentLimit) THEN
         #GP(0)
   IF (Instruction has Operand) THEN
      IF (StackOperandSize = 32) THEN
         ESP := ESP + Src
      ELSE           ; (StackOperandSize = 16)
         SP := SP + Src

ELSE                  ; Far Return

   ; real mode oder virtual 8086 mode
   IF (PE = 0 OR (PE = 1) AND (VM = 1)) THEN
      IF (OperandSize = 32) THEN
         IF ((ESP – 12) > StackSegmentLimit) THEN
            #SS(0)
         EIP := PopStack
      ELSE             ; (OperandSize = 16)
         IF ((ESP – 6) > StackSegmentLimit)) THEN
            #SS(0)
         EIP := (PopStack AND $0000FFFF)
         IF (EIP > CodeSegmentLimit) THEN
            #GP(0)

      IF (Instruction has Operand) THEN
```

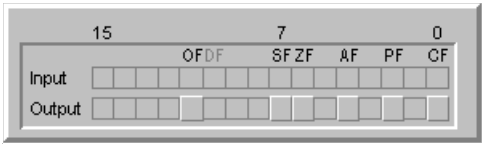
```
      IF (StackOperandSize = 32) THEN
         ESP := ESP + Src
      ELSE           ; (StackOperandSize = 16)
         SP := SP + Src

; protected mode
IF (PE = 1) AND (VM = 0) THEN
   IF (OperandSize = 32) THEN
      ; an [ESP + 4] = zweites DWord liegt CS!
      IF ((ESP + 8) > StackSegmentLimit) THEN
         #SS(0)
      Selector := [ESP + 2]
      Offset := [ESP]
   ELSE             ; (OperandSize = 16)
      ; an SP + 4 = zweites Word liegt CS!
      IF ((SP + 4) > StackSegmentLimit)) THEN
         #SS(0)
      Selector := [SP + 2]
      Offset := [SP]
   IF (Selector = Nullselektor) THEN
      #GP(0)
   IF (Selector > DescriptorTableLimit) THEN
      #GP(Selector)
   Descriptor := DescriptorTable[Selector]
   NewType := Descriptor[TypeField]
   IF (NewType ≠ CodeSegment) THEN
      #GP(0)
   IF (Selector[RPL] < CurrentCPL) THEN
      #GP(Selector)
   IF (NewType = Conforming Codesegment)
   AND (NewDPL > Selector[RPL]) THEN
      #GP(Selector)
   IF (Codesegment  nicht »present« markiert) THEN
      #NP(Selector)

   IF (Selector[RPL] > CPL) THEN
      ; Rückkehr aus höherprivilegiertem Segment. Dies
      ; kann nur über ein Call Gate angesprungen worden
      ; sein. Der Stack hat dann den Aufbau:
      ;
      ; ESP +   Inhalt       Bemerkungen
      ;
      ; 16+Src  RetSS        Rückkehr-StackSegment
      ; 12+Src  RetESP       Rückkehr-Stackpointer
      ; 08+Src  Param(1)     erster von (Src/4) Parametern,
      ;  :        :          die vom »alten« auf den aktu-
```

```
; 08      Param(Src/4) ellen Stack kopiert wurden.
; 04      RetCS        Rückkehr-Codesegment
; 00      RetEIP       Rückkehr-EIP
;
; Das macht insgesamt (16 + Src Bytes), da auch die
; 16-Bit-Selektoren als DWORD gePUSHt wurden.
; Analoge Betrachtungen gelten für 16-Bit-Umgebung
IF (OperandSize = 32) THEN
  ; Liegen die (16+Src) Bytes außerhalb des Segments?
  IF (ESP – (16 + Src) > StackSegmentLimit) THEN
     #SS(0)
   NewSS := [ESP + 16 + Src]
   NewESP := [ESP + 12 + Src]
ELSE          ; (OperandSize = 16)
  ; Liegen die (8+Src) Bytes außerhalb des Segments?
  IF (ESP – (8 + Src) > StackSegmentLimit) THEN
     #SS(0)
   NewSS := [ESP + 8 + Src]
   NewESP := [ESP + 6 + Src]
IF (NewSS = Nulldescriptor) THEN
   #GP(0)
IF (NewSS > DescriptorTableLimit) THEN
   #GP(Selector)
SSDescriptor := DescriptorTable[NewSS]
IF (NewSS[RPL] ≠ Selector[RPL])
OR (SSDescriptor[Type] ≠ writeable data segment)
OR (SSDescriptor[DPL] ≠ Selector[RPL]) THEN
   #GP(Selector)
IF (SSDescriptor[P] nicht »present« markiert) THEN
   #SS(NewSS)
IF (Offset > Descriptor[Limit]) THEN
   #GP(0)
CPL := Selector[RPL]
IF (OperandSize = 32) THEN
   EIP := PopStack
   CS[visible] := PopStack
   CS[hidden] := Descriptor
   CS[RPL] := CPL
   ESP := ESP + Src   ; Parameter vom Stack!
   ESP := PopStack    ; 32-Bit-POP
   NewSS := PopStack  ; 32-Bit-POP
   SS := LowWord(NewSS)
ELSE          ; (OperandSize = 16)
   EIP := PopStack AND $0000FFFF
   CS[visible] := PopStack
```

```
            CS[hidden] := Descriptor
            CS[RPL] := CPL
            ESP := ESP + Src   ; Parameter vom Stack!
            NewSP := PopStack  ; 16-Bit-POP
            ESP := ZeroExtend(NewSP)
            SS := PopStack     ; 16-Bit-POP
         ; Die restlichen Segmentregister prüfen!
         FOR (TestSelector = [DS], [ES], [FS], [GS]) DO
            TestDescriptor := DescriptorTable[TestSelector]
            TestType := TestDescriptor[Type]
            TestDPL := TestDescriptor[DPL]
            IF ((TestType = data segment)
                OR (TestType = nonconforming code segment))
            AND (CPL > TestDPL) THEN
               TestSelector := Nullselektor
            IF (TestSelector > DescriptorTableLimit)
            OR ((TestType ≠ data segment)
               AND (TestType ≠ readable code segment))
            OR (((TestType = data segment)
                  OR (TestType = nonconforming code segment))
               AND ((TestDPL < CPL)
                  OR (TestDPL < Selector[RPL]))) THEN
               TestSelector := Nullselektor
         ; Parameter auch vom Rückkehr-Stack entfernen
         IF (Instruction has Operand) THEN
            IF (StackOperandSize = 32) THEN
               ESP := ESP + Src
            ELSE           ; (StackOperandSize = 16)
               SP := SP + Src

      ELSE             ; (Selector[RPL] <= CPL)
         ; Rückkehr aus gleichprivilegiertem Segment
         IF (Selector > DescriptorTableLimit) THEN
            #GP(0)
         IF (OperandSize = 32) THEN
            EIP := PopStack
            CS[visible] := PopStack
            CS[hidden] := Descriptor
            ESP := PopStack    ; 32-Bit-POP
         ELSE          ; (OperandSize = 16)
            EIP := PopStack AND $0000FFFF
            CS[visible] := PopStack
            CS[hidden] := Descriptor
            ESP := ZeroExtend(NewSP)
```

```
; Parameter auch vom Rückkehr-Stack entfernen
IF (Instruction has Operand) THEN
   IF (StackOperandSize = 32) THEN
      ESP := ESP + Src
   ELSE           ; (StackOperandSize = 16)
      SP := SP + Src
```

(Verglichen mit JMP und CALL: Auch nicht schlecht!)



Statusflags

Input: keiner
Output: Status-Flags werden nicht verändert, es sei denn, es erfolgt ein task switch. Dann sind alle Flags betroffen.

Opcodes

RET kommt in zweimal zwei Versionen vor: Als near return und somit als Gegenstück zu einem near call mit oder ohne Operand und als far return (RETF) und somit als Gegenstück zu einem far call ebenfalls mit und ohne Operanden.

Opcode		Mnemonic	Operanden	
keine Operanden	16-Bit-Operanden		Src	Bemerkung
C3	C2	RET	(Const16)	near
CB	CA	RET	(Const16)	far

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine segment not present exception #NP, eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	7, 10 13, 14, 27, 28, 29, 39, 44, 45, 66, 67	0 selector	10 -	0 -	10 ./.
#NP	5	selector	-	-	./.
#PF	1	fault code	1	fault code	./.
#SS	9	0	-	0	

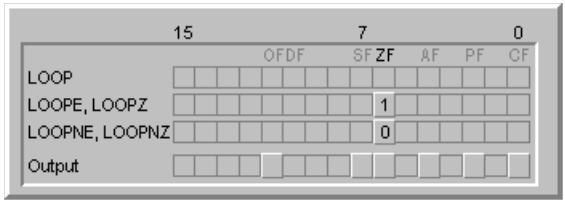
LOOP 8086
LOOPcc 8086

Funktion Führe eine Programmschleife aus (»loop«, »loop on condition«).

Operation

```
IF (AddressSize = 32) THEN
   Count := ECX
ELSE                    ; (AddressSize = 16)
   Count := CX
Count := Count – 1
IF (Instruction = (LOOPE OR LOOPZ)) THEN
   IF (ZF = 1) AND (Count ≠ 0) THEN
      Condition := 1
   ELSE
      Condition := 0
IF (Instruction = (LOOPNE OR LOOPNZ)) THEN
   IF (ZF = 0) AND (Count ≠ 0) THEN
      Condition := 1
   ELSE
      Condition := 0
IF (Instruction = LOOP) THEN
   IF (Count ≠ 0) THEN
      Condition := 1
   ELSE
      Condition := 0
IF (Condition = 1) THEN
   IF (OperandSize = 32) THEN
      EIP := EIP + SignExtend(Dest)
      IF (EIP < SegmentBase) OR (EIP > SegmentLimit) THEN
         #GP(0)
   ELSE                 ; (OperandSize = 16)
      EIP := (EIP + SignExtend(Dest)) AND $0000FFFF
ELSE                    (Condition = 0)
; Fortführung der Befehlsausführung bei EIP
```

Statusflags



Input:
Bis auf LOOP prüfen alle Befehle das ZF.

Output:
Die Flags bleiben unverändert.

Opcodes Bedingte Sprünge, und hierzu zählen die LOOP-Befehle, sind immer relative Sprünge, das heißt, der Operand wird als vorzeichenbehaftete Konstante aufgefasst und zum Inhalt des EIP-Registers addiert.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Src	Bemerkung
E2		LOOP ps	Const8	short; relative; direct
E1		LOOPE ps	Const9	short; relative; direct
		LOOPZ ps	Const8	short; relative; direct
E0		LOOPNE ps	Const8	short; relative; direct
		LOOPNZ ps	Const8	short; relative; direct

Die bei diesen Befehlen möglichen Exceptiontypen sind auf eine general protection exception (#GP) beschränkt. *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	10	0	11	0	11

SYSENTER Pentium II
SYSEXIT Pentium II
SYSCALL
SYSRET

Führe einen »einfachen« und direkten Sprung in eine Routine im Kernelmodus aus und zurück. *Funktion*

Operation

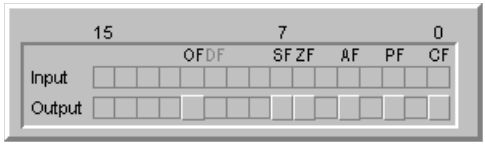
```
IF (Instruction = SYSENTER) THEN
   IF (CPUID[SEP] = 1)
   AND ((Family ≠ 6) OR (Model > 2) OR (Stepping > 2)) THEN
      IF (CR0[PE] = 0)
      OR (SYSENTER_CS_MSR = 0) THEN
         #GP(0)
      EFLAGS[VM] := 0
      EFLAGS[IF] := 0
      EFLAGS[RF] := 0
      ; CodeSegment-Selektor aus MSR auslesen
      CS := (SYSENTER_CS_MSR + 16)
      ; Felder im nicht sichtbaren Teil des Segmentregisters
      ; setzen
      CS[Base] := 0      ; Segmentbasis = 0
      CS[Limit] := $FFFF ; 4-GByte-Segment
      CS[G] := 1         ; 4-kByte-Körnung
      CS[S] := 1         ; Systemsegment
```

```
        CS[Type] := 1011b   ; code, nonconf., readonly, accessed
        CS[D] := 1          ; 32-Bit-Adressen
        CS[DPL] := 0        ; DPL = Kernelmodus
        CS[P] := 1          ; segment present
        ; Feld im sichtbaren Teil des Segmentregisters setzen
        ; (Teil des Selektors)
        CS[RPL] := 0        ; RPL = KernelModus
        ; Stacksegment-Selektor aus MSR auslesen
        SS := (SYSENTER_CS_MSR + 24)
        SS[Base] := 0       ; Segmentbasis = 0
        SS[Limit] := $FFFF ; 4-GByte-Segment
        SS[G] := 1          ; 4-kByte-Körnung
        SS[S] := 1          ; Systemsegment
        SS[Type] := 0011b   ; data, expand-up, writeable, accessed
        SS[D] := 1          ; 32-Bit-Operanden
        SS[DPL] := 0        ; DPL = Kernelmodus
        SS[P] := 1          ; segment present
        ; Feld im sichtbaren Teil des Segmentregisters setzen
        ; (Teil des Selektors)
        SS[RPL] := 0        ; RPL = Kernelmodus
        ; Stackpointer und Einsprungadresse setzen
        ESP := SYSENTER_ESP_MSR
        EIP := SYSENTER_EIP_MSR
IF (Instruction = SYSEXIT) THEN
   IF (CPUID[SEP] = 1)
   AND ((Family ≠ 6) OR (Model > 2) OR (Stepping > 2)) THEN
      IF (CR0[PE] = 0)
      OR (CPL ≠ 0) THEN
         #GP(0)
      ; CodeSegment-Selektor aus MSR auslesen
      CS := (SYSENTER_CS_MSR + 16)
      ; Felder im nicht sichtbaren Teil des Segmentregisters
      ; setzen
      CS[Base] := 0       ; Segmentbasis = 0
      CS[Limit] := $FFFF ; 4-GByte-Segment
      CS[G] := 1          ; 4-kByte-Körnung
      CS[S] := 1          ; Systemsegment
      CS[Type] := 1011b   ; code, nonconf., readonly, accessed
      CS[D] := 1          ; 32-Bit-Adressen
      CS[DPL] := 3        ; DPL = Usermodus
      CS[P] := 1          ; segment present
      ; Feld im sichtbaren Teil des Segmentregisters setzen
      ; (Teil des Selektors)
      CS[RPL] := 3        ; RPL = User-Modus
      ; Stacksegment-Selektor aus MSR auslesen
```

```
      SS := (SYSENTER_CS_MSR + 24)
      SS[Base] := 0      ; Segmentbasis = 0
      SS[Limit] := $FFFF ; 4-GByte-Segment
      SS[G] := 1         ; 4-kByte-Körnung
      SS[S] := 1         ; Systemsegment
      SS[Type] := 0011b  ; data, expand-up, writeable, accessed
      SS[D] := 1         ; 32-Bit-Operanden
      SS[DPL] := 3       ; DPL = Usermodus
      SS[P] := 1         ; segment present
      ; Feld im sichtbaren Teil des Segmentregisters setzen
      ; (Teil des Selektors)
      SS[RPL] := 3       ; RPL = User-Modus
      ; Stackpointer und Einsprungadresse setzen
      ESP := ECX
      EIP := EDX
IF (Instruction = SYSCALL) THEN
   IF (CPUID[SCE] = 1) THEN
      EFLAGS[VM] := 0
      EFLAGS[IF] := 0
      ECX := EIP
      ; CodeSegment-Selektor aus STAR auslesen
      CS := STAR[47..32]
      ; Felder im nicht sichtbaren Teil des Segmentregisters
      ; setzen
      CS[Base] := 0      ; Segmentbasis = 0
      CS[Limit] := $FFFF ; 4-GByte-Segment
      CS[G] := 1         ; 4-kByte-Körnung
      CS[S] := 1         ; Systemsegment
      CS[Type] := 1011b  ; code, nonconf., readonly, accessed
      CS[D] := 1         ; 32-Bit-Adressen
      CS[DPL] := 0       ; DPL = Kernelmodus
      CS[P] := 1         ; segment present
      ; Feld im sichtbaren Teil des Segmentregisters setzen
      ; (Teil des Selektors)
      CS[RPL] := 0       ; RPL = KernelModus
      ; Stacksegment-Selektor aus MSR auslesen
      SS := STAR[47..32] + 8
      SS[Base] := 0      ; Segmentbasis = 0
      SS[Limit] := $FFFF ; 4-GByte-Segment
      SS[G] := 1         ; 4-kByte-Körnung
      SS[S] := 1         ; Systemsegment
      SS[Type] := 0011b  ; data, expand-up, writeable, accessed
      SS[D] := 1         ; 32-Bit-Operanden
      SS[DPL] := 0       ; DPL = Kernelmodus
      SS[P] := 1         ; segment present
```

```
      ; Feld im sichtbaren Teil des Segmentregisters setzen
      ; (Teil des Selektors)
      SS[RPL] := 0       ; RPL = Kernelmodus
      ; Stackpointer und Einsprungadresse setzen
      EIP := STAR[31..00]
IF (Instruction = SYSRET) THEN
   IF (CPUID[SCE] = 1) THEN
      EFLAGS[IF] := 1
      ; CodeSegment-Selektor aus MSR auslesen
      CS := STAR[63..48]
      ; Felder im nicht sichtbaren Teil des Segmentregisters
      ; setzen
      CS[Base] := 0      ; Segmentbasis = 0
      CS[Limit] := $FFFF ; 4-GByte-Segment
      CS[G] := 1         ; 4-kByte-Körnung
      CS[S] := 1         ; Systemsegment
      CS[Type] := 1011b  ; code, nonconf., readonly, accessed
      CS[D] := 1         ; 32-Bit-Adressen
      CS[DPL] := 3       ; DPL = Usermodus
      CS[P] := 1         ; segment present
      ; Feld im sichtbaren Teil des Segmentregisters setzen
      ; (Teil des Selektors)
      CS[RPL] := 3       ; RPL = User-Modus
      ; Stacksegment-Selektor aus MSR auslesen
      SS := STAR[63..48] + 8
      SS[Base] := 0      ; Segmentbasis = 0
      SS[Limit] := $FFFF ; 4-GByte-Segment
      SS[G] := 1         ; 4-kByte-Körnung
      SS[S] := 1         ; Systemsegment
      SS[Type] := 0011b  ; data, expand-up, writeable, accessed
      SS[D] := 1         ; 32-Bit-Operanden
      SS[DPL] := 3       ; DPL = Usermodus
      SS[P] := 1         ; segment present
      ; Feld im sichtbaren Teil des Segmentregisters setzen
      ; (Teil des Selektors)
      SS[RPL] := 3       ; RPL = User-Modus
      ; Stackpointer und Einsprungadresse setzen
      EIP := ECX
```

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert, wohl aber ggf. die Systemflags VM, IF, RF.

Opcodes

Alle Mnemonics werden in Zwei-Byte-Opcodes ohne Operanden übersetzt, da diese implizit angegeben sind. Es sind die modellspezifischen Register (MSR) $174 bis $176 (SYSENTER_CS_MSR, SYSENTER_ESP_MSR und SYSENTER_EIP_MSR bzw. $C0000081 (SysCall/SysRet Target Address Register; STAR). Darüber hinaus werden auch die Register ECX und EDX verwendet.

Opcode		Mnemonic	Operanden
8-Bit-Operanden	32(16)-Bit-Op.		Dest
0F 34		SYSENTER	MSR($174), MSR($175), MSR($176)
0F 35		SYSEXIT	MSR($174), MSR($175), MSR($176)
0F 05		SYSCALL	MSR($C0000081)
0F 07		SYSRET	MSR($C0000081)

Die grau unterlegten Operanden sind implizit vorgegeben. SYSCALL und SYSRET sind nur auf AMD-Prozessoren verfügbar und daher inkompatibel.

Exceptions

Bei diesen Befehlen kommt als Exceptionquelle nur eine general protection exception #GP in Frage.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	(32,) 61	0	(32,) 61	0	1
#UD	12	0	12	0	12

Die grau unterlegten Exceptions kommen nur bei SYSCALL/SYSRET vor. Grund 32 ist nur bei SYSEXIT/SYSRET möglich

2.8 Andere bedingte Instruktionen

CMOVcc — Pentium Pro

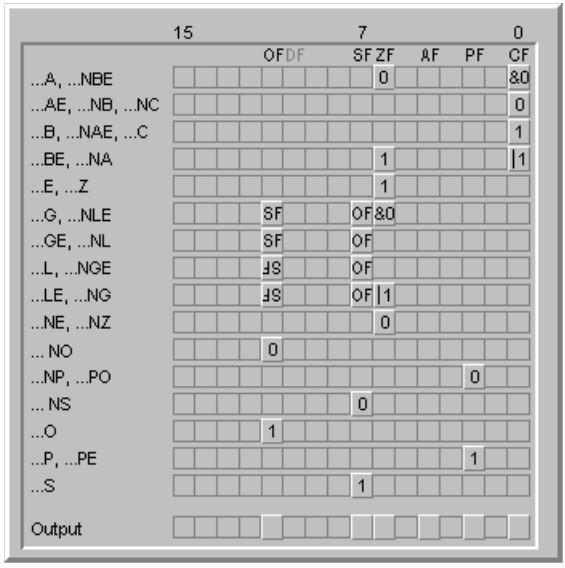
Funktion

Kopiere ein Datum, wenn eine Bedingung erfüllt ist (»move on condition«).

Operation

```
IF Condition THEN      ; wenn die Bedingung erfüllt ist
   Dest := Src
```

Statusflags



Input: Je nach CMOVcc-Befehl werden einzelne Flags oder Flag-Kombinationen ausgewertet. »&« steht dafür, dass beide Bedingungen zusammen erfüllt sein müssen, »|« dafür, dass eine Bedingung erfüllt sein muss. Auf dem Kopf stehende Beschriftungen zeigen umgekehrte Flag-Stellung.

Output: Die Flags bleiben unverändert.

Opcodes Der Befehl ist nur mit 32- bzw. 16-Bit-Operaden verfügbar.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	0F 47 /r	CMOVA r, r/m	Reg	Reg / Mem
		CMOVNBE r, r/m	Reg	Reg / Mem
	0F 43 /r	CMOVAE r, r/m	Reg	Reg / Mem
		CMOVNB r, r/m	Reg	Reg / Mem
		CMOVNC r, r/m	Reg	Reg / Mem
	0F 42 /r	CMOVB r, r/m	Reg	Reg / Mem
		CMOVNAE r, r/m	Reg	Reg / Mem
		CMOVC r, r/m	Reg	Reg / Mem
	0F 46 /r	CMOVBE r, r/m	Reg	Reg / Mem
		CMOVNA r, r/m	Reg	Reg / Mem
	0F 44 /r	CMOVE r, r/m	Reg	Reg / Mem
		CMOVZ r, r/m	Reg	Reg / Mem
	0F 4F /r	CMOVG r, r/m	Reg	Reg / Mem
		CMOVNLE r, r/m	Reg	Reg / Mem

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	`0F 4D /r`	CMOVGE r, r/m CMOVNL r, r/m	Reg Reg	Reg / Mem Reg / Mem
	`0F 4C /r`	CMOVL r, r/m CMOVNGE r, r/m	Reg Reg	Reg / Mem Reg / Mem
	`0F 4E /r`	CMOVLE r, r/m CMOVNG r, r/m	Reg Reg	Reg / Mem Reg / Mem
	`0F 45 /r`	CMOVNE r, r/m CMOVNZ r, r/m	Reg Reg	Reg / Mem Reg / Mem
	`0F 41 /r`	CMOVNO r, r/m	Reg	Reg / Mem
	`0F 4B /r`	CMOVNP r, r/m CMOVPO r, r/m	Reg Reg	Reg / Mem Reg / Mem
	`0F 49 /r`	CMOVNS r, r/m	Reg	Reg / Mem
	`0F 40 /r`	CMOVO r, r/m	Reg	Reg / Mem
	`0F 4A /r`	CMOVP r, r/m CMOVPE r, r/m	Reg Reg	Reg / Mem Reg / Mem
	`0F 48 /r`	CMOVS r, r/m	Reg	Reg / Mem

Exceptions

Die bei diesem Befehl möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	1, 2

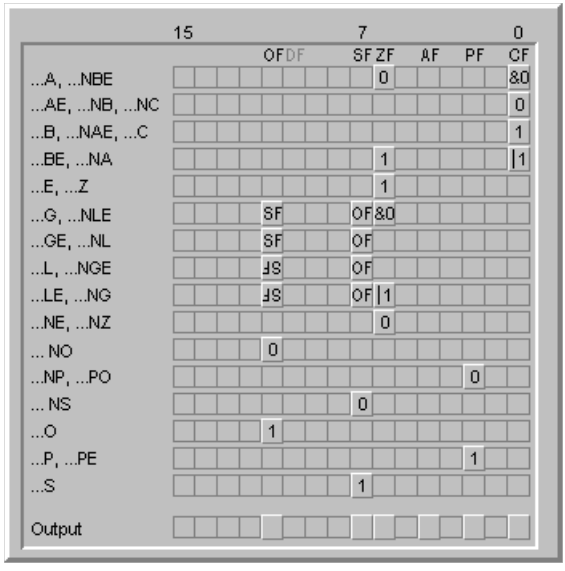
SETcc — Pentium

Funktion Setze je nach Bedingung ein Byte auf den Wert 0 oder 1 (»set on condition«).

Operation

```
IF Condition THEN      ; wenn die Bedingung erfüllt ist
   Dest := 1
ELSE
   Dest := 0
```

Statusflags



Input: Je nach SETcc-Befehl werden einzelne Flags oder Flag-Kombinationen ausgewertet. »&« steht dafür, dass beide Bedingungen zusammen erfüllt sein müssen, »|« dafür, dass eine Bedingung erfüllt sein muss. Auf dem Kopf stehende Beschriftungen zeigen umgekehrte Flag-Stellung.

Output: Die Flags bleiben unverändert.

Opcodes Der Befehl ist nur mit 32- bzw. 16-Bit-Operaden verfügbar.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 97 /r	SETA r/m	Reg / Mem	
		SETNBE r/m	Reg / Mem	
	0F 93 /r	SETAE r/m	Reg / Mem	
		SETNB r/m	Reg / Mem	
		SETNC r/m	Reg / Mem	
	0F 92 /r	SETB r/m	Reg / Mem	
		SETNAE r/m	Reg / Mem	
		SETC r/m	Reg / Mem	
	0F 96 /r	SETBE r/m	Reg / Mem	
		SETNA r/m	Reg / Mem	

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	0F 94 /r	SETE r/m SETZ r/m	Reg / Mem Reg / Mem	
	0F 9F /r	SETG r/m SETNLE r/m	Reg / Mem Reg / Mem	
	0F 9D /r	SETGE r/m SETNL r/m	Reg / Mem Reg / Mem	
	0F 9C /r	SETL r/m SETNGE r/m	Reg / Mem Reg / Mem	
	0F 9E /r	SETLE r/m SETNG r/m	Reg / Mem Reg / Mem	
	0F 95 /r	SETNE r/m SETNZ r/m	Reg / Mem Reg / Mem	
	0F 91 /r	SETNO r/m	Reg / Mem	
	0F 9B /r	SETNP r/m SETPO r/m	Reg / Mem Reg / Mem	
	0F 99 /r	SETNS r/m	Reg / Mem	
	0F 90 /r	SETO r/m	Reg / Mem	
	0F 9A /r	SETP r/m SETPE r/m	Reg / Mem Reg / Mem	
	0F 98 /r	SETS r/m	Reg / Mem	

Exceptions

Die bei diesem Befehl möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	1, 2

2.9 Programmunterbrechungen durch Interrupts/Exceptions

INT	**8086**
INT3	**8086**
INTO	**8086**

Funktion Unterbreche die aktuelle Programmausführung und rufe einen Interrupt-Handler auf (»interrupt«, »interrupt on overflow«).

Operation

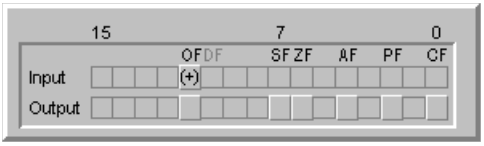
```
IF (PE = 0) THEN        ; Real Mode
   Vector := Dest * 4
   ; Liegt der Vektor außerhalb der IVT? ACHTUNG: die IVT hat
   ; 32-Bit-Einträge (16-Bit-Selektor und 16-Bit-Offset), die
   ; IDT des protected mode 64-Bit-Einträge (Descriptoren)!!
   IF (Vector + 3 > InterruptVektorTableLimit) THEN
      #GP(0)
   ; Genügend Platz für die Parameter auf dem Stack vorhanden?
   IF ((ESP + 6) > StackSegmentLimit) THEN
      #SS(0)
   PushStack(FLAGS)     ; NICHT EFLAGS! 16 Bit!
   IF := 0
   TF := 0
   AC := 0
   PushStack(CS)
   PushStack(IP)        ; NICHT EIP! 16 Bit!
   Offset := InterruptVectorTable[Vector]
   CS := InterruptvectorTable[Vector + 2]
   EIP := Offset AND $0000FFFF

ELSE                    ; Protected Mode
   ; Software-Interrupts im protected mode können, müssen aber
   ; nicht mit einem ErrorCode aufgerufen werden. Sein Spei-
   ; cherabbild wird in Abbildung 2.17 gezeigt. Im Rahmen
   ; der INT-Bearbeitung des Prozessors werden die drei
   ; niedrigstwertigen Bits gesetzt.
   IF (ErrorCode existent) THEN
      EXT := ErrorCode[0]

   ; Virtual 8086 mode: INT n nur mit IOPL = 3
   IF (VM = 1) AND (IOPL < 3) AND (Instruction = INT) THEN
      #GP(0)
   Selector := (Dest * 8)
   Descriptor := InterruptDescriptorTable[Selector]
   NewType := Descriptor[Type]
```

```
NewDPL := Descriptor[DPL]
Present := Descriptor[AVL]
IDT := 1
IF ((Selektor + 7) > InterruptDescriptorTableLimit)
OR (NewType ≠ interrupt, trap oder task gate) THEN
   #GP(Selector + [IDT] + [EXT])
IF (NewDPL < CPL) THEN
   #GP(Selector + [IDT])
IF (NOT Present) THEN
   #NP(Selector + [IDT] + [EXT])

IF (NewType = task gate) THEN
   ; Interrupt über task gate
   ; Neuer Selektor für TSS aus task gate descriptor
   ; Neuer Descriptor für TSS
   Selector := Descriptor[Selector]
   Global := Selector[G]
   IF (NOT Global)
   OR (Selector > DescriptorTableLimit) THEN
      #GP(Selector)
   Descriptor := DescriptorTable[Selector]
   Present := Descriptor[AVL]
   Busy := Descriptor[B]
   IF Busy THEN
      #GP(Selector)
   IF (NOT Present) THEN
      #NP(Selector)
   ; Der NestedTaskSwitch bei INT unterscheidet sich
   ; vom TaskSwitch bei z.B. JMP dadurch, dass in das PTL-
   ; Feld (previous task link) des TSS der Selector des
   ; TSS des aktuellen tasks eingetragen wird. Dies er-
   ; möglicht den Rücksprung in den rufenden task.
   NestedTaskSwitch(Selector)
   IF (ErrorCode erforderlich) THEN
      IF ((ESP + 4) > StackSegmentLimit) THEN
         #GP(0)
      PushStack(ErrorCode)
   IF (EIP > CodeSegmentLimit) THEN
      #GP(0)

ELSE               ; (NewType ≠ task gate)
   ; Interrupt via trap oder interrupt gate
   ; Neuer Selektor für Codesegment aus trap/interrupt gate
   ; descriptor
   ; Neuer Descriptor für Codesegment
   Selector := Descriptor[Selector]
```

```
IF (Selector = Nullselector) THEN
   #GP(0 + [EXT])
IF (Selector > DescriptorTableLimit) THEN
   #GP(Selector + [EXT])
Descriptor := DescriptorTable[Selector]
NewType := Descriptor[Type]
NewDPL := Descriptor[DPL]
IF (NewType ≠ code segment) OR (NewDPL > CPL) THEN
   #GP(Selector + ErrorCode[Bit0]
Present := Descriptor[AVL]
IF (NOT Present) THEN
   #NP(Selector + [EXT])

; Inter-Privileg-Level-Interrupt
IF (NewType = nonconforming code segment)
AND (NewDPL < CPL) THEN
   IF (VM = 0) THEN
      IF (aktueller TSS = 32-Bit-TSS) THEN
         ; Suche im TSS nach dem zu verwendenden Stack
         ; Privilegstufe 0 (DPL = 0) -> TSS-Offset  4
         ; Privilegstufe 1 (DPL = 1) -> TSS-Offset 12
         ; Privilegstufe 2 (DPL = 2) -> TSS-Offset 20
         FirstByte := (NewDPL * 8) + 4
         IF ((FirstByte + 7) > TSSLimit THEN
            #TS([task register] ; aktueller TSS Selektor
         NewSS := WORD(TSS[FirstByte + 4])
         NewESP := DWORD(TSS[FirstByte])
      ELSE       ; (aktueller TSS = 16-Bit-TSS
         ; Suche im TSS nach dem zu verwendenden Stack
         ; Privilegstufe 0 (DPL = 0) -> TSS-Offset  2
         ; Privilegstufe 1 (DPL = 1) -> TSS-Offset 6
         ; Privilegstufe 2 (DPL = 2) -> TSS-Offset 10
         FirstByte := (NewDPL * 4) + 2
         IF ((FirstByte + 3) > TSSLimit THEN
            #TS([task register] ; aktueller TSS Selektor
         NewSS := WORD(TSS[FirstByte + 2])
         NewESP := ZeroExtend(WORD(TSS[FirstByte]))
      IF (NewSS = Nullselektor) THEN
         #TS(EXT)
      IF (NewSS > DescriptorTableLimit) THEN
         #TS(NewSS + EXT)
      ; Lesen des Deskriptors, auf den NewSS zeigt
      ;  -> Neues Stacksegment
      SSDescriptor := DescriptorTable[Selector]
      SSRPL := NewSS[RPLField]
```

```
SSDPL := SSDescriptor[DPLField]
SSType := SSDescriptor[TypeField]
IF (SSRPL ≠ NewDPL)
OR (SSDPL ≠ NewDPL)
OR (SSType ≠ writeable data segment) THEN
   #TS(NewSS + EXT)
IF (NewSS nicht »present« markiert) THEN
   #SS(NewSS + EXT)
IF (Offset > NewCodeSegmentLimit) THEN
   #GP(0)
IF (CallGateSize = 32) THEN
   IF ((ErrorCode existent)
      AND (FreeStackSpace < 24))
   OR (NOT ErrorCode existent)
      AND (FreeStackSize < 20)) THEN
      #SS(NewSS + EXT)
ELSE       ; (CallGateSize = 16)
   IF ((ErrorCode existent)
      AND (FreeStackSpace < 12))
   OR (NOT ErrorCode existent)
      AND (FreeStackSize < 10)) THEN
      #SS(NewSS + EXT)
OldSS := SS
OldESP := ESP
SS[visible] := NewSS
SS[hidden] := SSDescriptor
ESP := NewESP
CS[visible] := Selector
CS[hidden] := Descriptor
EIP := Offset
PushStack(OldSS)
PushStack(OldESP)
PushStack(EFLAGS)
PushStack(OldCS)
PushStack(OldEI
IF (ErrorCode existent) THEN
   PushStack(ErrorCode)
CPL := NewDPL
CS[RPL] := CPL
IF (OldSS = interrupt gate) THEN
   IF := 0
TF := 0
VM := 0
RF := 0
NT := 0
```

```
   ELSE          ; (VM = 1)
      ; Interrupt FROM virtual 8086 mode
      IF (NewDPL ≠ 0) THEN
         #GP(Selector)

; Intra-Privileg-Level-Interrupt
ELSE             ; (DPL ≥ CPL)
   IF (VM = 1) THEN
      #GP(Selector)
   IF (NewType = conforming code segment)
   OR (NewDPL = CPL) THEN
      IF (CallGateSize = 32) THEN
         IF ((ErrorCode existent)
            AND (FreeStackSpace < 16))
         OR (NOT ErrorCode existent)
            AND (FreeStackSize < 12)) THEN
            #SS(NewSS + EXT)
      ELSE       ; (CallGateSize = 16)
         IF ((ErrorCode existent)
            AND (FreeStackSpace < 8))
         OR (NOT ErrorCode existent)
            AND (FreeStackSize < 6)) THEN
            #SS(NewSS + EXT)
      IF (Offset > CodeSegmentLimit) THEN
         #GP(0)
      CS[visible] := Selector
      CS[hidden] := Descriptor
      EIP := Offset
      PushStack(EFLAGS)
      PushStack(OldCS)
      PushStack(OldEI
      IF (ErrorCode existent) THEN
         PushStack(ErrorCode)
      CS[RPL] := CPL
      IF (OldSS = interrupt gate) THEN
         IF := 0
      TF := 0
      VM := 0
      RF := 0
      NT := 0
   ELSE          ; (DPL > CPL)
      #GP(Selector + [EXT])
```



Statusflags

Input: bei INT und INT3 keiner, INTO prüft das OF . Output: Status-Flags werden nicht verändert, es sei denn, es erfolgt ein task switch. Dann sind alle Flags betroffen. Die Systemflags IF, TF, NT, AC, RF und VM können je nach Betriebsmodus verändert werden.

Opcodes

Es gibt drei Versionen des INT-Befehls: Der allgemein gültige INT-Befehl, bei dem der zu nutzende Interruptvektor als 8-Bit-Konstante angegeben wird, sowie die operandenlosen Befehle INT 3, der einen INT mit Vektor $3 auslöst, und INTO, der als bedingter Interrupt aufgefasst werden kann, da er nur dann einen Interrupt mit Vektor $4 auslöst, wenn das overflow flag gesetzt ist.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	ohne Operanden		Dest	Bemerkung
	CC	INT 3	3	Debug Interrupt
CD		INT c	Const8	
	CE	INTO	4	interrupt on overflow

Die grau unterlegten Operanden sind implizit vorgegeben.

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine general protection exception (#GP), eine segment not present exception #NP, eine page fault exception (#PF), eine stack fault exception (#SS) sowie eine invalid TSS exception (#TS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	17 4, 12, 16, 24, 30, 42, 47, 64	0 selector	8, 10 4, 12, 16, 24, 30, 42, 47	0 selector	8, 10 ./.
#NP	1	selector	-	-	./.
#PF	1	fault code	1	fault code	./.
#SS	5 4, 8	0 selector	- 4, 7	0 selector	- -
#TS	1, 3, 4, 5, 6	selector	1, 3, 4, 5, 6	selector	./.

IRET 8086
IRETD 80386
IRETW 80386

Funktion Kehre aus einem Interrupt-Handler zurück (»return from interrupt«).

Operation

```
IF (PE = 0) THEN       ; Real Mode
      IF (StackOperandSize = 32) THEN
      IF ((ESP – 12) > StackSegmentLimit) THEN
         #SS(0)
      EIP := PopStack          ; 32-Bit-POP
      IF (EIP > SegmentLimit) THEN
         #GP(0)
      Selector := PopStack     ; 32-Bit-POP
      CS := LowWord(Selector)
      Temp := PopStack         ; 32-Bit-POP
      EFlags := (EFlags AND $001A0000) OR (Temp AND $00257FD5)
   ELSE             ; (StackOperandSize = 16)
      IF ((ESP – 6) > StackSegmentLimit)) THEN
         #SS(0)
      Offset := PopStack       ; 16-Bit-POP
      EIP := ZeroExtend(Offset)
      IF (EIP > SegmentLimit) THEN
         #GP(0)
      CS := PopStack           ; 16-Bit-POP
      Temp := PopStack         ; 16-Bit-POP
      EFlags := ZeroExtend(Temp)

ELSE                   ; protected mode

   ; Return FROM virtual 8086 mode
   IF (VM = 1) THEN
      IF (IOPL = 3) THEN
         IF (StackOperandSize = 32) THEN
            IF ((ESP – 12) > StackSegmentLimit) THEN
               #SS(0)
            EIP := PopStack               ; 32-Bit-POP
            Selector := PopStack          ; 32-Bit-POP
            CS := LowWord(Selector)
            EFlags := PopStack            ; 32-Bit-POP
         ELSE             ; (StackOperandSize = 16)
            IF ((ESP – 6) > StackSegmentLimit)) THEN
               #SS(0)
            EIP := ZeroExtend(PopStack)   ; 16-Bit-POP
```

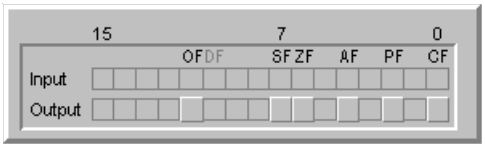
```
         CS := PopStack                ; 16-Bit-POP
         EFlags := ZeroExtend(PopStack) ; 16-Bit-POP
   ELSE             ; (IOPL < 3)
      #GP(0)        ; trap to virtual 8086 monitor

; Return from task
ELSEIF (NT = 1) THEN
   ; Im task register (TR) steht der Selector des aktuellen
   ; tasks
   OldSelector := [TR]
   OldDescriptor := DescriptorTable[Selector]
   ; Im previous task link field des aktuellen TSS steht
   ; der Selector des tasks, der den aktuellen gerufen hat
   Selector := CurrentTSS[PTL]
   IF (Selector[G] = local)
   OR (Selector > DescriptorTableLimit) THEN
      #GP(OldSelector)
   Descriptor := DescriptorTable[Selector]
   NewType := Descriptor[Type]
   Busy := Descriptor[B]
   Present := [Descriptor[P]
   IF (NewType ≠ TSS) OR ( NOT Busy) THEN
      #GP(Selector)
   IF ( NOT Present) THEN
      #NP(Selector)
   TaskSwitch(Selector)     ; NICHT NestedTaskSwitch!
   OldDescriptor[B] := NOT busy
   IF (EIP > CodeSegmentLimit) THEN
      #GP(0)

; Return nicht von v86m und task
ELSE
   IF (StackOperandSize = 32) THEN
      IF ((ESP – 12) > StackSegmentLimit) THEN
         #SS(0)
      Offset := PopStack             ; 32-Bit-POP
      Selector := PopStack           ; 32-Bit-POP
      TFlags := PopStack             ; 32-Bit-POP
   ELSE             ; (StackOperandSize = 16)
      IF ((ESP – 6) > StackSegmentLimit)) THEN
         #SS(0)
      Offset := ZeroExtend(PopStack) ; 16-Bit-POP
      CS := PopStack                 ; 16-Bit-POP
      TFlags := ZeroExtend(PopStack) ; 16-Bit-POP
```

```
 ; return TO virtual 8086 mode
 IF ((TFlags[VM] = 1) AND (CPL = 0) THEN
    IF ((ESP – 24) > StackSegmentLimit) THEN
       #SS(0)
    IF (Offset > CodeSegmentLimit) THEN
       #GP(0)
    CS := Selector
    EIP := Offset
    EFlags := TFlags
    TempESP := PopStack
    TempSS := PopStack
    ES := PopStack
    DS := PopStack
    FS := PopStack
    GS := PopStack
    ESP := TempESP
    SS := TempSS

; Return to protected mode
ELSE
    IF (Selector = Nullselektor) THEN
       #GP(0)
    IF (Selector > DescriptorTableLimit) THEN
       #GP(Selector)
    Descriptor := DescriptorTable[Selector]
    NewType := Descriptor[Type]
    NewDPL := Descriptor[DPL]
    IF (NewType ≠ CodeSegment)
    OR (Selector[RPL] < CPL)
    OR ((NewType = conforming code segment)
       AND (NewDPL > SelectorRPL)) THEN
       #GP(Selector)
    IF (Descriptor[p] = not present) THEN
       #NP(Selektor)

    IF (Selector[RPL] > CPL) THEN
       ; Rückkehr in ein niedrigerprivilegiertes Segment.
       ; In das aktuelle Segment musste somit über ein trap
       ; oder interrupt gate gesprungen worden sein. Der
       ; Stack hat dann den Aufbau:
       ;
       ; ESP +   Inhalt       Bemerkungen
       ;
       ; 16 (20) RetSS        Rückkehr-StackSegment
       ; 12 (16) RetESP       Rückkehr-Stackpointer
```

```
; 08 (12) EFLAGS       Rückkehr-EFlagregister
; 04 (08) RetCS        Rückkehr-Codesegment
; 00 (04) RetEIP       Rückkehr-EIP
;    (00) ErrorCode    bei manchen Exceptions
;
; Das macht insgesamt 20 (24) Bytes, da auch die
; 16-Bit-Selektoren als DWORD gePUSHt wurden. Ana-
; loge Betrachtungen gelten für 16-Bit-Umgebung
; Falls ein ErrorCode übergeben wurde, lag es in
; der Verantwortung des Handlers, diesen vom Stack
; zu nehmen, bevor IRET aufgerufen wurde! Daher
; gelten die nicht geklammerten Offset zu ESP,
; obwohl der INT-Befehl ggf. den Stack mit den
; geklammerten Offsets belegt hat.
; Im Rahmen der Prüfungen bis an diese Stelle wur-
; den bereits EIP, CS und EFlags vom Stack genom-
; men, sodass nun nur noch die 8 den Stack be-
; treffenden Bytes übrig bleiben.
IF (OperandSize = 32) THEN
   ; Liegen die 8 restlichen Bytes außerhalb des
   ; Segments?
   IF (ESP – 8 > StackSegmentLimit) THEN
      #SS(0)
   NewSS := [ESP + 8]
   NewESP := [ESP]
ELSE          ; (OperandSize = 16)
   ; Liegen die 4 restlichen Bytes außerhalb des
   ; Segments?
   IF (ESP – 4 > StackSegmentLimit) THEN
      #SS(0)
NewSS := [ESP + 4]
NewESP := [ESP]
IF (NewSS = Nulldescriptor) THEN
   #GP(0)
IF (NewSS > DescriptorTableLimit) THEN
   #GP(Selector)
SSDescriptor := DescriptorTable[NewSS]
IF (NewSS[RPL] ≠ Selector[RPL])
OR (SSDescriptor[Type] ≠ writeable data segment)
OR (SSDescriptor[DPL] ≠ Selector[RPL]) THEN
   #GP(Selector)
IF (SSDescriptor[P] nicht »present« markiert) THEN
   #SS(NewSS)
IF (Offset > Descriptor[Limit]) THEN
   #GP(0)
```

```
      EIP := Offset
      CS[visible] := Selector
      CS[hidden] := Descriptor
      IF (OperandSize = 32) THEN
         FOR (F=CF,PF,AF,ZF,SF,TF,DF,OF,NT,RF,AC,ID) DO
            EFLAGS[F] := TFlags[F]
         IF (CPL ≤ IOPL) THEN
            EFLAGS[IF] := TFlags[IF
         IF (CPL = 0) THEN
            FOR (F=IOPL,VM, VIF, VIP) DO
               EFLAGS[F] := TFlags[F]
         ESP := PopStack    ; 32-Bit-POP
         NewSS := PopStack  ; 32-Bit-POP
         SS := LowWord(NewSS)
      ELSE          ; (OperandSize = 16)
         FOR (F=CF,PF,AF,ZF,SF,TF,DF,OF,NT) DO
            EFLAGS[F] := TFlags[F]
         IF (CPL ≤ IOPL) THEN
            EFLAGS[IF] := TFlags[IF
         IF (CPL = 0) THEN
            EFLAGS[IOPL] := TFlags[IOPL]
         NewSP := PopStack  ; 16-Bit-POP
         ESP := ZeroExtend(NewSP)
         SS := PopStack     ; 16-Bit-POP
      CPL := Selector[RPL]
      ; Die restlichen Segmentregister prüfen!
      FOR (TestSelector = [DS], [ES], [FS], [GS]) DO
         TestDescriptor := DescriptorTable[TestSelector]
         TestType := TestDescriptor[Type]
         TestDPL := TestDescriptor[DPL]
         IF ((TestType = data segment)
             OR (TestType = nonconforming code segment))
         AND (CPL > TestDPL) THEN
            TestSelector := Nullselektor

   ELSE             ; (Selector[RPL] <= CPL)
      ; Rückkehr aus gleichprivilegiertem Segment
      ; Da hier kein Stack-Switch erfolgt, sind bereits
      ; alle Parameter vom Stack genommen worden. Es
      ; geht also nur noch um Zuordnungen!
      EIP := Offset
      CS[visible] := Selector
      CS[hidden] := Descriptor
      IF (Selector > DescriptorTableLimit) THEN
         #GP(0)
```

```
IF (OperandSize = 32) THEN
   FOR (F=CF,PF,AF,ZF,SF,TF,DF,OF,NT,RF,AC,ID) DO
      EFLAGS[F] := TFlags[F]
   IF (CPL ≤ IOPL) THEN
      EFLAGS[IF] := TFlags[IF
   IF (CPL = 0) THEN
      FOR (F=IOPL,VM, VIF, VIP) DO
         EFLAGS[F] := TFlags[F]
ELSE          ; (OperandSize = 16)
   FOR (F=CF,PF,AF,ZF,SF,TF,DF,OF,NT) DO
      EFLAGS[F] := TFlags[F]
   IF (CPL ≤ IOPL) THEN
      EFLAGS[IF] := TFlags[IF
   IF (CPL = 0) THEN
      EFLAGS[IOPL] := TFlags[IOPL]
```



Statusflags

Input: keiner
Output: Die Statusflags werden durch den korrespondierenden INT-Befehl auf den Stack kopiert. Der INT-Befehl verändert auch einige Systemflags. Nach Rückkehr durch IRET sind die Statusflags dann unverändert, an einigen Systemflags können sich Änderungen ergeben haben. Im Vergleich zum EFlags-Inhalt in der Interrupt-Prozedur kann sich am Zustand der Statusflags etwas geändert haben, muss aber nicht. Die Systemflags sind jedoch zumindest teilweise verändert worden.

Opcodes

IRET kommt in drei Versionen vor: Als IRETW in 16-Bit-Umgebungen und als IRETD in 32-Bit-Umgebungen. Dennoch teilen beide Formen den gleichen Opcode: den von IRET, dem Chamäleon, das in 16-Bit-Umgebungen mit IRETW und in 32-Bit-Umgebungen mit IRETD identisch ist. Daher machen die wenigsten Assembler tatsächlich einen Unterschied, viele unterstützen die Mnemonics IRETD und/oder IRETW erst gar nicht. Die weitaus meisten Assembler benutzen IRET, IRETW und IRETD als Synonyme, so auch MASM und TASM

Opcode keine Operanden	Mnemonic
CF	IRET IRETD IRETW

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine segment not present exception #NP, eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	7, 10 14, 27, 35, 37, 39, 43, 45, 47, 64	0 selector	10, 31 -	0 -	10 ./.
#NP	5	selector	-	-	./.
#PF	1	fault code	1	fault code	./.
#SS	9	0	9	0	9

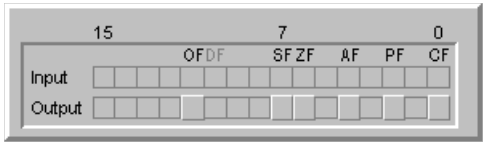
BOUND 80186

Funktion Prüfe, ob ein Wert innerhalb bestimmter Grenzen liegt.

Operation

```
IF (OperandSize = 32) THEN
   Lower := Src[31..00]
   Upper := Src[63..32]
ELSE                    ; (OperandSize = 16)
   Lower := Src[15..00]
   Upper := Src[31..16]
IF (Dest < Lower) OR (Dest > Upper) THEN
   #BR
```

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert.

Opcodes Die obere und untere Grenze werden über eine Speicherstelle angegeben. Dort repräsentiert das erste DoubleWord bzw. Word die vorzeichenbehaftete untere, das zweite die vorzeichenbehaftete obere Grenze, gegen die der im Zielregister stehende Wert getestet werden soll.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	62	BOUND r, m	Reg	Mem

Die bei diesem Befehl möglichen Exceptiontypen sind eine alignment check exception (#AC), eine page fault exception (#PF) und eine stack fault exception (#SS). *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#BR	1	-	1	-	1
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1
#UD	2	-	2	-	2
#AC	1	0	1	0	./:

2.10 Instruktionen zur gezielten Veränderung des Flagregisters

POPF **8086**
POPFD **80386**
(POPFW **80386)**

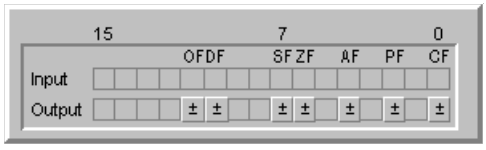
Kopiere ein Datum vom Stack in das EFlags-Register (»pop flags«). *Funktion*

Operation

```
IF (VM = 0) THEN       ; nicht v86-Modus
   IF (CPL = 0) THEN
      IF (OperandSize = 32) THEN
         EFLAGS := PopStack
         VIP := 0
         VIF := 0
         VM bleibt unverändert
      ELSE             ; (OperandSize = 16)
         FLAGS := PopStack
   ELSE                ; (CPL > 0)
      IF (OperandSize = 32) THEN
         EFLAGS := PopStack
         VIP := 0
         VIF := 0
         VM bleibt unverändert
         IOPL bleibt unverändert
      ELSE             ; (OperandSize = 16)
         FLAGS := PopStack
         IOPL bleibt unverändert
```

```
ELSE                    ; (V86-Modus)
   IF (IOPL = 3) THEN
      IF (OperandSize = 32) THEN
         EFLAGS := PopStack
         VIP bleibt unverändert
         VIF bleibt unverändert
         VM bleibt unverändert
         RF bleibt unverändert
         IOPL bleibt unverändert
      ELSE              ; (IOPL < 3)
         #GP(0)
```

Statusflags



Input: keiner

Output: Alle status- und modusabhängigen und die meisten System-Flags werden verändert.

Opcodes POPF und POPFD haben den identischen Opcode. Die Operanden sind alle impliziert:

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	9D	POPF POPFD (POPFW)	EFlags-Register	SS:(E)SP

Die grau unterlegten Operanden sind implizit an- und vorgegeben.

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	1	0	1	0	-
#GP	31, 57	0	31, 57	0	-
#PF	1	fault code	1	fault code	-
#SS	9	0	9	0	9

Bemerkungen

Nicht alle Assembler implementieren das Mnemonic POPFW, sondern setzen es mit POPF gleich. In diesem Fall steht POPF für die 16-Bit-Operandenform, POPFD für die 32-Bit-Operandenform und es liegt in der Verantwortung des Programmierers, die korrekten Mnemonics zu verwenden. Sollen Assemblermodule für beide Fälle geschrieben werden, ist dann eine Fallunterscheidung mit Hilfe der bedingten Assemblierung erforderlich.

MASM und TASM dagegen sind Assembler, die POPFW implementieren. Es erzwingt die Nutzung von 16-Bit-Operanden auch in 32-Bit-Umgebungen und setzt dort ein operand size override prefix vor den Opcode. Analog erzwingt POPFD die Verwendung von 32-Bit-Operanden und kodiert in 16-Bit-Umgebungen ebenfalls ein operand size override prefix. POPF repräsentiert den Opcode $9D ohne prefix und ist somit das Chamäleon unter den Befehlen: In 16-Bit-Umgebungen werden dadurch 16-Bit-Operanden verwendet, in 32-Bit-Umgebungen 32-Bit-Operanden.

PUSHF 80186
PUSHFD 80386
(PUSHFW 80386)

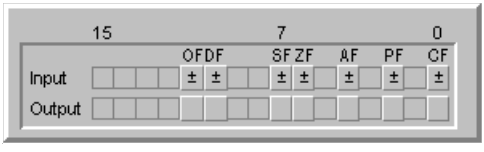
Funktion

Kopiere den Inhalt des EFlags-Registers auf den Stack (»push flags«).

Operation

```
; real mode, protected mode oder (v86 mode mit IOPL = 3)
IF (PE = 0)
OR ((PE = 1) AND ((VM = 0) OR ((VM = 1) AND (IOPL = 3)))) THEN
   IF (OperandSize = 32) THEN
      Temp := EFLAGS AND $00FCFFFF
      PushStack(Temp)
   ELSE                ; (OperandSize = 16)
      PushStack(FLAGS)
ELSE                   ; V86 mode mit IOPL < 3
   #GP(0)
```

Statusflags



Input: Die Status- und die meisten Systemflags werden auf den Stack kopiert.
Output: Die Flags werden nicht verändert.

Opcodes PUSHF und PUSHFD haben den identischen Opcode. Die Operanden sind alle impliziert:

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	9C	PUSHF PUSHFD (PUSHFW)	SS:(E)SP	EFlags

Die grau unterlegten Operanden sind implizit an- und vorgegeben.

Exceptions Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception #GP, eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	-
#GP	-	0	31	0	-
#PF	1	fault code	1	fault code	-
#SS	10	0	-	0	-

Bemerkungen Nicht alle Assembler implementieren das Mnemonic PUSHFW, sondern setzen es mit PUSHF gleich. In diesem Fall steht PUSHF für die 16-Bit-Operandenform, PUSHFD für die 32-Bit-Operandenform und es liegt in der Verantwortung des Programmierers, die korrekten Mnemonics zu verwenden. Sollen Assemblermodule für beide Fälle geschrieben werden, ist dann eine Fallunterscheidung mit Hilfe der bedingten Assemblierung erforderlich.

MASM und TASM dagegen sind Assembler, die PUSHFW implementieren. Es erzwingt die Nutzung von 16-Bit-Operanden auch in 32-Bit-Umgebungen und setzt dort ein operand size override prefix vor den Opcode. Analog erzwingt PUSHFD die Verwendung von 32-Bit-Operanden und kodiert in 16-Bit-Umgebungen ebenfalls ein operand size override prefix. PUSHA repräsentiert den Opcode $9C ohne prefix und ist somit das Chamäleon unter den Befehlen: In 16-Bit-Umgebungen werden dadurch 16-Bit-Operanden verwendet, in 32-Bit-Umgebungen 32-Bit-Operanden.

LAHF 8086
SAHF 8086

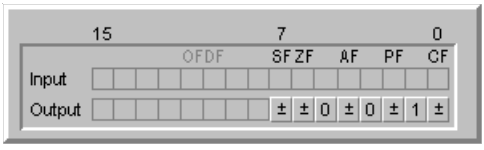
Funktion

Kopiere Statusflags aus dem FLAG-Register in das AH-Register oder umgekehrt (»load AH with flags«, »store AH into flags«).

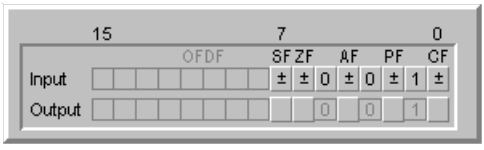
Operation

```
IF (Instruction = LAHF) THEN
   AH := (LowByte(EFLAGS) AND $D5) OR $02
ELSE                    ; (Instruction = SAHF)
   LowByte(EFLAGS) := (AH AND $D5) OR $02
```

Statusflags



SAHF
Input: Die Flagstellungen spielen keine Rolle.
Output: Die Flags 0 bis 15 werden anhand der Bits im Operanden gesetzt.



LAHF
Input: Die Flags 0 bis 15 werden in den Operanden kopiert.
Output: Die Flags bleiben unverändert

Opcodes

Beide Befehle besitzen mit dem EFlags-Register und dem AH-Register nur implizite Operanden.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	9F	LAHF	AH	EFLAGS
	9E	SAHF	EFLAGS	AH

Die grau unterlegten Operanden sind implizit an- und vorgegeben.

Exceptions

Beide Befehle erzeugen keine Exceptions.

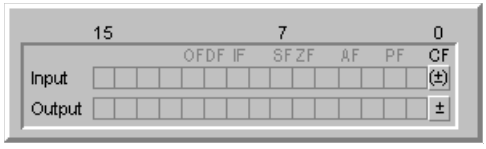
CLC, CMC, STC 8086
CLD, STD 8086
CLI, STI 8086

Funktion Gib dem spezifizierten Flag einen definierten Zustand (»clear/complement/set carry/direction/interrupt flag«).

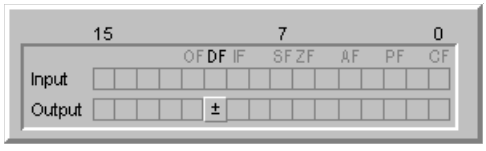
Operation

```
IF (Instruction = CLC) THEN
   CF := 0
ELSEIF (Instruction = CMC) THEN
   CF := NOT(CF)
ELSEIF (Instruction = STC) THEN
   CF := 1
ELSEIF (Instruction = CLD) THEN
   DF := 0
ELSEIF (Instruction = STD) THEN
   DF := 1
ELSEIF (Instruction = CLI) THEN
   IF := 0
ELSE                    ; (Instruction = STI)
   IF := 1
```

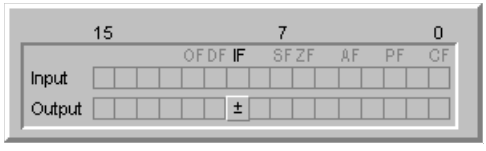
Statusflags



CLC, CMC und STC:
Input: CF wird nur durch CMC ausgewertet.
Output: ±: gesetzt oder gelöscht je nach Befehl.



CLD, STD:
Input: keiner
Output: ±: gesetzt oder gelöscht je nach Befehl.



CLI, STI:
Input: keiner
Output: ±: gesetzt oder gelöscht je nach Befehl.

Diese Befehle haben alle keinen Operanden, sondern wirken direkt auf einzelne Felder des EFlags-Registers. *Opcodes*

Opcode	Mnemonic	Operanden	
		Dest	Src
F8	CLC		
F5	CMC		
F9	STC		
FC	CLD		
FD	STD		
FA	CLI		
FB	STI		

Exceptions kommen nur bei CLI und STI vor! CLC, CMC, STC, CLD und STD können keine Exceptions auslösen. *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	33	0	33	0	-

2.11 String-Instruktionen

CMPS / CMPSB / CMPSW / CMPSD **8086 (80386)**
LODS / LODSB / LODSW / LODSD **8086 (80386)**
MOVS / MOVSB / MOVSW / MOVSD **8086 (80386)**
SCAS / SCASB / SCASW / SCASD **8086 (80386)**
STOS / STOSB / STOSW / STOSD **8086 (80386)**

Vergleiche zwei Strings miteinander (»compare strings«), lade Daten aus einem String (»load string«), kopiere Daten zwischen zwei Strings (»move string«), suche ein Datum in einem String (»scan string«) oder schreibe ein Datum in einen String (»store string«). *Funktion*

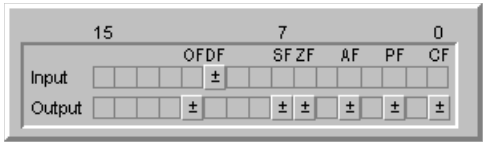
Operation

```
; Die Operandengröße ergibt sich entweder aus dem Mnemonic
; oder den Dummy-Operanden des Befehls
IF (OperandSize = 32) THEN
   Inc := 4
ELSEIF (OperandSize = 16) THEN
   Inc := 2
```

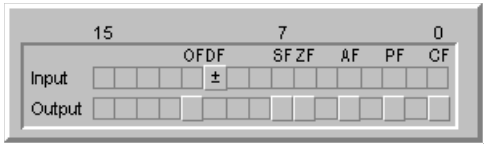
```
ELSE                    ; (OperandSize = 8)
   Inc := 1
IF (DF = 1) THEN
   Inc := -Inc

IF (Instruction = CMPS, CMPSB, CMPSW, CMPSD) THEN
   ; Temp hat doppelte Operandengröße
   Temp := DS:[(E)SI] – ES:[(E)DI]
   (E)SI := (E)SI + Inc
   (E)DI := (E)DI + Inc
   IF (High(Temp) ≠ 0) THEN
      CF := 1           ; (Temp > Max) oder (Temp < Min)
   ELSE
      CF := 0
   IF (Temp[MSB] = Dest[MSB]) THEN
      OF := 0           ; kein Übertrag aus dem /in das MSB
   ELSE
      OF := 1           ; Übertrag aus dem / in das MSB!
   IF (Temp[4] = Dest[4]) THEN
      AF := 0           ; kein Übertrag aus dem /in das Nibble
   ELSE
      AF := 1           ; Übertrag aus dem / in das Nibble!
   IF (Temp = 0) THEN
      ZF := 1
   ELSE
      ZF := 0
   PF := 1              ; Annahme: Null = gerade Parität
   FOR (I := 0) TO (I = OperandSize - 1) DO
      PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
   SF := Dest[MSB]      ; SF = Kopie vom MSB des Zieloperanden

IF (Instruction = LODS, LODSB, LODSW, LODSD) THEN
   IF (OperandSize = 32) THEN
      EAX := DS:[(E)SI]
   ELSEIF (OperandSize = 16) THEN
      AX := DS:[(E)SI]
   ELSE                 ; (OperandSize = 8)
      AL := DS:[(E)SI]
   (E)SI := (E)SI + Inc

IF (Instruction = MOVS, MOVSB, MOVSW, MOVSD) THEN
   ES:[(E)DI] := DS:[(E)SI]
   (E)SI := (E)SI + Inc
   (E)DI := (E)DI + Inc
```

```
IF (Instruction = SCAS, SCASB, SCASW, SCASD) THEN
   ; Temp hat doppelte Operandengröße
   IF (OperandSize = 32) THEN
      Temp := EAX
   ELSEIF (OperandSize = 16) THEN
      Temp := AX
   ELSE                ; (OperandSize = 8)
      Temp := AL
   Temp := Temp – ES:[(E)DI]
   (E)DI := (E)DI + Inc
   IF (High(Temp) ≠ 0) THEN
      CF := 1          ; (Temp > Max) oder (Temp < Min)
   ELSE
      CF := 0
   IF (Temp[MSB] = Dest[MSB]) THEN
      OF := 0          ; kein Übertrag aus dem /in das MSB
   ELSE
      OF := 1          ; Übertrag aus dem / in das MSB!
   IF (Temp[4] = Dest[4]) THEN
      AF := 0          ; kein Übertrag aus dem /in das Nibble
   ELSE
      AF := 1          ; Übertrag aus dem / in das Nibble!
   IF (Temp = 0) THEN
      ZF := 1
   ELSE
      ZF := 0
   PF := 1             ; Annahme: Null = gerade Parität
   FOR (I := 0) TO (I = OperandSize - 1) DO
      PF := PF XOR Temp[I]; wenn Bit gesetzt, Parität ändern
   SF := Dest[MSB]     ; SF = Kopie vom MSB des Zieloperanden

IF (Instruction = STOS, STOSB, STOSW, STOSD) THEN
   IF (OperandSize = 32) THEN
      ES:[(E)DI] := EAX
   ELSEIF (OperandSize = 16) THEN
      ES:[(E)DI] := AX
   ELSE                ; (OperandSize = 8)
      ES:[(E)DI] := AL
   (E)DI := (E)DI + Inc
```

Statusflags



CMPS und SCAS:
Input: Das DF steuert die Bearbeitungsrichtung des/der Strings.
Output: ±: gesetzt oder gelöscht gemäß Ergebnis.



LODS, MOVS, STOS:
Input: Das DF steuert die Bearbeitungsrichtung des/der Strings.
Output: Die Flags werden nicht verändert.

Opcodes Bei allen Befehlen handelt es sich um einen Ein-Byte-Opcode. Die Operanden sind alle implizit vorgegeben, auch wenn die Mnemonics teilweise einen expliziten Operanden vorgeben. Dieser Dummy-Operand dient nur dazu, die Operandengröße festzustellen.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op		Dest	Src
A6	A7	CMPS, m, m	ES:(E)DI	DS:(E)SI
A6		CMPSB	ES:(E)DI	DS:(E)SI
	A7	CMPSW	ES:(E)DI	DS:(E)SI
	A7	CMPSD	ES:(E)DI	DS:(E)SI
AC	AD	LODS m	Akkumulator	DS:(E)SI
AC		LODSB	AL	DS:(E)SI
	AD	LODSW	AX	DS:(E)SI
	AD	LODSD	EAX	DS:(E)SI
A4	A5	MOVS m, m	ES:(E)DI	DS:(E)SI
A4		MOVSB	ES:(E)DI	DS:(E)SI
	A5	MOVSW	ES:(E)DI	DS:(E)SI
	A5	MOVSD	ES:(E)DI	DS:(E)SI
AE	AF	SCAS m	Akkumulator	ES:(E)DI
AE		SCASB	AL	ES:(E)DI
	AF	SCASW	AX	ES:(E)DI
	AF	SCASD	EAX	ES:(E)DI

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op		Dest	Src
AA	AB	STOS m	ES:(E)DI	Akkumulator
AA		STOSB	ES:(E)DI	AL
	AB	STOSW	ES:(E)DI	AX
	AB	STOSD	ES:(E)DI	EAX

Die grau unterlegten Operanden sind implizit vorgegeben.

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1

Bemerkung

Bitte beachten Sie, dass die Mnemonics CMPSD und MOVSD auch bei den SSE2-Befehlen verwendet werden. Dort werden sie als »Erweiterung« der Befehle CMPSS und MOVSS auf PackedDoubles verwendet. Einen tatsächlichen Konflikt bei der Nutzung der Mnemonics gibt es jedoch nicht, da der Programmierer aufgrund des Kontextes weiß, welcher Befehl gemeint ist, und der Assembler die korrekte Befehlssequenz aufgrund der Operanden feststellen kann.

INS / INSB / INSW / INSD 8086 (80386)
OUTS / OUTSB / OUTSW / OUTSD 8086 (80386)

Funktion

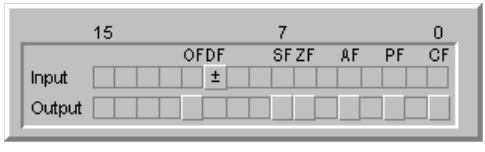
Kopiere Daten aus einem Port in einen String oder umgekehrt (»input to string«, »output from string«).

Operation

```
; Die Operandengröße ergibt sich entweder aus dem Mnemonic
; oder den Dummy-Operanden des Befehls
IF (Instruction = INS, INSB, INSW, INSD) THEN
   Port := Src
ELSE               ; (Instruction = OUTS, OUTSB, OUTSW, OUTSD)
   Port := Dest
```

```
IF (OperandSize = 32) THEN
   Inc := 4
   I/OBits := 4
ELSEIF (OperandSize = 16) THEN
   Inc := 2
   I/OBits := 2
ELSE                    ; (OperandSize = 8)
   Inc := 1
IF (DF = 1) THEN
   Inc := -Inc
   I/OBits := 1
IF (PE = 1) AND ((CPL > IOPL) OR (VM = 1) THEN
   FOR (I = 0) TO (I = I/OBits – 1) DO
      IF (ActualTSS[I/OPermissionTable][Port + I] = 1)
      OR ((Port + I) > I/OPermissionTableLimit) THEN
         #GP(0)
; Real mode oder protected mode mit CPL ≤ IOPL)

IF (Instruction = INS, INSB, INSW, INSD) THEN
   IF (OperandSize = 32) THEN
      EAX := ES:[(E)DI]
   ELSEIF (OperandSize = 16) THEN
      AX := ES:[(E)DI]
   ELSE                 ; (OperandSize = 8)
      AL := ES:[(E)DI]
   (E)DI := (E)DI + Inc

IF (Instruction = OUTS, OUTSB, OUTSW, OUTSD) THEN
   IF (OperandSize = 32) THEN
      DS:[(E)SI] := EAX
   ELSEIF (OperandSize = 16) THEN
      DS:[(E)SI] := AX
   ELSE                 ; (OperandSize = 8)
      DS:[(E)SI] := AL
   (E)SI := (E)SI + Inc
```

Statusflags



Input: DF steuert die Bearbeitungsrichtung des/der Strings.
Output: Die Flags werden nicht verändert.

Opcodes Bei allen Befehlen handelt es sich um einen Ein-Byte-Opcode. Die Operanden sind alle implizit vorgegeben, auch wenn die Mnemonics teilweise zwei expliziten Operanden vorgeben. Diese Dummy-Operanden dienen nur dazu, die Operandengröße festzustellen.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op		Dest	Src
6C	6D	INS m, DX	ES:(E)DI	Port [DX]
6C		INSB	ES:(E)DI	Port [DX]
	6D	INSW	ES:(E)DI	Port [DX]
	6D	INSD	ES:(E)DI	Port [DX]
6E	6F	OUTS DX, m	Port [DX]	ES:(E)DI
6E		OUTSB	Port [DX]	ES:(E)DI
	6F	OUTSW	Port [DX]	ES:(E)DI
	6F	OUTSD	Port [DX]	ES:(E)DI

Die grau unterlegten Operanden sind implizit vorgegeben.

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine page fault exception (#PF) und eine stack fault exception (#SS).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	8, 20, 34	0	8, 48	0	8
#PF	1	fault code	1	fault code	./.

2.12 Präfixe

<segment override prefix> **8086**
<operand size override prefix> **80386**
<address size override prefix> **80386**
<branch hints> **Pentium 4**
LOCK **8086**

Funktion

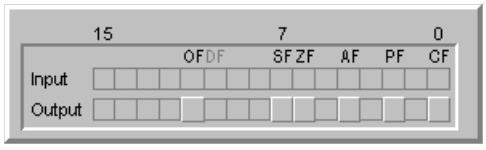
Modifiziere die folgende Instruktion.

Operation

```
IF (Präfix = segment override prefix) THEN
   Benutze durch Präfix angegebenes Segment als Segment-
   Anteil zur Adress-Berechnung
IF (Präfix = operand size override prefix) THEN
   IF (Größenattribut im Daten- oder Stacksegment = 32) THEN
      Operandengröße für folgenden Befehl := 16 Bit
```

```
   ELSE
      Operandengröße für folgenden Befehl := 32 Bit
IF (Präfix = address size override prefix) THEN
   IF (Größenattribut im Codesegment = 32) THEN
      Adressgröße für folgenden Befehl := 16 Bit
   ELSE
      Adressgröße für folgenden Befehl := 32 Bit
IF (Präfix = branch hints) THEN
   IF (branch hit = branch taken) THEN
      Branch prediction logic ← vermutlich Verzweigung
   ELSE                ; (branch not taken)
      Branch prediction logic ← vermutlich keine Verzweigung
IF (Präfix = LOCK) THEN
   AssertLock#(Ausführungsdauer des nächsten Befehls)
```

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert.

Opcodes Präfixe besitzen streng genommen keinen Opcode, da sie selbst keine eigenständige Aktion ausführen, sondern nur im Zusammenhang mit einem Befehl wirken. Sie sind somit »Umschalter«, die einen Befehls-Opcode *modifizieren*.

»Opcode«	Mnemonic	Aktion	Bemerkungen
2E	CS:	segment override	Neuer Bezugspunkt: angegebenes Segment anstelle von Standardsegment (CS, DS und SS) Nur gültig mit Befehlen, die explizit oder implizit eine Segmentangabe benötigen
3E	DS:	segment override	
26	ES:	segment override	
64	FS:	segment override	
65	GS:	segment override	
36	SS:	segment override	
66	-	operand size override	Umschalten der Operandengröße
67	-	address size override	Umschalten der Adressgröße
2E	-	branch taken	Nur gültig mit bedingten Sprungbefehlen
3E	-	branch not taken	
F0	LOCK	assert lock	Nur sinnvoll in Mehrprozessorsystemen

Exceptions

Die Präfixe können bis auf LOCK selbst keine Exceptions auslösen, allerdings ist es möglich, dass der folgende Befehl, auf den sich die Präfixe beziehen, Exceptions auslösen. LOCK kann nur eine Exception vom Typ invalid opcode (#UD) generieren.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#UD	7	-	7	-	7

Die Exception ist nur bei LOCK definiert.

REP 8086
REPcc 8086

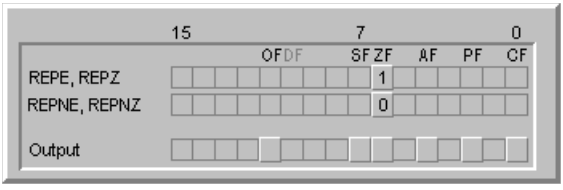
Funktion

Wiederhole die folgende Instruktion (»repeat«, »repeat until condition«).

Operation

```
IF (AdressSize = 32) THEN
   Count := [ECX]
ELSE                    ; (AddressSize = 16)
   Count := [CX]
IF (Count > 0) THEN
   REPEAT
      Bediene anhängige Interrupts
      Führe die folgende String-Operation aus
      Count := Count – 1
   UNTIL (Count = 0)
   OR ((Präfix = REPZ, REPE) AND (ZF = 0))
   OR ((Präfix = REPNZ, REPNE) AND (ZF = 1))
```

Statusflags



Input: Bis auf REP prüfen alle Präfixe das ZF

Output: Die Flags bleiben unverändert

Opcodes

In Verbindung mit INSx, MOVSx, OUTSx und STOSx heißt der Befehl mit dem Opcode $F3 REP, in Verbindung mit CMPSx und SCASx REPE bzw. REPZ. Der Opcode $F2 ist nur in Verbindung mit CMPSx und SCASx erlaubt und heißt dann REPNE oder REPNZ.

»Opcode«	Mnemonic	Aktion	Bemerkungen
F3	REP	repeat until (E)CX = 0	mit INS, MOVS, OUTS, LODS und STOS
F3	REPE REPZ	repeat until (E)CX = 0 or ZF = 1	mit CMPS, SCAS
F2	RENE RENZ	repeat until (E)CX = 0 or ZF = 1	mit CMPS, SCAS

Exceptions Die Präfixe können selbst keine Exceptions auslösen, allerdings ist es möglich, dass der folgende Befehl, auf den sich die Präfixe beziehen, Exceptions auslöst.

2.13 Adressierungsinstruktionen

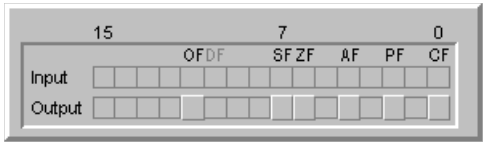
LEA 8086

Funktion Berechne eine effektive Adresse (»load effective address«).

Operation

```
IF (OperandSize = 32) THEN
   IF (AddressSize = 32) THEN
      EA := EffectiveAddress(Src)
   ELSE                 ; (AdressSize = 16)
      EA := ZeroExtend(EffectiveAddress(Src))
ELSE                    ; (OperandSize = 16)
   IF (AddressSize = 32) THEN
      EA := LowWord(EffectiveAddress(Src))
   ELSE                 ; (AdressSize = 16)
      EA := EffectiveAddress(Src)
Dest := EA
```

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert.

Opcodes LEA stellt einen Ein-Byte-Opcode dar, der zwei Operanden hat: Ziel der Berechnung der effektiven Adresse muss ein Register sein, Quelle ist die Adresse einer Speicherstelle, die durch ein Symbol (Variablenname) repräsentiert wird.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op**		**Dest**	**Src**
	8D /r	LEA r, m	Reg	Mem

Exceptions

Die bei diesem Befehl möglichen Exceptiontypen beschränken sich auf eine invalid opcode exception (#UD).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#UD	2	-	2	-	2

LDS 8086
LES 8086
LFS 80386
LGS 80386
LSS 80386

Funktion

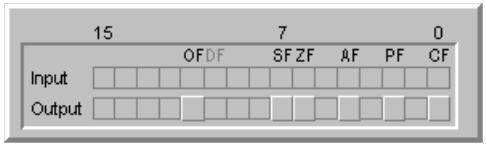
Lade eine vollständige Logische Adresse (»load segment register xx and ...«).

Operation

```
IF (OperandSize = 32) THEN
   Selector := Src[47..32]
   Offset := Src[31..00]
ELSE                    ; (OperandSize = 16)
   Selector := Src[31..16]
   Offset := Src[15..00]
IF (Instruction = LDS) THEN
   SegReg := DS
ELSEIF (Instruction = LES) THEN
   SegReg := ES
ELSEIF (Instruction = LFS) THEN
   SegReg := FS
ELSEIF (Instruction = LGS) THEN
   SegReg := GS
ELSE              ;(Instruction = LSS)
   SegReg := SS
IF (PE = 1) THEN       ; protected mode
   Descriptor := DescriptorTable[Selector]
   Type := Descriptor[Type]
   DPL := Descriptor[DPL]
   RPL := Selector[RPL]
   Present := Descriptor[AVL]
```

```
   IF (Instruction = LSS)
      IF (Selector = Nullselektor) THEN
         #GP(0)
      IF (Selector > DescriptorTableLimit)
      OR (RPL ≠ CPL)
      OR (Type = nonwritable data segment)
      OR (DPL ≠ CPL) THEN
         #GP(Selector)
      IF (NOT Present) THEN
         #SS(Selector)
      SS[visible] := Selector
      SS[hidden] := Descriptor
   ELSE               ; (Instruction = LDS, LES, LFS, LGS)
      IF (Selector ≠ Nullselektor) THEN
         IF (Selector > DescriptorTableLimit)
         OR (Type ≠ (data segment OR readable code segment))
         OR ((Type = (data segment OR nonconforming segment))
            AND (RPL > DPL) AND (CPL > DPL)) THEN
            #GP(Selector)
         IF (NOT Present) THEN
            #NP(Selector)
         SegReg[visible] := Selector
         SegReg[hidden] := Descriptor
      ELSE            ; (Selector = Nullselector)
         SegReg[visible] := 0
         SegReg[hidden] := not valid
ELSE                  ; virtual 8086 mode oder real mode
   SegReg := Selector
Dest := Offset
```

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert.

Opcodes Bei allen Befehlen, die ab dem 8086 implementiert wurden, handelt es sich um einen Ein-Byte-Opcode. Die ab dem 80386 implementierten Befehle besitzen einen Zwei-Byte-Opcode. Der Quelloperand ist eine Speicherstelle, die je nach Umgebung einen Pointer vom Typ Selector16:Offset16 oder Selector16:Offset32 besitzt. Zieloperand muss ein Register sein.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	C5	LDS r, m	Reg	Mem48 / Mem32
	C4	LES r, m	Reg	Mem48 / Mem32
	0F B4	LFS r, m	Reg	Mem48 / Mem32
	0F B5	LGS r, m	Reg	Mem48 / Mem32
	0F B2	LSS r, m	Reg	Mem48 / Mem32

Exceptions

Die bei diesen Befehlen möglichen Exceptiontypen sind eine alignment check exception (#AC), eine general protection exception (#GP), eine segment not present exception (#NP). eine page fault exception (#PF), eine stack fault exception (#SS) sowie eine invalid opcode exception (#UD).

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 6, 8	0	8	0	8
	62, 63	Selector	-	Selector	./.
#NP	4	Selector	-	Selector	./.
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1
	4	Selector	-	Selector	./.
#UD	2	-	2	-	2

2.14 Spezielle Instruktionen

CPUID — Pentium (später 80486)

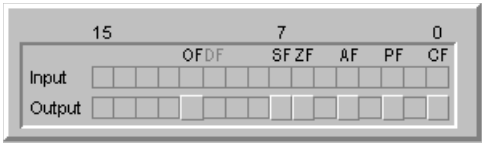
Funktion

Generiere Informationen zur CPU (»CPU identification«).

Operation

```
Function := Read(EAX)
SetRegisters(Information[Function])
```

Statusflags



Input: keiner
Output: Status-Flags werden nicht verändert.

Opcodes Der Befehl CPUID besteht aus einem Zwei-Byte-Opcode ohne weitere Bytes in der Befehlssequenz.

Opcode	Mnemonic	Operanden	
		Dest	Src
0F A2	CPUID	EAX, EBX, ECX, EDX	EAX

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions CPUID löst keine Exceptions aus.

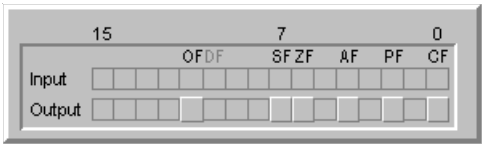
ENTER 80186
LEAVE 80186

Funktion Richte einen (verschachtelten) Stack-Rahmen ein bzw. entferne ihn wieder.

Operation

```
IF (Instruction = ENTER) THEN
   Level := Src2 MOD 32
   IF (StackOperandSize = 32) THEN
      PushStack(EBP)
      Temp := ESP
   ELSE                ; (StackOperandSize = 16)
      PushStack(BP)
      Temp := SP
   IF (Level > 0) THEN
      FOR (I = 1) TO (i = Level - 1) DO
         IF (OperandSize = 32) THEN
            IF (StackOperandSize = 32) THEN
               EBP := EBP – 4
               PushStack(DWord[EBP])
            ELSE       ; (StackOperandSize = 1&)
               BP := BP – 4
               PushStack(DWord([BP])
         ELSE          ; (OperandSize = 16) THEN
            IF (StackOperandSize = 32) THEN
               EBP := EBP – 2
               PushStack(Word[EBP])
            ELSE       ; (StackOperandSize = 1&)
               BP := BP – 2
               PushStack(Word([BP])
   IF (OperandSize = 32) THEN
      PushStack(Temp)
```

```
   ELSE                ; (OperandSize = 16)
      PushStack(Temp)
   IF (StackOperandSize = 32) THEN
      EBP := Temp
      ESP := EBP – Src1
   ELSE                ; (StackOperandSize = 16)
      BP := Temp
      SP := BP – Src1
ELSE                   ; (Instruction = LEAVE)
   IF (StackOperandSize = 32) THEN
      ESP := EBP
   ELSE                ; (StackOperandSize = 16)
      SP := BP
   IF (OperandSize = 32) THEN
      EBP := PopStack
   ELSE                ; (OperandSize = 16)
      BP := PopStack
```



Statusflags

Input: keiner
Output: Status-Flags werden nicht verändert.

Opcodes

ENTER erwartet zwei Operanden, bei denen es sich um zwei 8-Bit-Konstanten handelt. Der erste Operand kodiert die Größe des Bereichs für lokale Variablen, der zweite die Verschachtelungstiefe (»nesting level«). LEAVE benötigt keine Operanden.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Scr1	Src2
	C8		Const16	$00
		ENTER c16, c8	Const16	$01
			Const16	Const8
	C9	LEAVE		

Exceptions

Bei ENTER sind eine page fault exception (#PF) sowie eine stack fault exception (#SS) möglich:

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#PF	1	fault code	1	fault code	./.
#SS	2	0	2	0	2

LEAVE dagegen löst ggf. eine alignment check exception #AC, eine general protection exception #GP oder eine page fault exception #PF aus.

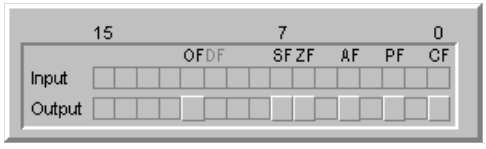
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	18	0	19	0	19
#PF	1	fault code	1	fault code	./.

NOP 8086

Funktion Mach dir einen schönen Feiertakt (»no operation«).

Operation
```
IF (OperandSize = 32) THEN
   XCHG EAX, EAX
ELSE                    ; (OperandSize = 16)
   XCHG AX, AX
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Befehl NOP besteht aus einem Ein-Byte-Opcode ohne weitere Bytes in der Befehlssequenz.

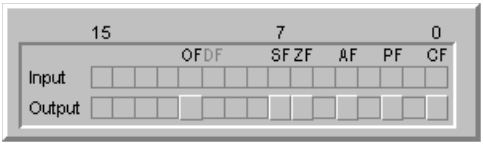
Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
90	NOP			

Exceptions NOP löst keine Exceptions aus.

WAIT (FWAIT) 8086

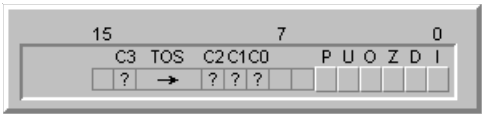
Funktion Synchronisiere FPU und CPU, indem wartende FPU-Exceptions behandelt werden.

Operation
```
IF (#MF pending) THEN
   CallExceptionHandler(#MF)
```



Input: keiner
Output: Status-Flags werden nicht verändert.

CPU-Statusflags



Alle Flags des condition codes sind undefiniert. FWAIT ist stackneutral, Exceptions werden nicht ausgelöst.

FPU-Statusflags

Opcodes

FWAIT und WAIT sind Synonyme für die CPU-Instruktion $9B, die die Synchronisierung zwischen FPU-Exceptionbearbeitung und CPU-Instruktionen herbeiführt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
9B	FWAIT / WAIT			

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

UD2 — Pentium Pro

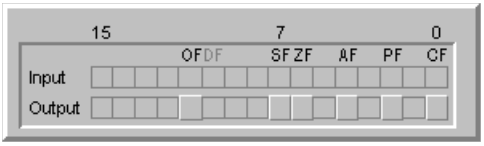
Funktion

Löse zu Prüfzwecken eine Exception des Typs #UD aus (»undefined instruction«).

Operation

```
CallExceptionHandler(#UD)
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Befehl UD2 besteht aus einem Zwei-Byte-Opcode ohne weitere Bytes in der Befehlssequenz.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
0F 0B	UD2			

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#UD	11	-	11	-	11

2.15 System-Instruktionen

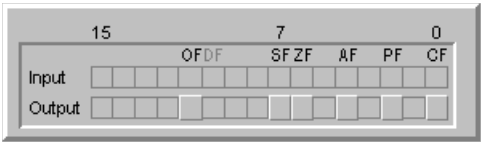
LGDT **80286**
LIDT **80286**

Funktion Schreibe einen neuen Selektor für die *global descriptor table* bzw. die *interrupt descriptor table* in das entsprechende Register (»load global descriptor table«, »load interrupt descriptor table«).

Operation

```
IF (Instruction = LGDT) THEN
   IF (OperandSize = 32) THEN
      GDTR[Limit] := Src[15..00]
      GDTR[Base] := Src[47..16]
   ELSE                ; (OperandSize = 16)
      GDTR[Limit] := Src[15..00]
      GDTR[Base] := Src[47..16] AND $00FFFFFF
ELSE                   ; (Instruction = LIDT)
   IF (OperandSize = 32) THEN
      IDTR[Limit] := Src[15..00]
      IDTR[Base] := Src[47..16]
```

```
ELSE                ; (OperandSize = 16)
   IDTR[Limit] := Src[15..00]
   IDTR[Base] := Src[47..16] AND $00FFFFFF
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Der Quelloperand ist eine Speicherstelle, die einen Pointer vom Typ Selector16:Offset32 besitzt. In 16-Bit-Umgebungen wird in Offset32 ein 24-Bit-Zeiger übergeben.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 01 /2	LGDT m		Mem48
	0F 01 /3	LIDT m		Mem48

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	3, 8, 32	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1
#UD	2	-	2	-	2

Erklärung

Die Befehle sind absolut privilegierte Befehle, was bedeutet, dass sie nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden können.

SGDT 80286
SIDT 80286

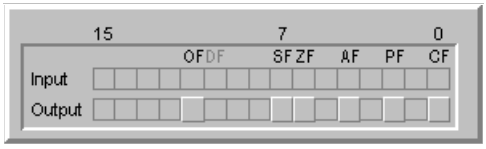
Funktion

Hole den Selektor für die *global descriptor table* bzw. die *interrupt descriptor table* aus dem entsprechenden Register (»set global descriptor table«, »set interrupt descriptor table«).

Operation

```
IF (Instruction = LGDT) THEN
   IF (OperandSize = 32) THEN
      Dest[15..00] := GDTR[Limit]
      Dest[47..16] := GDTR[Base]
   ELSE                ; (OperandSize = 16)
      Dest[15..00] := GDTR[Limit]
      Dest[39..16] := GDTR[Base]
ELSE                   ; (Instruction = LIDT)
   IF (OperandSize = 32) THEN
      Dest[15..00] := IDTR[Limit]
      Dest[47..16] := IDTR[Base]
   ELSE                ; (OperandSize = 16)
      Dest[15..00] := IDTR[Limit]
      Dest[39..16] := IDTR[Base]
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Quelloperand ist eine Speicherstelle, die einen Pointer vom Typ Selector16:Offset32 besitzt. In 16-Bit-Umgebungen wird in Offset32 ein 24-Bit-Zeiger übergeben.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	**32(16)-Bit-Op.**		**Dest**	**Src**
	0F 01 /0	SGDT m	Mem48	
	0F 01 /1	SIDT m	Mem48	

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#GP	3, 8, 20	0	8	0	8
#PF	1	fault code	1	fault code	./.
#SS	1	0	1	0	1
#UD	4	-	4	-	4

Erklärung

Die Befehle sind zwar nicht absolut privilegierte Befehle wie ihre schreibenden Counterparts (LGDT, LIDT), was bedeutet, dass sie nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden können. Dennoch sind ihr Nutzen und ihre Nutzbarkeit begrenzt.

LLDT 80286

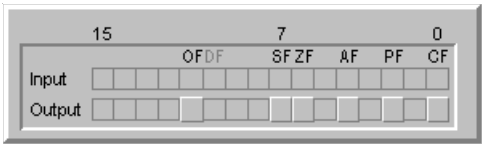
Funktion

Schreibe einen neuen Selektor für eine *local descriptor table* in das LDT-Register (»load local descriptor table«).

Operation

```
IF (Src > DescriptorTableLimit) THEN
   #GP(Src)
Descriptor := GDT[Src]
DTType := Descriptor[Type]
Present := Descriptor[AVL]
IF (DTType ≠ LDT) THEN
   #GP(Src)
IF (NOT Present) THEN
   #NP(Src)
LDTR[visible] := Src
LDTR[hidden] := Descriptor
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Der Quelloperand ist eine Speicherstelle, die einen Pointer vom Typ Selector16 besitzt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 00 /2	LLDT m		Mem48

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	3, 8, 32 15	0 selector	- -	0 selector	- -
#NP	2	selector	-	selector	-
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-
#UD	-	-	1	-	1

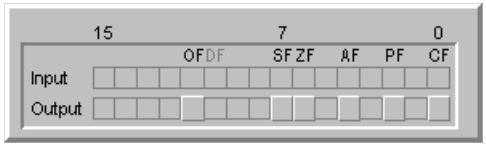
Erklärung Der Befehl ist ein absolut privilegierter Befehl, was bedeutet, dass er nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann.

SLDT 80286

Funktion Speichere den Selektor der aktuellen *local descriptor table* aus dem LDT-Register (»set local descriptor table«).

Operation
```
Dest := LDTR[visible]
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Zieloperand ist eine Speicherstelle, die einen Pointer vom Typ Selector16 besitzt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 00 /0	SLDT m	Mem16	

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	3, 8, 20	0		0	
#PF	1	fault code		fault code	
#SS	1	0		0	
#UD	-	-	1	-	1

Erklärung

Der Befehl ist zwar nicht ein absolut privilegierter Befehl wie sein schreibender Counterpart (LLDT), was bedeutet, dass er nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann. Dennoch sind sein Nutzen und seine Nutzbarkeit begrenzt.

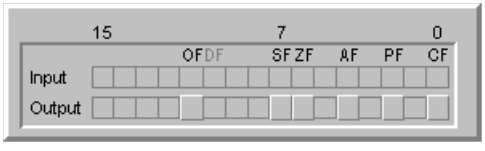
LTR 80286

Funktion

Lade einen neuen Selektor für ein *task state segment* in das *task register* (»load task register«).

Operation

```
Global := Selector[TI]
IF (NOT Global)
OR (Src > GDT-Limit) THEN
   #GP(Src)
Descriptor := GDT[Src]
DTType := Descriptor[Type]
Present := Descriptor[AVL]
IF (DTType ≠ TSS) THEN
   #GP(Src)
IF (NOT Present) THEN
   #NP(Src)
Descriptor[B] := 1
GDT[Src] := Descriptor
TR[visible] := Src
TR[hidden] := Descriptor
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Quelloperand ist eine Speicherstelle, die einen Pointer vom Typ Selector16 besitzt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 00 /3	LTR m		Mem16

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	3, 8, 32	0	-	0	-
	15, 65	selector	-	selector	-
#NP	3	selector	-	selector	-
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-
#UD	-	-	1	-	1

Erklärung Der Befehl ist ein absolut privilegierter Befehl, was bedeutet, dass er nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann.

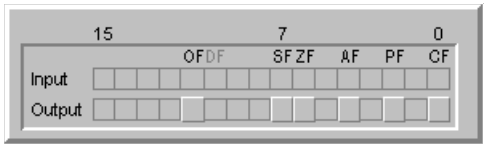
STR 80286

Funktion Hole den Selektor für das aktuelle *task state segment* aus dem *task register* (»set task register«).

Operation

```
Dest := TR[visible]
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Quelloperand ist eine Speicherstelle, die einen Pointer vom Typ Selector16 besitzt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 00 /1	STR m	Mem16	

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	3, 8, 20	0	-	0	-
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-
#UD	-	-	1	-	1

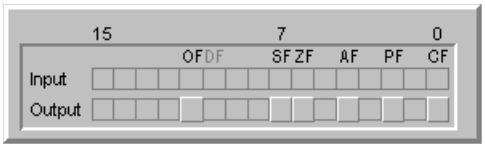
Der Befehl ist zwar nicht ein absolut privilegierter Befehl wie sein schreibender Counterpart (LTR), was bedeutet, dass er nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann. Dennoch sind sein Nutzen und seine Nutzbarkeit begrenzt. *Erklärung*

RDMSR Pentium
WRMSR Pentium

Lies ein *machine specific register* (MSR) aus oder beschreibe es (»read/write machine specific register«). *Funktion*

Operation

```
IF (Instruction = RDMSR) THEN
   EDX: EAX := MSR[ECX]
ELSE                    ; (Instruction = WRMSR)
   MSR[ECX] := EDX:EAX
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Die Quell- und Zielregister der Befehle sind impliziert. Es handelt sich in jedem Fall um Register bzw. -kombinationen. *Opcodes*

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 32	RDMSR	EDX:EAX	MSR[ECX]
	0F 30	WRMSR	MSR[ECX]	EDX:EAX

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	32, 59	0	2	0	59

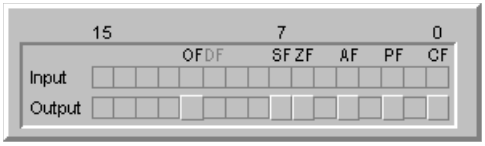
Erklärung Die Befehle sind absolut privilegierte Befehle, was bedeutet, dass sie nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden können.

RDPMC — Pentium Pro (später Pentium)

Funktion Lies ein *performance monitoring register* aus (»read performance monitoring register«).

Operation

```
IF (CPU = P6 oder Pentium mit MMX) THEN
   IF ((ECX = 0) OR (ECX = 1))
   AND ((CR4.PCE = 1) OR ((CR4.PCE = 0) AND (CPL = 0))) THEN
      EAX := PMC(ECX)[31..00]
      EDX := PMC(ECX)[39..32]
   ELSE
      #GP(0)
ELSE                    ; (CPU = Pentium 4)
   IF (ECX = < 18)
   AND ((CR4.PCE = 1) OR ((CR4.PCE = 0) AND (CPL = 0))) THEN
      IF (ECX[31] = 0 THEN
         EAX := PMC(ECX)[31..00]
         EDX := PMC(ECX)[39..32]
      ELSE
         EAX := PMC(ECX)[31..00]
         EDX := 0
   ELSE
      #GP(0)
```



Input: keiner

Output: Status-Flags werden nicht verändert.

Statusflags

Die Quell- und Zielregister der Befehle sind impliziert. Es handelt sich in jedem Fall um Register bzw. -kombinationen.

Opcodes

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 33	RDPMC	EDX:EAX	MSR[ECX]

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Exceptions

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	52, 60	0	53, 60	0	53, 60

Der Befehl ist zwar »nur« ein bedingt privilegierter Befehl, was bedeutet, dass er entweder nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann oder wenn das Betriebssystem durch Setzen eines Bits in einem Kontrollregister die Nutzung freigegeben hat. Somit besteht unter bestimmten Bedingungen durchaus die Möglichkeit, ihn sinnvoll einzusetzen. Allerdings ist seine Existenz auch prozessorabhängig.

Erklärung

RDTSC — Pentium

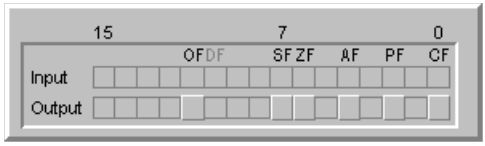
Lies das *time stamp counter register* aus (»read time stamp counter register«).

Funktion

Operation

```
IF ((CR4.TSD = 0) OR ((CR4.TSD = 1) AND (CPL = 0))) THEN
   EAX := MSR(TSC)[31..00]
   EDX := MSR(TSD)[63..32]
ELSE
   #GP(0)
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Die Quell- und Zielregister der Befehle sind impliziert. Es handelt sich in jedem Fall um Register bzw. -kombinationen.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 31	RDTSC	EDX:EAX	MSR[TSC]

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	54	0	55	0	55

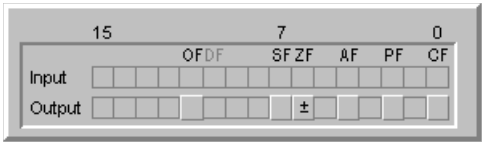
Erklärung Der Befehl ist zwar »nur« ein bedingt privilegierter Befehl, was bedeutet, dass er entweder nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann oder wenn das Betriebssystem durch Setzen eines Bits in einem Kontrollregister die Nutzung freigegeben hat. Somit besteht unter bestimmten Bedingungen durchaus die Möglichkeit, ihn sinnvoll einzusetzen. Allerdings ist seine Existenz auch prozessorabhängig.

ARPL 80286

Funktion Prüfe die RPL-Felder zweier Selektoren auf Gleichheit und passe das des einen ggf. an das des anderen an (»adjust RPL«).

Operation

```
OldRPL := Dest[RPL]
NewRPL := Src[RPL]
IF (OldRPL < NewRPL) THEN
   Dest[RPL] := NewRPL
   ZF := 1
ELSE
   ZF := 0
```



Input: keiner

Output: ZF wird anhand der Prüfung gesetzt, alle anderen Status-Flags werden nicht verändert.

Statusflags

Bei beiden Operanden handelt es sich um 16-Bit-Selektoren. Quelle kann nur ein Register sein, Ziel ein Register oder eine Speicherstelle.

Opcodes

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	63 /r	ARPL r/m, r	Reg16 / Mem16	Reg16

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Exceptions

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	-	0	-
#GP	3, 8, 20	0	-	0	-
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-
#UD	-	-	1	-	1

LAR 80286
LSL 80286

Lade die Zugriffsrechte (»load access rights«) bzw. das Segmentlimit (»load segment limit«) des durch den übergebenen Selektoren spezifizierten Segments.

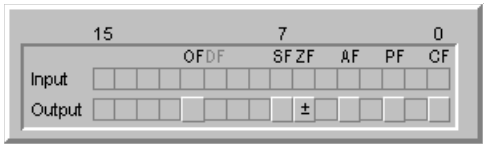
Funktion

Operation

```
IF (Src > DescriptorTableLimit) THEN
   ZF := 0
   EXIT
Descriptor := DescriptorTable[Src]
SType := Descriptor[Type]
SDPL := Descriptor[DPL]
Granu := Descriptor[G]
```

```
IF ((SType ≠ conforming code segment)
   AND ((CPL > SDPL) OR (RPL > SDPL))
OR (SType = InterruptGate, TrapGate, reserved) THEN
   ZF := 0
ELSE
   IF (Instruction = LAR) THEN
      IF (OperandSize = 32) THEN
         ; Access rights: Type, DPL, S, P, AVL, D/B, G
         Dest := Src AND $00FxFF00
      ELSE                ; (OperandSize = 16)
         ; Access rights: Type, DPL
         Dest := Src AND $FF00
   ELSE                ; (Instruction = LSL)
      Temp := Src[Limit]
      IF (Granu = 1) THEN
         Dest := SHL(Temp, 12) OR $00000FFF
      ELSE                ; (Granu = 0)
         Dest := Temp
```

Statusflags



Input: keiner

Output: ZF wird anhand der Prüfung gesetzt, alle anderen Status-Flags werden nicht verändert.

Opcodes Quelle ist ein 16-Bit-Selektor aus einem Register oder einer Speicherstelle. Ziel ist jeweils ein Register.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 02 /r	LAR r, r/m	Reg	Reg / Mem
	0F 03 /r	LSL r, r/m	Reg	Reg / Mem

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	-	0	-
#GP	3, 8	0	-	0	-
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-
#UD	-	-	1	-	1

VERR 80286
VERW 80286

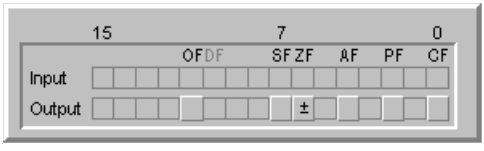
Funktion

Überprüfe das durch den übergebenen Selektoren spezifizierte Segment auf Les-/ Beschreibbarkeit (»verify read/write access«).

Operation

```
IF (Src > DescriptorTableLimit) THEN
   ZF := 0
   EXIT
Descriptor := DescriptorTable[Src]
Sys := Descriptor[S]
SType := Descriptor[Type]
SDPL := Descriptor[DPL]
IF ((SType ≠ conforming code segment)
   AND ((CPL > SDPL) OR (RPL > SDPL))
OR (Sys = 0) THEN
   ZF := 0
ELSE

   IF (Instruction = VERR) AND (SType = readable)
   OR (Instruction = VERW) AND (SType = writeable) THEN
      ZF := 1
   ELSE
      ZF := 0
```

Statusflags



Input: keiner

Output: ZF wird anhand der Prüfung gesetzt, alle anderen Status-Flags werden nicht verändert.

Opcodes Quelle ist ein 16-Bit-Selektor aus einem Register oder einer Speicherstelle. Ziel ist das zero flag im EFlags-Register.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 00 /4	VERR r/m		Reg16 / Mem16
	0F 00 /5	VERW r/m		Reg 16/ Mem16

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

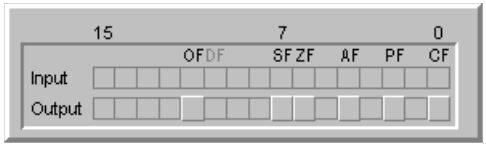
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	-	0	-
#GP	3, 8	0	-	0	-
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-
#UD	-	-	1	-	1

CLTS 80286

Funktion Lösche das Flag TS (task switch) im Kontrollregister CR0, um zu signalisieren, dass ein erfolgter task switch zur Kenntnis genommen wurde (»clear task switch flag«).

Operation
```
CR0[TS] := 0
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Der Befehl besitzt keine Operanden, er löscht lediglich ein Flag in Kontrollregister CR0.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 06	CLTS		

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	32	0	32	0	-

Erklärung

Der Befehl ist ein absolut privilegierter Befehl, was bedeutet, dass er nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann.

HLT 8086

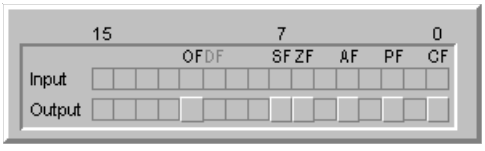
Funktion

Stelle die Programmausführung ein! Prüfe auf Interrupts (INT, NMI, SMI, BINT#, INT#) oder Resets (RESET#). Nimm in diesem Fall die Programmausführung wieder auf (»halt execution«).

Operation

```
Schalte um in HALT-Zustand
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Der Befehl besitzt keine Operanden, er löscht lediglich ein Flag in Kontrollregister CR0.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	F4	HLT		

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	32	0	32	0	-

Erklärung Der Befehl ist ein absolut privilegierter Befehl, was bedeutet, dass er nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann.

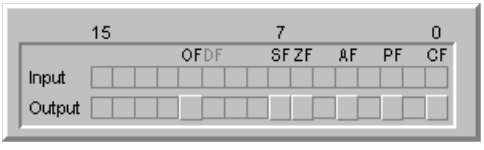
INVD 80486
WBINVD 80486

Funktion Invalidiere den gesamten Cache des Prozessors und signalisiere externen Caches, dies ebenfalls zu tun (»invalidate«, »write back and invalidate«).

Operation

```
IF (Instruction = WBINVD) THEN
  WriteBack(Interne Caches)
Flush(Interne Caches)
IF (Instruction = WBINVD) THEN
  signalisiere WriteBack(Externe Caches)
signalisiere Flush(Externe Caches)
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes INVD und WBINVD haben keine Operanden.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 08	INVD		
	0F 09	WBINVD		

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	32	0	32	0	-

Erklärung Die Befehle sind absolut privilegierte Befehle, was bedeutet, dass sie nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden können.

INVLPG 80486

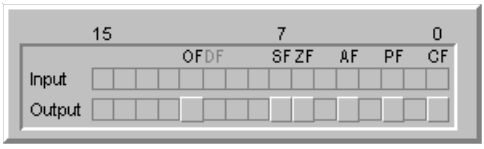
Funktion

Invalidiere (»invalidate page«) einen durch den übergebenen Zeiger spezifizierten Eintrag im translation lookaside buffer (TLB).

Operation

```
Flush(betroffene TLB-Einträge)
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

INVLPG hat keine Operanden.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 01 /7	INVLPG m		Mem

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	32	0	32	0	-

Erklärung

Der Befehl ist ein absolut privilegierter Befehl, was bedeutet, dass er nur im Rahmen von Code, der unter privileg level 0 abläuft, ausgeführt werden kann.

RSM 80486

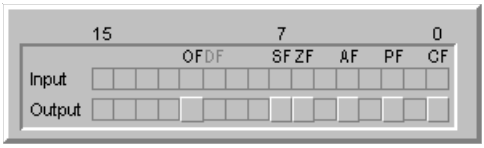
Funktion

Nimm die Programmausführung aus dem System Management Mode wieder auf (»resume from system management mode«).

Operation

```
ReturnFromSMM
ProcessorState := Restore(SMM-Dump)
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes RSM restauriert den Prozessorzustand aus einem besonderen Speicherbereich, der sich nur im System Manage Mode ansprechen lässt.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F AA	RSM		

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	32	0	32	0	-

Erklärung Der Befehl ist lediglich in Code einsetzbar, der im Rahmen des System Management Mode (SMM) ausgeführt wird.

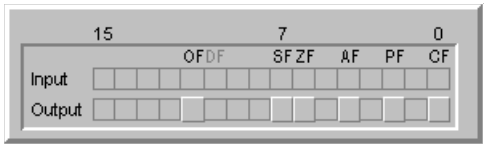
2.16 Performance-steigernde Verwaltungsbefehle

CLFLUSH SSE2

Funktion Schreibe die durch einen übergebenen Zeiger spezifizierte, veränderte (»dirty«) Zeile im Cache zurück in den Speicher und invalidiere sie im Cache (»flush cache line«).

Operation
```
FlushCacheLine(Src)
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Unabhängig von der aktuellen Umgebung erwartet CLFLUSH immer einen 8-Byte-Operanden.

Opcode	Mnemonic	Operanden	
8-Bit-Operanden		Dest	Src
0F AE /7	CLFLUSH m	Mem8	

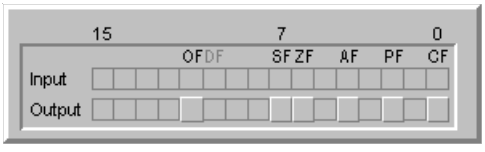
Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#GP	3, 8	0	9	0	9
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-
#UD	20	-	20	0	20

PREFETCHx SSE
PREFETCH, PREFETCHW 3D-Now!

Lade die spezifizierten Datenzeilen in den Cache (»prefetch«). *Funktion*

```
Fetch(Dest)
```
Operation



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Alle Befehle und Befehlsversionen erwarten einen 8-Bit-Speicheroperanden, der die Adresse der Zeile angibt, die in den Cache geladen werden soll. *Opcodes*

Opcode 8-Bit-Operanden	Mnemonic	Operanden Dest	Src
0F 18 /1	PREFTECH0 m8	Cache	Mem8
0F 18 /2	PREFETCH1 m8	Cache	Mem8
0F 18 /3	PREFETCH2 m8	Cache	Mem8
0F 18 /0	PREFTECHNTA m8	Cache	Mem8
0F 0D	PREFETCH(W)	Cache	Mem8

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Die Befehle erzeugen keine Exceptions. *Exceptions*

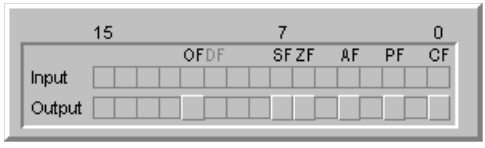
LFENCE SSE2
MFENCE SSE2
SFENCE SSE

Funktion Serialisiere Lade- und Speicheroperationen (»load/memory/store fence«).

Operation

```
IF (Instruction = LFENCE) THEN
   WAIT
   UNTIL (preceeding loads globally visible)
   LOAD(new loads)
ELSEIF (Instruction = MFENCE) THEN
   WAIT
   UNTIL (preceeding loads and stores globally visible)
   LOAD/STORE(new loads/stores)
ELSE                    ; (Instruction = SFENCE)
   WAIT
   UNTIL (preceeding stores globally visible)
   STORE(new stores)
```

Statusflags



Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes Die Befehle haben keine Operanden, sie sichern lediglich die Reihenfolge von Lade- und Speicheroperationen.

Opcode	Mnemonic	Operanden	
		Dest	Src
0F AE /5	LFENCE		
0F AE /6	MFENCE		
0F AE /7	SFENCE		

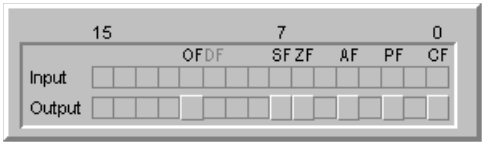
Exceptions Die Befehle erzeugen keine Exceptions.

PAUSE SSE2

Funktion Pentium-4-Performancesteigerung bei »spin-wait loops«.

Operation

```
DELAY
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

Der Befehl hat keine Operanden. Es handelt sich um einen erweiterten NOP-Befehl, wie man am vorangestellten Präfix $F3 erkennt.

Opcode	Mnemonic	Operanden	
		Dest	Src
F3] 90	PAUSE		

Exceptions

PAUSE erzeugt keine Exceptions.

2.17 Obsolete Instruktionen

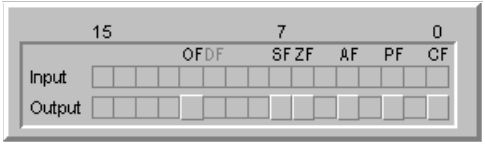
LMSW **80286**
SMSW **80286**

Funktion

Laden und Speichern des *machine status word* (Bits 15 bis 0 des Kontrollregisters CR0).

Operation

```
IF (Instruction = LMSW) THEN
   CR0[PE] := Src[0]
   CR0[MP] := Src[1]
   CR0[EM] := Src[2]
ELSE                    ; (Instruction = SMSW)
   Dest := CR0[15..00]
```



Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.

Opcodes

LMSW und SMSW haben einen 16-Bit-Operanden, der entweder eine Speicherstelle oder ein Register darstellen kann. Falls 32-Bit-Operanden verwendet werden, bleiben die oberen 16 Bit unverändert.

Opcode		Mnemonic	Operanden	
8-Bit-Operanden	32(16)-Bit-Op.		Dest	Src
	0F 01 /6	LMSW		Reg16 / Mem16
	0F 01 /4	SMSW	Reg16 / Mem16	

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

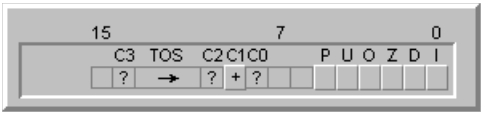
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	-	0	-
#GP	3, 8, 20	0	-	0	-
#PF	1	fault code	-	fault code	-
#SS	1	0	-	0	-

3 FPU-Instruktionen

Befehl(e) **eingeführt mit NPX / FPU**

Kurze Darstellung der Funktion des Befehls. *Funktion*

`Formale Darstellung der durchgeführten Operation in einer Pascal-ähnlichen Notation.` *Operation*



Statusflags

Stellung der Flags des condition codes (C3 bis C0) und der exception flags sowie des Einflusses des Befehls auf den stack pointer. Hinweis: Die Flags des Statuswords sind »sticky«, das heißt, ihr Zustand bleibt erhalten, wenn ein Befehl das Flag nicht betrifft. Dem wird in der Abbildung dadurch Rechnung getragen, dass nicht betroffene exception flags erhaben dargestellt werden und frei bleiben, also – je nach Historie, gesetzt oder gelöscht sein können.

Opcodes

Anders als bei CPU-Befehlen gibt es bei FPU-Opcodes keine 8-Byte-Operanden, weshalb auf die Unterscheidung in den Tabellen verzichtet wird:

Opcode	Mnemonic	Operanden	
		Dest	Src
D8 /0	FADD m32	ST(0)	Mem32

In der Tabelle bedeuten in der Opcode-Spalte

/r Der Opcode besitzt ein *ModR/M*-Byte, da als Operanden sowohl ein Register als auch ein Register bzw. eine Speicherstelle angegeben werden müssen, die mittels der Felder *reg* und *r/m* kodiert werden.

/0 ... /7 Der Opcode besitzt ein *ModR/M*-Byte mit modifizierter Bedeutung: Das *reg*-Feld im *ModR/M*-Byte kann nur den spezifizierten Wert annehmen (z.B. 010_b bei /2) und dient der Erwei-

terung des Opcodes. Zur Kodierung von Operanden kann somit nur das *r/m*-Feld herangezogen werden. Das bedeutet entweder, dass es nur einen Operanden gibt, oder dass ein evtl. möglicher zweiter Operand nur eine Konstante sein kann, die nicht via *reg* kodiert werden muss.

+r Zu dem Byte des Opcodes wird ein Wert zwischen 0 und 7 addiert, der das Stack-Register der FPU kodiert, das im Befehl eine Rolle spielt:

			r =				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)

Ferner bedeuten in der Mnemonic-Spalte:

cX Konstante der Operandengröße X.

mX Speicherstelle der Operandengröße X, die ein Datum aufnimmt.

r FPU-Stack-Register.

Schließlich bedeuten in der Operanden-Spalte analog zur Mnemonic-Spalte:

ConstX Konstante der Operandengröße x.

MemX Speicherstelle der Operandengröße X, die ein Datum aufnimmt. In der Befehlssequenz stellt Mem eine effektive Adresse, also einen Offset dar, der als Konstante hinter den Opcode geschrieben wird. Die Größe dieses Offsets ist abhängig vom aktuellen Betriebsmodus, der aktuellen Umgebung und dem address size override prefix. Sie kann 16 oder 32 Bit betragen, sodass dem Opcode je nach Situation zwei oder vier Adressenbytes folgen. Falls sich der Offset auf ein anderes Segment bezieht, kann der Einsatz eines segment override prefix erforderlich werden.

Reg FPU-Stack-Register. Als Angaben sind ST(0) bis ST(7) möglich.

Exceptions Die bei dem entsprechenden Befehl möglichen Gründe werden tabellarisch in Abhängigkeit des Betriebsmodus des Prozessors angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#MF	1	-	1	-	1
Exception-Flags	#D	#IA #IS	#O	#P #U	#Z

Bitte beachten Sie, dass der Grund für die Auslösung einer #MF mindestens ein gesetztes, unmaskiertes FPU-Exception-Flag beim Eintritt in den Befehl ist! Die Exception-Flags für den aktuellen Befehl werden erst dann gesetzt, wenn die anhängige Exception behandelt wurde.

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Bemerkungen

Hier werden verschiedene Bemerkungen oder Hinweise gegeben. Dieser Abschnitt ist optional und nicht bei allen Befehlen vorhanden.

FADD, FADDP **8087**
FIADD **8087**

FSUB, FSUBP, FSUBR, FSUBRP **8087**
FISUB, FISUBR **8087**

FMUL, FMULP **8087**
FIMUL **8087**

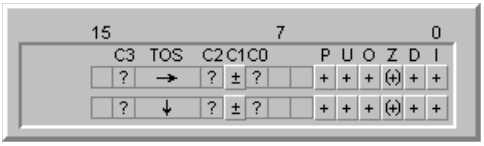
FDIV, FDIVP, FIDVR, FDIVRP **8087**
FIDIV, FIDIVR **8087**

Funktion

Addiere, subtrahiere, multipliziere oder dividiere eine Realzahl oder Integer mit/zu/durch eine Realzahl (bzw. Integer) mit oder ohne POPpen des Stack und unter Berücksichtigung der Kommutativität, wo erforderlich (»floating-point addition/subtraction/multiplication/division [use integer] [pop fpu stack] [reverse]«).

Operation

```
IF (Instruction = FADD, FADDP) THEN
   Dest := Dest + Src
   IF (Instruction = FADDP) THEN
      PopStack
ELSEIF (Instruction = FIADD) THEN
   Dest := Dest + Convert2Extended(Src)
ELSEIF (Instruction = FSUB, FSUBP) THEN
   Dest := Dest – Src
   IF (Instruction = FSUBP) THEN
      PopStack
ELSEIF (Instruction = FISUB) THEN
   Dest := Dest - Convert2Extended(Src)
ELSEIF (Instruction = FSUBR, FSUBRP) THEN
   Dest := Src - Dest
   IF (Instruction = FSUBRP) THEN
      PopStack
ELSEIF (Instruction = FISUBR) THEN
   Dest := Convert2Extended(Src) - Dest
ELSEIF (Instruction = FMUL, FMULP) THEN
   Dest := Dest * Src
   IF (Instruction = FMULP) THEN
      PopStack
ELSEIF (Instruction = FIMUL) THEN
   Dest := Dest * Convert2Extended(Src)
ELSEIF (Instruction = FDIV, FDIVP) THEN
   IF Src = 0 THEN
      #Z
   Dest := Dest / Src
   IF (Instruction = FDIVP) THEN
      PopStack
ELSEIF (Instruction = FIDIV) THEN
   IF Src = 0 THEN
      #Z
   Dest := Dest / Convert2Extended(Src)
ELSEIF (Instruction = FDIVR, FDIVRP) THEN
   IF Dest = 0 THEN
      #Z
   Dest := Src / Dest
   IF (Instruction = FDIVRP) THEN
      PopStack
ELSE                    ; (Instruction = FIDIVR) THEN
   IF Dest = 0 THEN
      #Z
   Dest := Convert2Extended(Src) / Dest
```



Statusflags

C3, C2 und C0 sind undefiniert, C1 nur bei #I und #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. Die Befehle verhalten sich stackneutral, es sei denn, es handelt sich um die POP-Versionen (»P«-Suffixe), die einen stack pop (Inkrementieren des stack pointer) durchführen. Eine #Z ist nur bei FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, FIDIVR möglich, ansonsten können alle FPU exceptions ausgelöst werden.

Opcodes

Bei allen arithmetischen Instruktionen sind zwei Quelloperanden erforderlich, von denen jeweils einer implizit vorgegeben ist: der TOS. Bei den Instruktionen mit Speicherstelle als zweitem Operanden ist dieser erste Operand immer impliziert und darf nicht im Mnemonic angegeben werden, bei denen mit zwei Registeroperanden müssen beide Operanden explizit angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D8 /0	FADD m32	ST(0)	Mem32
DC /0	FADD m64	ST(0)	Mem64
DA /0	FIADD m32	ST(0)	Mem32
DE /0	FIADD m16	ST(0)	Mem16
D8 (C0+r)	FADD ST(0), ST(r)	ST(0)	ST(r)
DC (C0+r)	FADD ST(r), ST(0)	ST(r)	ST(0)
DE (C0+r)	FADDP ST(r), ST(0)	ST(r)	ST(0)
DE C1	FADDP	ST(1)	ST(0)
D8 /4	FSUB m32	ST(0)	Mem32
D8 /5	FSUBR m32	ST(0)	Mem32
DC /4	FSUB m64	ST(0)	Mem64
DC /5	FSUBR m64	ST(0)	Mem64
DA /4	FISUB m32	ST(0)	Mem32
DA /5	FISUBR m32	ST(0)	Mem32
DE /4	FISUB m16	ST(0)	Mem64
DE /5	FISUBR m16	ST(0)	Mem64
D8 (E0+r)	FSUB ST(0), ST(r)	ST(0)	ST(r)

Opcode	Mnemonic	Operanden	
		Dest	Src
D8 (E8+r)	FSUBR ST(0), ST(r)	ST(0)	ST(r)
DC (E8+r)	FSUB ST(r), ST(0)	ST(r)	ST(0)
DC (E0+r)	FSUBR ST(r), ST(0)	ST(r)	ST(0)
DE (E0+r)	FSUBP ST(r), ST(0)	ST(r)	ST(0)
DE (E8+r)	FSUBRP ST(r), ST(0)	ST(r)	ST(0)
DE E9	FSUBP	ST(1)	ST(0)
DE E1	FSUBRP	ST(1)	ST(0)
D8 /1	FMUL m32	ST(0)	Mem32
DC /1	FMUL m64	ST(0)	Mem64
DA /1	FIMUL m32	ST(0)	Mem32
DE /1	FIMUL m16	ST(0)	Mem16
D8 (C8+r)	FMUL ST(0), ST(r)	ST(0)	ST(r)
DC (C8+r)	FMUL ST(r), ST(0)	ST(r)	ST(0)
DE (C8+r)	FMULP ST(r), ST(0)	ST(r)	ST(0)
DE C9	FMULP	ST(1)	ST(0)
D8 /6	FDIV m32	ST(0)	Mem32
D8 /7	FDIVR m32	ST(0)	Mem32
DC /6	FDIV m64	ST(0)	Mem64
DC /7	FDIVR m64	ST(0)	Mem64
DA /6	FIDIV m32	ST(0)	Mem32
DA /7	FIDIVR m32	ST(0)	Mem32
DE /6	FIDIV m64	ST(0)	Mem64
DE /7	FIDIVR m16	ST(0)	Mem16
D8 (F0+r)	FDIV ST(0), ST(r)	ST(0)	ST(r)
D8 (F8+r)	FDIVR ST(0), ST(r)	ST(0)	ST(r)
DC (F8+r)	FDIV ST(r), ST(0)	ST(r)	ST(0)
DC (F0+r)	FDIVR ST(r), ST(0)	ST(r)	ST(0)
DE (F0+r)	FDIVP ST(r), ST(0)	ST(r)	ST(0)
DE (F8+r)	FDIVRP ST(r), ST(0)	ST(r)	ST(0)
DE F9	FDIVP	ST(1)	ST(0)
DE F1	FDIVRP	ST(1)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und teilweise angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1, 7, 8, 13	1	1	1	1	(1)

Die geklammerten Gründe kommen nur bei den Divisionsbefehlen vor. Nicht jeder Grund bei der FPU exception #IA kommt bei jedem Befehl vor.

FABS 8087
FCHS 8087

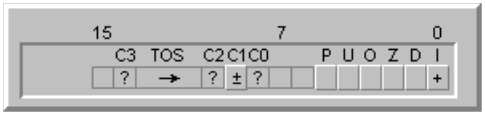
Funktion

Bilde den Absolutwert (»floating-point absolute«) einer Realzahl oder wechsle das Vorzeichen (»floating-point change sign«).

Operation

```
IF (Instruction = FABS) THEN
   ST(0)[MSB] := 0
ELSE                    ; (Instruction = FCHS)
   ST(0)[MSB] := NOT(ST(0)[MSB])
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert und zeigt dann im gesetzten Zustand einen stack underflow an. FABS und FCHS sind stackneutral. Es kann bei diesen Befehlen nur eine #I auftreten.

Opcodes Bei beiden Befehlen sind die Operanden impliziert und dürfen nicht im Mnemonic angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 E1	FABS	ST(0)	ST(0)
D9 E0	FCHS	ST(0)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

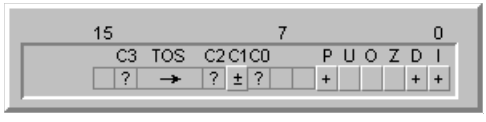
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1
Exception-Flags	#D	#IA #IS	#O	#P #U	#Z
		1			

FSQRT 8087

Funktion Bilde die Quadratwurzel einer Realzahl (»floating-point square root«).

Operation
```
ST(0) := SQRT(ST(0))
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 nur bei #I oder #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. FSQRT ist stackneutral, an exceptions kommen nur #P, #D und #I in Frage.

Opcodes FSQRT impliziert die beiden Operatoren. Sie dürfen nicht im Mnemonic angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 FA	FSQRT	ST(0)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1, 3	1	-	1	-	-

FPREM 8087
FPREM1 80387

Funktion

Reduziere eine Zahl (»floating-point partial remainder«): bilde den Divisionsrest.

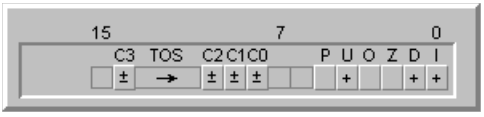
Operation

```
D := Exponent(ST(0)) – Exponent(ST(1))
IF D < 64 THEN
   IF (Instruction = FPREM) THEN
      Quotient := Truncate2Zero(ST(0) / ST(1))
   ELSE                ; (Instruction = FPREM1)
      Quotient := Round2Nearest(ST(0) / ST(1))
   ST(0) := ST(0) – (ST(1) * Quotient)
   C2 := 0
   C0 := Quotient[2]
   C3 := Quotient[1]
   C1 := Quotient[0]
ELSE                   ; (D ≥ 64)
   C2 := 1
```

```
N := Constant       ; implementationsabhängig: 32≤ N ≤63
Quotient := Truncate2Zero((ST(0) / ST(1)) / 2^(D-N))
ST(0) := ST(0) – (ST(1) * Quotient * 2^(D-N))
```

Hinweis: Der hier vorgestellte Algorithmus ist der von Intel dokumentierte.

Statusflags



C0, C3 und C1 bilden (in dieser Reihenfolge!) die Bits 2, 1 und 0 des Quotienten der Division, wenn keine #I erfolgte; andernfalls zeigt es einen stack underflow an. C2 zeigt an, ob die Reduktion vollständig durchgeführt werden konnte. Die Befehle sind stackneutral, an Exceptions kommen #U, #D und #I in Betracht.

Opcodes

FPREM und FPREM1 erwarten zwei Quelloperanden und einen Zieloperanden, die jedoch alle impliziert sind und nicht extra angegeben werden dürfen.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 F8	FPREM	ST(0)	ST(0), ST(1)
D9 F5	FPREM1	ST(0)	ST(0), ST(1)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode			Virtual 8086 Mode			Real Mode
	Grund	ErrorCode		Grund	ErrorCode		Grund
#NM	2	-		2	-		2
#MF	1	-		1	-		1
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	11	1	-	-	1	-

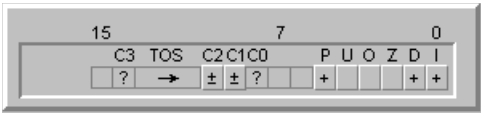
FSIN 8087
FCOS 8087

Bilde den Sinus bzw. Cosinus der Realzahl (»floating-point sine/cosine«). *Funktion*

Operation

```
IF (ST(0) < 2^63 THEN
   C2 := 0
   IF (Instruction = FSIN) THEN
      ST(0) := Sin(ST(0))
   ELSE                ; (Instruction = FCOS)
      ST(0) := Cos(ST(0))
```

Statusflags



C3 und C0 sind undefiniert, C1 nur bei #I und #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. C2 ist gesetzt, wenn der Operand außerhalb des Wertebereiches -2^{63} bis $+2^{63}$ liegt, andernfalls gelöscht. FSIN und FCOS verhalten sich stackneutral. An Exceptions sind nur #P, #D und #I möglich.

Opcodes

FSIN und FCOS implizieren Quell- und Zieloperanden. Sie dürfen nicht im Mnemonic angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 FE	FSIN	ST(0)	ST(0)
D9 FF	FCOS	ST(0)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode			Virtual 8086 Mode			Real Mode
	Grund	ErrorCode		Grund	ErrorCode		Grund
#NM	2	-		2	-		2
#MF	1	-		1	-		1
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1,5	1	-	1	-	-

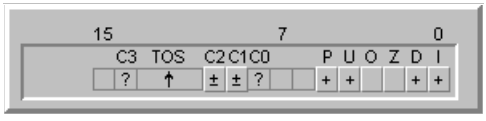
FSINCOS 80387
FPTAN 8087

Funktion Bilde den Sinus und Cosinus bzw. den Tangens der Realzahl (»floating-point sine and cosine / partial tangens«).

Operation

```
IF (ST(0) < 2^63 THEN
   C2 := 0
   IF (Instruction = FSINCOS) THEN
      Temp := Cos(ST(0))
      ST(0) := Sin(ST(0))
   ELSE                ; (Instruction = FPTAN)
      Temp := 1.0
      ST(0) := Tan(ST(0))
   TOS := (TOS + 7) MOD 8 ; (≡ zyklisches Dekrementieren)
   ST(0) := Temp       ; neuer TOS vorher ST(1)!
```

Statusflags



C3 und C0 sind undefiniert, C1 nur bei #I und #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. C2 ist gesetzt, wenn der Operand außerhalb des Wertebereiches -2^{63} bis $+2^{63}$ liegt, andernfalls gelöscht. FSINCOS und FPTAN führen einen stack push (Dekrementieren des TOS-Feldes) durch. An Exceptions sind nur #P, #U, #D und #I möglich.

Opcodes FSINCOS und FPTAN implizieren Quell- und Zieloperanden. Sie dürfen nicht im Mnemonic angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 FB	FSINCOS	ST(0), ST(1)	ST(0)
D9 F2	FPTAN	ST(0), ST(1)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode		
	Grund	ErrorCode	Grund	ErrorCode	Grund		
#NM	2	-	2	-	2		
#MF	1	-	1	-	1		
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1,5	1	-	1	1	-

FPATAN — 8087

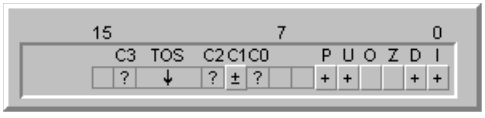
Funktion

Bilde des Arcustangens der Realzahl (»floating-point partial arc tangens«).

Operation

```
ST(1) := ArcTan(ST(1) / ST(0))
PopFPUStack             ; ≡ FFREE ST(0) - FDECSTP
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 nur bei #I und #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. FPATAN führt einen stack pop (Inkrementieren des TOS-Feldes) durch. An Exceptions sind nur #P, #U, #D und #I möglich.

Opcodes FPATAN erwartet beide Quelloperanden und den Zieloperanden implizit, sie dürfen im Mnemonic nicht angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 F3	FPATAN	ST(0)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

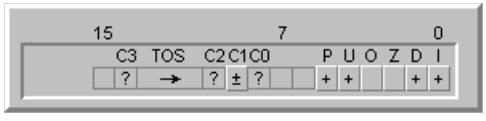
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1	1	-	1	1	-

F2XM1 8087

Funktion Erhebe die Realzahl in die Zweite Potenz und subtrahiere 1 (»floating-point two power x minus one«).

Operation $ST(0) := 2^{ST(0)} - 1$

Statusflags



C3, C2 und C0 sind undefiniert, C1 nur bei #I und #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. F2XM1 verhält sich stackneutral. Alle Exceptiontypen außer #O und #Z sind möglich.

Opcodes Wie die meisten arithmetischen Befehle sind Quell- und Zieloperand von F2XM1 impliziert und dürfen nicht im Mnemonic angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 F0	F2XM1	ST(0)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1	1	-	1	1	-

FYL2X 8087
FYL2XP1 8087

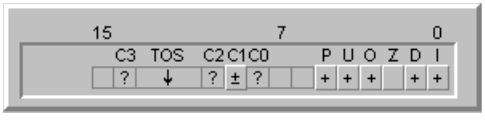
Funktion

Bilde den Logarithmus Dualis (Logarithmus zur Basis 2) der Realzahl, die ggf. um 1 erhöht wurde (»floating-point y times $\log_2$ of (x [plus 1])«).

Operation

```
IF (Instruction = FYL2X) THEN
   Temp := ST(0)
ELSE                        ; (Instruction = FYL2XP1)
   Temp := ST(0) + 1
ST(1) := ST(1) * LD(Temp)   ; LD = logarithmus dualis = Log2
PopFPUStack                 ; ≡ FFREE ST(0) - FDECSTP
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 nur bei #I und #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. FYL2X und FYL2XP1 führen einen stack pop (Inkrementieren des TOS-Feldes) durch. Alle Exceptiontypen außer #Z sind möglich.

Opcodes Wie die meisten arithmetischen Befehle sind Quell- und Zieloperand impliziert und dürfen nicht im Mnemonic angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 F1	FYL2X	ST(1)	ST(0), ST(1)
D9 F9	FYL2XP1	ST(1)	ST(0), ST(1)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	(3,) 10	1	1	1	1	(3)

#Z und Grund 3 bei #IA können nur bei FYL2X, nicht aber bei FYL2XP1 auftreten!

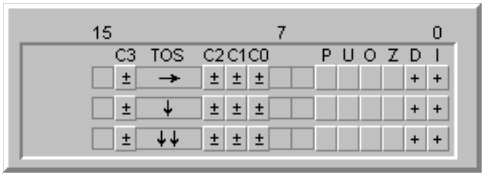
FCOM, FCOMP, FCOMPP 8087
FICOM, FICOMP 8087
FUCOM, FUCOMP, FUCOMPP 8087
FTST 8087

Funktion Vergleiche zwei Realzahlen, eine Realzahl und eine Integer oder eine Realzahl und 0.0 miteinander, ggf. mit oder ohne (zweifachem) POPpen des Stack und ggf. mit/ohne Berücksichtigung von qNaNs (»floating point test«, »floating-point [unordered] compare [with integer] [pop fpu stack [twice]]«).

Operation

```
IF (Instruction = FICOM, FICOMP, FICOMPP) THEN
   Temp := Convert2Extended(Src)
ELSEIF (Instruction = FTST) THEN
   Temp := 0.0
ELSE
   Temp := Src
IF (ST(0) > Temp) THEN
   C3 := 0
   C2 := 0
   C0 := 0
ELSEIF (ST(0) < Temp) THEN
   C3 := 0
   C2 := 0
   C0 := 1
ELSE                              ; (ST(0) = Temp)
   C3 := 1
   C2 := 0
   C0 := 0
IF ((Instruction = FCOM, FCOMP, FCOMPP)
   AND (TOS = sNaN,qNaN,Pseudo) OR (Src = sNaN,qNaN,Pseudo))
OR ((Instruction = FUCOM, FUCOMP, FUCOMPP)
   AND (TOS = sNaN,Pseudo) OR (Src = sNaN,Pseudo))
OR ((Instruction = FTST) AND (TOS = sNaN,qNaN,Pseudo) THEN
   IF ControlWord[I] = 1 THEN  ; Exception maskiert
      C3 := 1
      C2 := 1
      C0 := 1
   ELSE                           ; Exception unmaskiert
      #IA                         ; Flags nicht gesetzt!
IF (Instruction = FCOMP, FICOMP, FUCOMP) THEN
   PopFPUStack                    ; ≡ FFREE ST(0) – FDECSTP
IF (Instruction = FCOMPP, FUCOMPP) THEN
   PopFPUStack                    ; ≡ FFREE ST(0) – FDECSTP
   PopFPUStack                    ; ≡ FFREE ST(0) – FDECSTP
```

Statusflags



C3, C2 und C0 geben das Ergebnis des Vergleichs gemäß der folgenden Tabelle zurück. ACHTUNG: Wird eine #IA ausgelöst (nicht maskierte Exception), so werden C3, C2 und C0 nicht gesetzt! C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Die Befehle verhalten sich stackneutral, es sei denn, es handelt sich um die POP-

(»P«-Suffixe) bzw. Doppel-POP-Versionen (»PP«-Suffixe), die einen einfachen oder doppelten stack pop (Inkrementieren des stack pointer) durchführen. Die Befehle können lediglich eine #D oder #I auslösen.

Bedingung	C3	C2	C0
ST(0) > Quelloperand	0	0	0
ST(0) < Quelloperand	0	0	1
ST(0) = Quelloperand	1	0	0
»unordered«, nicht vergleichbar *)	1	1	1

*) Bei den FCOMx-Befehlen und bei FTST werden diese Flags nicht gesetzt, sondern stattdessen eine #IA ausgelöst, wenn einer der Operanden eine NaN (sNaN *oder* qNaN) oder ein nicht unterstütztes Format ist. Bei FUCOMx erfolgt das nur, wenn eine sNaN oder ein nicht unterstütztes Format verwendet wird.

Opcodes Bei einigen Befehlen sind die Operanden impliziert, sie dürfen dann nicht im Mnemonic angegeben werden. Bei den restlichen ist immerhin der erste Operand impliziert, der dann auch nicht angegeben werden darf. ACHTUNG: FCOMIx und FUCOMIx zeigen hier unterschiedliches Verhalten!

Opcode	Mnemonic	Operanden	
		Dest	Src
D8 /2	FCOM m32	ST(0)	Mem32
DC /2	FCOM m64	ST(0)	Mem64
DE /2	FICOM m16	ST(0)	Mem16
DA /2	FICOM m32	ST(0)	Mem32
D8 (D0+r)	FCOM ST(r)	ST(0)	ST(r)
D8 D1	FCOM	ST(0)	ST(1)
D8 /3	FCOMP m32	ST(0)	Mem32
DC /3	FCOMP m64	ST(0)	Mem64
DE /3	FICOMP m16	ST(0)	Mem16
DA /3	FICOMP m32	ST(0)	Mem32
D8 (D8+r)	FCOMP ST(r)	ST(0)	ST(r)
D8 D9	FCOMP	ST(0)	ST(1)
DE D9	FCOMPP	ST(0)	ST(1)
DD (E0+r)	FUCOM ST(r)	ST(0)	ST(r)
DD E1	FUCOM	ST(0)	ST(1)

Opcode	Mnemonic	Operanden	
		Dest	Src
DD (E8+r)	FUCOMP ST(r)	ST(0)	ST(r)
DD E9	FUCOMP	ST(0)	ST(1)
DA E9	FUCOMPP	ST(0)	ST(1)
D9 E4	FTST	ST(0)	0.0

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	(9,) (10), 12	1	-	-	-	-

Grund 9 bei #IA tritt nur bei F(I)COMxx auf, Grund 10 bei FUCOMxx und FTST. Grund 12 kann nur bei Befehlen mit explizit anzugebendem FPU-Register auftreten. Die grau unterlegten Exceptions können nur bei den Mnemonics ausgelöst werden, die einen Speicheroperanden besitzen.

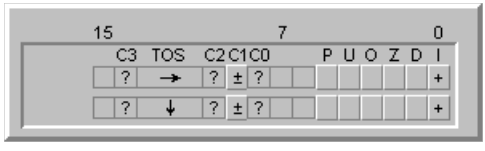
FCOMI, FCOMIP — Pentium Pro
FUCOMI, FUCOMIP — Pentium Pro

Funktion Vergleiche zwei Realzahlen gleicher oder unterschiedlicher Ordnung miteinander mit oder ohne POPpen des Stack und mit/ohne Berücksichtigung von qNaNs und signalisiere das Ergebnis in den CPU-Statusflags (»floating-point [unordered] compare [pop fpu stack] and set integer flags«).

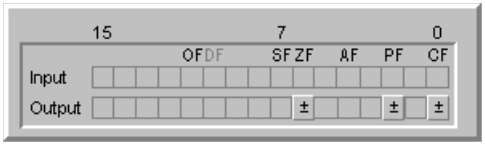
Operation

```
IF (ST(0) > Src) THEN
   ZF := 0
   PF := 0
   CF := 0
ELSEIF (ST(0) < Src) THEN
   ZF := 0
   PF := 0
   CF := 1
ELSE                              ; (ST(0) = Src)
   ZF := 1
   PF := 0
   CF := 0
IF ((Instruction = FCOMI, FCOMIP)
   AND (TOS = sNaN,qNaN,Pseudo) OR (Src = sNaN,qNaN,Pseudo))
OR ((Instruction = FUCOMI, FUCOMIP)
   AND (TOS = sNaN,Pseudo) OR (Src = sNaN,Pseudo)) THEN
   IF ControlWord[I] = 1 THEN  ; Exception maskiert
      ZF := 1
      PF := 1
      CF := 1
   ELSE                           ; Exception unmaskiert
      #IA                         ; Flags nicht gesetzt!
IF (Instruction = FCOMIP, FUCOMIP) THEN
   PopFPUStack                    ; ≡ FFREE ST(0) – FDECSTP
```

FPU-Statusflags



C3, C2 und C0 sind undefiniert, das Ergebnis des Vergleichs wird in den CPU-Statusflags gespeichert! C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Die Befehle verhalten sich stackneutral, es sei denn, es handelt sich um die POP-Versionen (»P«-Suffixe), die einen einfachen stack pop (Inkrementieren des stack pointer) durchführen. Die Befehle können lediglich eine #I auslösen.



CPU-Statusflags

Die Ergebnisse des Vergleichs werden bei diesen Befehlen in den Statusflags der CPU zurückgegeben! Hierbei werden trotz der Verwendung vorzeichenbehafteter Realzahlen in den FPU-Registern die Flags für vorzeichenlose Integer verwendet (also CF anstelle von OF und SF!)

Bedingung	ZF	PF	CF
ST(0) > Quelloperand	0	0	0
ST(0) < Quelloperand	0	0	1
ST(0) = Quelloperand	1	0	0
»unordered«, nicht vergleichbar *)	1	1	1

*) Bei den FCOMIx-Befehlen werden diese Flags nicht gesetzt, sondern stattdessen eine #IA ausgelöst, wenn einer der Operanden eine NaN (sNaN *oder* qNaN) oder ein nicht unterstütztes Format ist. Bei FUCOMIx erfolgt das nur, wenn eine sNaN oder ein nicht unterstütztes Format verwendet wird.

Opcodes

Bei den Befehlen dieser Gruppe ist der erste Operand impliziert, muss jedoch im Mnemonic angegeben werden! ACHTUNG: Dies ist ein unterschiedliches Verhalten verglichen mit FCOMx, FUCOMx und FICOMx!

Opcode	Mnemonic	Operanden	
		Dest	Src
`DB (F0+r)`	FCOMI ST(0), ST(r)	ST(0)	ST(r)
`DF (F0+r)`	FCOMIP ST(0). ST(r)	ST(0)	ST(r)
`DB (E8+r)`	FUCOMI ST(0), ST(r)	ST(0)	ST(r)
`DF (E8+r)`	FUCOMIP ST(0), ST(r)	ST(0)	ST(r)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	(9,) (10)	1	-	-	-	-

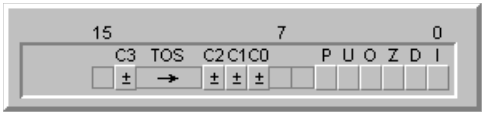
Grund 9 bei #IA tritt nur bei FCOMIx auf, Grund 10 bei FUCOMIx.

FXAM 8087

Funktion Untersuche die Realzahl (»floating-point examine«) und ordne sie einer der der FPU bekannten Datenart zu.

Operation

```
C1 := ST(0)[MSB]
IF (Type(ST(0)) = Pseudo) THEN
   C3 := 0
   C2 := 0
   C0 := 0
ELSEIF (Type(ST(0)) = NaN) THEN
   C3 := 0
   C2 := 0
   C0 := 1
ELSEIF (Type(ST(0)) = Normal) THEN
   C3 := 0
   C2 := 1
   C0 := 0
ELSEIF (Type(ST(0)) = Infinity) THEN
   C3 := 0
   C2 := 1
   C0 := 1
ELSEIF (Type(ST(0)) = Zero) THEN
   C3 := 1
   C2 := 0
   C0 := 0
IF (Type(TOS) = Empty) THEN
   C3 := 1
   C2 := 0
   C0 := 1
ELSE                    ; (Type(ST(0)) = Denormal)
   C3 := 1
   C2 := 1
   C0 := 0
```



Statusflags

C3, C2 und C0 geben den Typ des Registerinhalts gemäß der folgenden Tabelle an, C1 dessen Vorzeichen. FXAM verhält sich stackneutral. Es werden keine FPU-Exceptions ausgelöst.

Typ	C3	C2	C0
nicht unterstützt (z.B. Pseudo-Zahlen)	0	0	0
NaN (sNaNs und qNaNs)	0	0	1
normale finite Zahl (»echte« Realzahlen)	0	1	0
normale infinite Zahl (±∞)	0	1	1
Null	1	0	0
Register leer	1	0	1
denormale Zahl (gelöschtes J-Bit)	1	1	0
reserviert	1	1	1

Opcodes

Bei FXAM ist der Operand impliziert und darf nicht im Mnemonic angegeben werden!

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 E5	FXAM	ST(0)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Der Befehl löst keine FPU-Exceptions aus.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	1	-	-	-	-

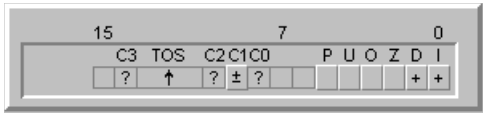
FLD 8087
FILD 8087
FBLD 8087

Funktion Lade eine Realzahl, eine Integer oder eine BCD und konvertiere sie in das ExtendedReal-Format (»floating-point load [integer/BCD]«).

Operation

```
IF (Scr = Register) THEN
   Temp := ST(I)
ELSE                    ; (Src = Memory)
   Temp := Convert2Extended(Src)
TOS := (TOS + 7) MOD 8 ; (≡ zyklisches Dekrementieren)
ST(0) := Temp
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Die Ladebefehle führen naturgemäß einen stack push (Dekrementieren des stack pointer) durch. Es können nur #D und #I ausgelöst werden.

Opcodes Bei den Befehlen dieser Gruppe ist der erste Operand impliziert und darf nicht im Mnemonic angegeben werden!

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 /0	FLD m32	ST(0)	Mem32
DD /0	FLD m64	ST(0)	Mem64
DB /5	FLD m80	ST(0)	Mem80
D9 (C0+r)	FLD ST(r)	ST(0)	ST(r)
DF /0	FILD m16	ST(0)	Mem16
DB /0	FILD m32	ST(0)	Mem32
DF /5	FILD m64	ST(0)	Mem64
DF /4	FBLD m80	ST(0)	Mem80

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebe-

nen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	(1)	(1)	2	-	-	-	-

#D und #IA sind nur bei FLD möglich.

FLD1, FLDZ — 8087
FLDPI — 8087
FLDL2T, FLDL2E — 8087
FLDLG2, FLDLN2 — 8087

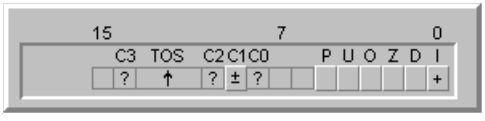
Funktion

Lade eine Konstante (»floating-point load 1.0 / 0.0 / π / ld(10) / ld(e) / lg(2) / ln(2)«).

Operation

```
TOS := (TOS + 7) MOD 8 ; (≡ zyklisches Dekrementieren)
ST(0) := Konstante
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Die Ladebefehle führen naturgemäß einen stack push (Dekrementieren des stack pointer) durch. Es kann nur eine #I ausgelöst werden.

Opcodes

Bei den Befehlen dieser Gruppe sind die Operanden impliziert und können nicht im Mnemonic angegeben werden!

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 E8	FLD1	ST(0)	$3FFF_8000_0000_0000_0000
D9 E9	FLDL2T	ST(0)	$4000_D49A_784B_CD1B_8AFE
D9 EA	FLDL2E	ST(0)	$3FFF_B8AA_3B29_5C17_F0BC
D9 EB	FLDPI	ST(0)	$4000_C90F_DAA2_2168_C235
D9 EC	FLDLG2	ST(0)	$3FFE_B172_17F7_D1CF_79AC
D9 ED	FLDLN2	ST(0)	$3FFD_9A20_9A84_FBCF_F799
D9 EE	FLDZ	ST(0)	$0000_0000_0000_0000_0000

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

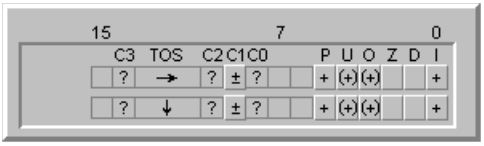
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	2	-	-	-	-

FST, FSTP 8087
FIST, FISTP 8087
FBSTP 8087

Funktion Konvertiere eine Realzahl in das Zielformat als Realzahl, Integer oder BCD und speichere sie ggf. mit POPpen des FPU-Stack (»floating-point store [as integer/BCD] [pop fpu stack]«).

Operation
```
IF (Dest = Register) THEN
   Dest := ST(0)
ELSE                    ; (Dest = Memory)
   Dest := Convert2Format(ST(0))
```

```
IF (Instruction = FSTP, FISTP, FBSTP) THEN
   PopFPUStack          ; ≡ FFREE ST(0) – FDECSTP
```



Statusflags

C3, C2 und C0 sind undefiniert, C1 nur bei #I und #P definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. Bei #P zeigt ein gesetztes C1 eine erfolgte Aufrundung an, ansonsten Abrundung. Die Befehle verhalten sich stackneutral, es sei denn, es handelt sich um die POP-Versionen (»P«-Suffixe), die einen stack pop (Inkrementieren des stack pointer) durchführen. Außer #Z und #D können bei FST(P) alle Exceptions auftreten, underflow #U und overflow exception #O sind bei den Integers naturgemäß nicht möglich.

Opcodes

Bei den Befehlen dieser Gruppe ist der zweite Operand impliziert und darf nicht im Mnemonic angegeben werden!

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 /2	FST m32	Mem32	ST(0)
DD /2	FST m64	Mem64	ST(0)
DD (D0+r)	FST ST(r)	ST(r)	ST(0)
D9 /3	FSTP m32	Mem32	ST(0)
DD /3	FSTP m64	Mem64	ST(0)
DB /7	FSTP m80	Mem80	ST(0)
DD (D8+r)	FSTP ST(r)	ST(r)	ST(0)
DF /2	FIST m16	Mem16	ST(0)
DB /2	FIST m32	Mem32	ST(0)
DF /3	FISTP m16	Mem16	ST(0)
DB /3	FISTP m32	Mem32	ST(0)
DF /7	FISTP m64	Mem64	ST(0)
DF /6	FBSTP	Mem80	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode

übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	1	1, (4)	(1)	1	(1)	-

FSTx kann die exceptions #U und #O auslösen, wenn der zu speichernde Wert nicht in das Zielformat passt. FIST, FISTP und FBSTP lösen stattdessen #IS mit Grund 4 aus, #O und #U sind hier nicht möglich.

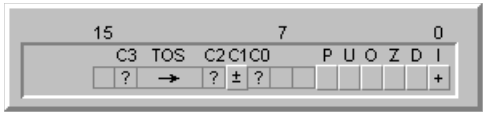
FXCH 8087

Funktion Vertausche den Inhalt zweier FPU-Register (»floating-point exchange«).

Operation

```
IF (Src explicit) THEN
   Temp := Src
   Src := ST(0)
   ST(0) := Temp
ELSE
   Temp := ST(1)
   ST(1) := ST(0)
   ST(0) := Temp
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. FXCH ist stackneutral. Außer #I werden keine Exceptions ausgelöst.

Opcodes

FXCH gibt es in der operandenfreien Form, in der beide Operanden impliziert werden, und in der Ein-Operanden-Form, in der der erste Operand impliziert wird und nicht angegeben werden darf.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 (C8+r)	FXCH ST(r)	ST(0)	ST(r)
D9 C9	FXCH	ST(0)	ST(1)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	1	-	-	-	-

FCMOVcc — Pentium Pro

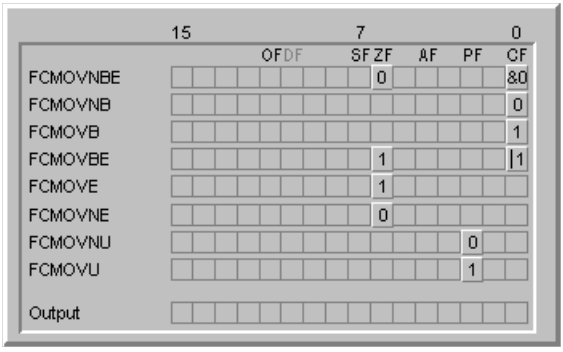
Funktion

Kopiere ein Datum aus einem FPU-Register, wenn eine Bedingung erfüllt ist (»floating-point move on condition«).

Operation

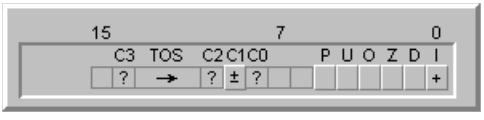
```
IF (Bedingung ist wahr) THEN
   ST(0) := ST(I)
```

CPU-Statusflags



FCMOVcc ist einer der wenigen FPU-Befehle, die direkt mit dem EFlags-Register der CPU interagieren. Bei FCMOVcc dient die Stellung der Flags als Input, Output in das EFlags-Register erfolgt nicht! Die einzelnen FCMOV-Varianten prüfen die Flagstellung der in der Abbildung gezeigten Flags. »|« bedeutet, dass eines der Flags gesetzt sein muss, damit die Bedingung erfüllt ist, bei »&« müssen beide gezeigten Flags den entsprechenden Zustand aufweisen.

FPU-Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. FXCH ist stackneutral. Außer #I werden keine Exceptions ausgelöst.

Opcodes

Bei dieser Befehlsgruppe ist der erste Operand zwar impliziert, muss aber dennoch angegeben werden. Es ist der TOS, ST(0).

Opcode	Mnemonic	Operanden	
		Dest	Src
DA (C0+r)	FCMOVB ST(0), ST(r)	ST(0)	ST(r)
DA (C8+r)	FCMOVB ST(0), ST(r)	ST(0)	ST(r)
DA (D0+r)	FCMOVBE ST(0), ST(r)	ST(0)	ST(r)
DA (D8+r)	FCMOVU ST(0), ST(r)	ST(0)	ST(r)
DB (C0+r)	FCMOVNB ST(0), ST(r)	ST(0)	ST(r)
DB (C8+r)	FCMOVNE ST(0), ST(r)	ST(0)	ST(r)
DB (D0+r)	FCMOVNBE ST(0), ST(r)	ST(0)	ST(r)
DB (D8+r)	FCMOVNU ST(0), ST(r)	ST(0)	ST(r)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebe-

nen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	1	-	-	-	-

FRNDINT 8087

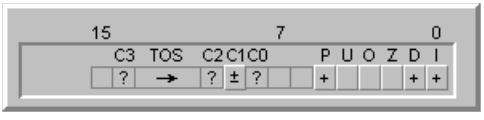
Funktion

Runde die Realzahl gemäß der festgelegten Rundungsregeln zu einer Integer (»floating-point round to integer«).

Operation

```
ST(0) := Round2Integer(ST(0))
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. FRNDINT ist stackneutral. An möglichen Exceptions kommen #P, #D und #I in Betracht.

Opcodes

Bei FRNDINT sind Quell- und Zieloperand impliziert. Es handelt sich in beiden Fällen um ST(0).

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 FC	FRNDINT	ST(0)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die

Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

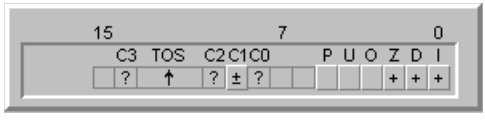
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1	1	-	1	-	-

FXTRACT 8087

Funktion Zerlege die Realzahl in ihre Mantisse und den Exponenten (zur Basis 2!!; »floating point extract exponent and significant«).

Operation
```
Temp := Mantisse(ST(0)
ST(0) := Exponent(ST(0)
TOS := (TOS + 7) MOD 8 ; (≡ zyklisches Dekrementieren)
ST(0) := Temp
```

Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. FRNDINT ist stackneutral. An möglichen Exceptions kommen #Z, #D und #I in Betracht.

Opcodes Bei FXTRACT sind beide Zieloperanden und der Quelloperand impliziert und dürfen nicht angegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 F4	FXTRACT	ST(0), ST(1)	ST(0)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die

Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1	1, 2	-	-	-	3

FSCALE 8087

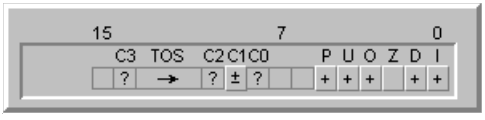
Funktion

Setze eine Mantisse und einen Exponenten (zur Basis 2!!) zu einer Realzahl zusammen (»floating-point scale«).

Operation

$ST(0) := ST(0) * 2^{ST(1)}$

Statusflags



C3, C2 und C0 sind undefiniert, C1 ist nur bei #I definiert. Bei #I (Subtyp: #IS) ist C1 bei einem stack underflow gesetzt, sonst gelöscht. FSCALE ist stackneutral. An möglichen Exceptions kommen außer #Z alle in Betracht.

Opcodes

Ziel- und beide Quelloperanden sind bei FSCALE impliziert, sie können nicht vorgegeben werden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 FD	FSCALE	ST(0)	ST(0), ST(1)

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1	1	1	1	1	-

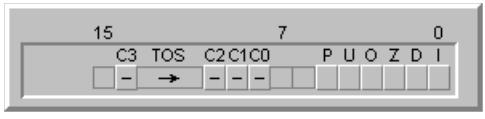
FINIT, FNINIT 8087

Funktion Initialisiere die FPU mit/ohne Synchronisation (»floating-point initialize FPU [no wait]«).

Operation
```
[ControlRegister] := $037F
[StatusRegister] := 0
[TagRegister] := $FFFF
[LDP] := 0
[LIP] := 0
[Op] := 0
```

Siehe auch FLDCW auf Seite 207.

Statusflags



Alle Flags des condition code werden gelöscht. FINIT/FNINIT sind stack-neutral. Exceptions werden nicht ausgelöst.

Opcodes FINIT und FNINIT besitzen keine Operanden. Die beiden Befehle unterscheiden sich dadurch, dass bei FINIT dem eigentlichen Befehl der Opcode $9B vorangestellt wird, sodass FINIT identisch mit der Sequenz FWAIT – FNINIT ist.

Opcode	Mnemonic	Operanden	
		Dest	Src
DB E3	FNINIT		
9B DB E3	FINIT		

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

FLDCW 8087
FSTCW, FNSTCW 8087
FSTSW, FNSTSW 8087

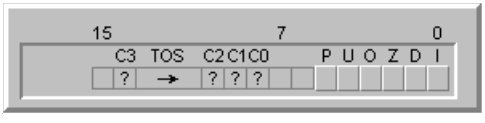
Funktion

Lade oder speichere das Kontroll-Word aus dem control register bzw. speichere das Status-Word aus dem status register der FPU mit/ohne Synchronisation (»floating-point load/store control/status word [no wait]«).

Operation

```
IF (Instruction = FLDCW) THEN
   [ControlRegister] := Src
ELSEIF (Instruction = F(N)STCW THEN
   Dest := [ControlRegister]
ELSE                    ; (Instruction = F(N)STSW
   Dest := [StatusRegister]
```

Statusflags



Der Zustand der Flags des condition code ist undefiniert. Die Befehle sind stackneutral, Exceptions werden nicht ausgelöst.

Opcodes

F(N)STSW gibt es in einer operandenlosen und einer Ein-Operanden-Form, in der wie bei den restlichen Befehlen die Quelle bzw. das Ziel vorgegeben werden kann. FSTCW und FSTSW sind wie FINIT zusam-

mengesetzte Befehlssequenzen. So steht FSTCW für FWAIT – FNSTCW und FSTSW analog für FWAIT – FNSTSW.

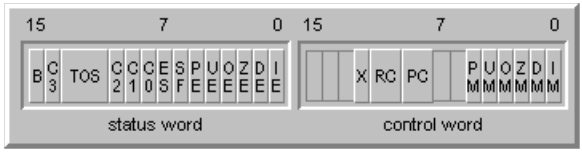
Opcode	Mnemonic	Operanden	
		Dest	Src
D9 /5	FLDCW m	control register	Mem2
D9 /7	FNSTCW m	Mem2	control register
9B D9 /7	FSTCW m	Mem2	control register
DD /7	FNSTSW m	Mem2	status register
DF E0	FNSTSW AX	AX	status register
9B DD /7	FSTSW m	Mem2	status register
9B DF E0	FSTSW AX	AX	status register

Die grau unterlegten Operanden sind implizit vor- und angegeben.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.

Bemerkungen Die Befehle greifen direkt auf das status bzw. control word der FPU zu:



Es muss somit gesichert sein, dass das Word, das FLDCW als Operand übergeben wird, den entsprechenden Aufbau besitzt. F(N)INIT trägt den Wert $037F ein, was bedeutet, dass round control (RC) auf 00_b gesetzt wird (Rundung zur

nächsten oder geraden Zahl), precision control (PC) auf 11_b (64-Bit-ExtendedReal-Genauigkeit) sowie alle Exception-Masken gesetzt werden.

Soll die FPU-Umgebung, nicht aber die Rechenregister gesichert werden, kann der Befehl F(N)STENV benutzt werden. F(N)SAVE sichert darüber hinaus auch die FPU-Rechenregister.

FXSAVE ist ein Befehl, der analog F(N)SAVE alle FPU-Register sichert, jedoch auch die mit den MMX- und in einer zweiten Stufe auch die mit den SSE-Befehlen eingeführten Erweiterungen berücksichtigt. FXSAVE und F(N)SAVE sind nicht kompatibel zueinander!

FCLEX, FNCLEX — 8087

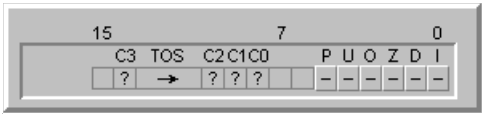
Funktion

Setze die exception flags der FPU zurück (»floating-point clear exception flags [no wait]«).

Operation

```
StatusRegister[07..00] := 0
StatusRegister[15] := 0
```

Statusflags



Die Flags des condition code sind undefiniert, die Exception-Flags werden gelöscht. FCLEX/ FNCLEX sind stackneutral.

Opcodes

FCLEX ist wie FINIT eine zusammengesetzte Befehlssequenz. So steht FCLEX für FWAIT – FNCLEX. Beide Befehle haben keine Operanden.

Opcode	Mnemonic	Operanden	
		Dest	Src
DB E2	FNCLEX		
9B DB E2	FCLEX		

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

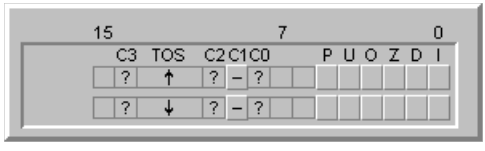
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

FDECSTP 8087
FINCSTP 8087

Funktion Erkläre das nächsthöhere/nächstniedrigere FPU-Register zum Top Of Stack (»floating-point increment/decrement stack pointer«).

Operation
```
IF (Instruction = FDECSTP) THEN
   TOS := (TOS + 7) MOD 8 ; (≡ zyklisches Dekrementieren)
ELSE                      ; (Instruction = FINCSTP)
   TOS := (TOS + 1) Mod 8 ; (≡ zyklisches Inkrementieren)
```

Statusflags



Alle Flags des condition codes außer C1 sind undefiniert, C1 wird explizit gelöscht. FDECSTP führt einen stack push durch (Dekrementieren des Stack-Pointers), FINCSTP einen stack pop (Inkrementieren des stack pointer). Exceptions werden nicht ausgelöst.

Opcodes FDECSTP und FINCSTP besitzen keine Operanden. Sie verändern lediglich den Inhalt des TOS-Feldes im status register.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 F6	FDECSTP		
D9 F7	FINCSTP		

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die

Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

FFREE — 8087

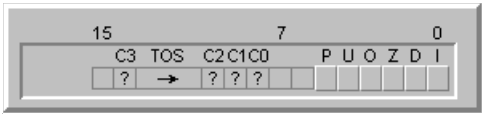
Funktion

Markiere das spezifizierte FPU-Register als leer (»floating-point free register«).

Operation

```
TagRegister[Dest] := 11b
```

Statusflags



Alle Flags des condition codes sind undefiniert. FFREE ist stackneutral, Exceptions werden nicht ausgelöst.

Opcodes

FFREE besitzt einen Operanden, der das FPU-Register spezifiziert, dessen Inhalt als empty markiert werden soll.

Opcode	Mnemonic	Operanden	
		Dest	Src
DD (C0+r)	FFREE ST(r)	ST(r)	

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

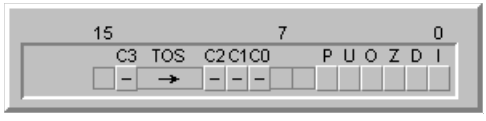
FSAVE, FNSAVE 8087

Funktion Sichere die FPU-Register mit/ohne Synchronisation (»floating-point save all fpu registers [no wait]«).

Operation

```
; Daten sichern
Dest[ControlWord] := [ControlRegister]
Dest[StatusWord] := [StatusRegister]
Dest[TagWord] := [TagRegister]
Dest[LDP] := [LDP]
Dest[LIP] := [LIP]
Dest[Op] := [Op]
FOR (I = 0) TO (I = 7) DO
   Dest[ST(I)] := [ST(I)]
; Initialisieren
[ControlRegister] := $037F
[StatusRegister] := 0
[TagRegister] := $FFFF
[LDP] := 0
[LIP] := 0
[Op] := 0
```

Statusflags



Alle Flags des condition code werden gelöscht, nachdem sie gesichert wurden. FSAVE/FNSAVE sind stackneutral. Exceptions werden nicht ausgelöst.

Opcodes FSAVE ist wie FINIT eine zusammengesetzte Befehlssequenz. So steht FSAVE für FWAIT – FNSAVE. Je nach Betriebsmodus erwarten die Befehle einen 108-Byte- oder einen 94-Byte-Speicheroperanden.

Opcode	Mnemonic	Operanden	
		Dest	Src
DD /6	FNSAVE m	Mem108 (Mem94)	
9B DD /6	FSAVE m	Mem108 (Mem94)	

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

Bemerkungen

Speicherabbilder der Datenstruktur, die F(N)SAVE/FRSTOR benutzen, sind abhängig vom Betriebsmodus und der Umgebung.

Soll nur die FPU-Umgebung, nicht aber die Rechenregister gesichert werden, kann der Befehl F(N)STENV benutzt werden. F(N)STCW/F(N)STSW sichern nur control bzw. status register.

FXSAVE ist ein Befehl, der analog F(N)SAVE die FPU-Register sichert, jedoch auch die mit den MMX- und in einer zweiten Stufe auch die mit den SSE-Befehlen eingeführten Erweiterungen berücksichtigt. FXSAVE und F(N)SAVE sind nicht kompatibel zueinander!

FRSTOR — 8087

Funktion

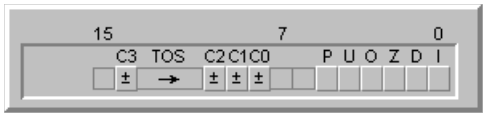
Restauriere die FPU-Register (»floating-point restore all FPU-Registers«).

Operation

```
[ControlRegister] := Src[ControlWord]
[StatusRegister] := Src[StatusWord]
[TagRegister] := Src[TagWord]
[LDP] := Src[LDP]
```

```
[LIP] := Src[LIP]
[Op] := Src[Op]
FOR (I = 0) TO (I = 7) DO
   [ST(I)] := Src[ST(I)]
```

Statusflags



Alle Flags des condition code werden gemäß den Daten im Operanden verändert. FRSTOR ist stackneutral. Exceptions werden nicht ausgelöst.

Opcodes Je nach Betriebsmodus erwartet FRSTOR einen 108-Byte- oder einen 94-Byte-Speicheroperanden. FRSTOR erwartet als Operanden eine Speicherstelle, die zuvor mittels FSAVE mit Daten belegt wurde. Die Datenstrukturen von FXSAVE/FXRSTOR und F(N)SAVE/FRSTOR sind nicht kompatibel!

Opcode	Mnemonic	Operanden	
		Dest	Src
DD /4	FRSTOR m		Mem108 (Mem94)

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

FXSAVE — Pentium II (erweitert durch SSE)

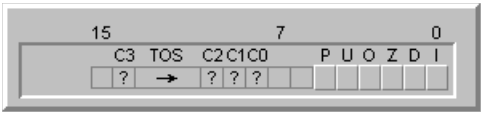
Funktion

Sichere die FPU-, MMX- und ggf. XMM-Register (»floating-point extended save all fpu, mmx and xmm registers«).

Operation

```
Dest[ControlWord] := [ControlRegister]
Dest[StatusWord] := [StatusRegister]
Dest[TagWord] := [TagRegister]
Dest[LDP] := [LDP]
Dest[LIP] := [LIP]
Dest[Op] := [Op]
; FPU/MMX-Register
FOR (I = 0) TO (I = 7) DO
   Dest[ST(I)] := [ST(I)]
; XMM-Register
IF (SSE2 in feature flags nach CPUID = 1) THEN
   Dest[MXCSR] := [MXCSR]
   Dest[MXCSR-Mask] := [MXCSR-Mask]
   FOR (I = 0) TO (I = 7) DO
      Dest[XMM(I)] := [XMM(I)]
```

Statusflags



Die Flags des condition codes sind undefiniert. FXSAVE ist stackneutral. Exceptions werden keine ausgelöst.

Opcodes

FXSAVE kommt im Gegensatz zu FSAVE, FSTSW, FSTSW und FSTENV nicht in einer N-Variante vor!

Opcode	Mnemonic	Operanden	
		Dest	Src
0F AE /0	FXSAVE m	Mem512	

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	3	0	1	0	./:
#GP	3, 8, 20, 69	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.
#UD	17, 18, 19	-	17, 18, 19	-	17, 18, 19

Bemerkungen Soll nur die FPU-Umgebung, nicht aber die FPU-Rechen- und MMX- bzw. XMM-Register gesichert werden, kann der Befehl F(N)STENV benutzt werden. F(N)STCW/F(N)STSW sichern nur das FPU-Kontroll- bzw. Status-Register.

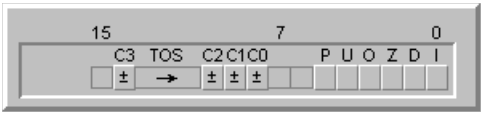
F(N)SAVE ist ein Befehl, der analog FXSAVE die FPU-Register sichert, jedoch nicht die mit den MMX- bzw. SSE-Befehlen eingeführten Erweiterungen berücksichtigt. FXSAVE und F(N)SAVE sind nicht kompatibel zueinander!

FXRSTOR — Pentium II (erweitert durch SSE)

Funktion Restauriere die FPU-, MMX- und ggf. XMM-Register (»floating-point extended restore all fpu, mmx and xmm registers«).

Operation

```
[ControlRegister] := Src[ControlWord]
[StatusRegister] := Src[StatusWord]
[TagRegister] := Src[TagWord]
[LDP] := Src[LDP]
[LIP] := Src[LIP]
[Op] := Src[Op]
; FPU/MMX-Register
FOR (I = 0) TO (I = 7) DO
   [ST(I)] := Src[ST(I)]
; XMM-Register
IF (SSE2 in feature flags nach CPUID = 1) THEN
   [MXCSR] := Src[MXCSR]
   [MXCSR-Mask] := Src[MXCSR-Mask]
FOR (I = 0) TO (I = 7) DO
   [ST(I)] := Src[ST(I)]
```



Statusflags

Alle Flags des condition code werden gemäß den Daten im Operanden verändert. FXRSTOR ist stackneutral. Exceptions werden nicht ausgelöst.

Opcodes

FXRSTOR erwartet als Operanden eine Speicherstelle, die zuvor mittels FXSAVE mit Daten belegt wurde. Die Datenstrukturen von FXSAVE/FXRSTOR und F(N)SAVE/FRSTOR sind nicht kompatibel!

Opcode	Mnemonic	Operanden	
		Dest	Src
0F AE /1	FXRSTOR m		Mem512

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	3	0	1	0	./:
#GP	3, 8, 20, 69	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.
#UD	17, 18, 19	-	17, 18, 19	-	17, 18, 19

FSTENV, FNSTENV — 8087

Funktion

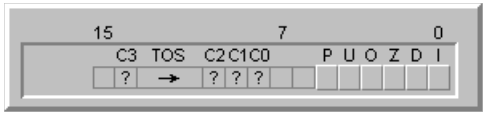
Sichere die FPU-Umgebung mit/ohne Synchronisation (»floating-point store environment [no wait]«).

Operation

```
Dest[ControlWord] := [ControlRegister]
Dest[StatusWord] := [StatusRegister]
Dest[TagWord] := [TagRegister]
```

```
Dest[LDP] := [LDP]
Dest[LIP] := [LIP]
Dest[Op] := [Op]
```

Statusflags



Die Flags des condition codes sind undefiniert. FSTENV/FNSTENV sind stackneutral. Exceptions werden keine ausgelöst.

Opcodes FSTENV ist wie FINIT eine zusammengesetzte Befehlssequenz. So steht FSTENV für FWAIT – FNSTENV. Je nach Betriebsmodus erwarten die Befehle einen 28-Byte- oder einen 14-Byte-Speicheroperanden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 /6	FNSTENV m	Mem28 (Mem14)	
9B D9 /6	FSTENV m	Mem28 (Mem14)	

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	3	0	1	0	./:
#GP	3, 8, 20	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.

Bemerkungen Sollen zusätzlich die Rechenregister gesichert werden, kann der Befehl F(N)SAVE benutzt werden. F(N)STCW/F(N)STSW sichern nur control bzw. status register der FPU.

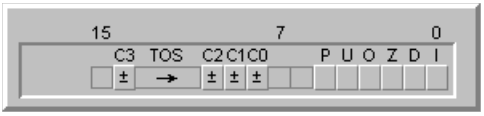
FXSAVE ist ein Befehl, der analog F(N)SAVE die FPU-Register sichert, jedoch auch die mit den MMX- und in einer zweiten Stufe auch die mit den SSE-Befehlen eingeführten Erweiterungen berücksichtigt. FXSAVE und F(N)SAVE sind nicht kompatibel zueinander!

FLDENV 8087

Restauriere die FPU-Umgebung (»floating-point load environment«). *Funktion*

Operation

```
[ControlRegister] := Src[ControlWord]
[StatusRegister] := Src[StatusWord]
[TagRegister] := Src[TagWord]
[LDP] := Src[LDP]
[LIP] := Src[LIP]
[Op] := Src[Op]
```



Statusflags

Alle Flags des condition code werden gemäß den Daten im Operanden verändert. FLDENV ist stackneutral. Exceptions werden nicht ausgelöst.

Opcodes

Je nach Betriebsmodus erwartet FLDENV einen 28-Byte- oder einen 14-Byte-Speicheroperanden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 /4	FLDENV m		Mem28 (Mem14)

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

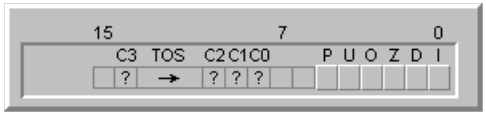
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	3	0	1	0	./:
#GP	3, 8	0	8	0	8
#NM	2	-	2	-	2
#PF	1	fault code	1	fault code	1
#SS	1	0	1	0	./.

FNOP 8087

Funktion FPU-Variante des CPU-NOP-Befehls (»floating-point no operation«).

Operation
```
Führe keine Aktion durch!
```

Statusflags



Alle Flags des condition codes sind undefiniert. FNOP ist stackneutral, Exceptions werden nicht ausgelöst.

Opcodes Der Befehl hat keine Operanden.

Opcode	Mnemonic	Operanden	
		Dest	Src
D9 D0	FNOP		

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

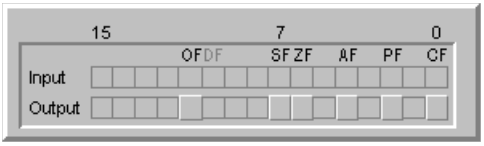
Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

FWAIT (WAIT) 8087

Funktion Synchronisiere FPU und CPU, indem wartende FPU-Exceptions behandelt werden.

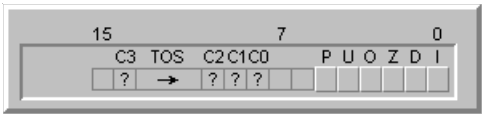
Operation
```
IF (#MF pending) THEN
   CallExceptionHandler(#MF)
```



CPU-Statusflags

Input: keiner

Output: Status-Flags werden nicht verändert.



FPU-Statusflags

Alle Flags des condition codes sind undefiniert. FWAIT ist stackneutral, Exceptions werden nicht ausgelöst.

Opcodes

FWAIT und WAIT sind Synonyme für die CPU-Instruktion $9B, die die Synchronisierung zwischen FPU-Exceptionbearbeitung und CPU-Instruktionen herbeiführt.

Opcode	Mnemonic	Operanden	
		Dest	Src
9B	FWAIT / WAIT		

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die FPU-Exceptions entnehmen Sie bitte Tabelle 5.14 bis Tabelle 5.20 ab Seite 295.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	2	-	2	-	2
#MF	1	-	1	-	1
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

4 SIMD-Instruktionen

Befehl(e) **eingeführt unter (erweitert unter)**

Kurze Beschreibung der Funktion. *Funktion*

`Formale Darstellung der durchgeführten Operation in einer Pascal-ähnlichen Notation.` *Operation*

Die Rubrik »Statusflags«, die bei CPU- und FPU-Instruktionen eine Rolle spielt, gibt es bei SIMD-Instruktionen nicht. Zwar verfügen auch die XMM-Register analog den FPU-Registern über ein Status- und Kontrollregister, das MXCSR (multi-media extensions control and status register). Dieses dient jedoch, was den Teil mit den Statusangaben betrifft, lediglich dazu, den Grund für das Vorliegen einer XMM-Exception zu signalisieren. Flags, die einen »Status« nach bestimmten SIMD-Operationen signalisieren sollen und Statusflags im Sinne der Statusflags der CPU oder FPU darstellen, gibt es unter SIMD nicht! *Statusflags*

Integer-SIMD-Befehle mit unterschiedlichen Operandengrößen unterscheiden sich weder im Mnemonic noch im Opcode voneinander. Wie bei den CPU-Befehlen auch erfolgt eine Spezifizierung anhand des operand size override prefix. So beziehen sich alle Befehle ohne diesen Präfix auf ShortPackedIntegers und damit auf die MMX-Register, während alle Befehle mit dem Präfix PackedIntegers verarbeiten und damit in den XMM-Registern ablaufen. Der Assembler kann daher den Einsatz des operand size override prefix davon abhängig machen, ob der Programmierer als beteiligtes Register ein MMX- oder ein XMM-Register angibt. *Opcodes*

In den Tabellen wird auch ein ModR/M-Byte ausgewiesen, falls eines erforderlich ist. Seine Notwendigkeit wird ggf. durch »/r« angegeben. Grund: Einige Befehle verwenden das »reg«-Feld dieses Bytes (Bit 5-3) als »spec«-Feld und damit für die Erweiterung des Opcodes. Dies wird signalisiert, indem dem Schrägstrich eine Ziffer (0 – 7) folgt: »/5«.

Es gibt Befehle, bei denen nicht nur der operand size override prefix herangezogen wird, um die Operanden zu unterscheiden. So besitzen z.B. PSHUFW, PSHUFD, PSHUFLW und PSHUFHW den gleichen Opcode 0F 70. Ohne jeden Präfix korrespondiert dieser Opcode mit dem Mnemonic PSHUFW und richtet sich an MMX-Register. Mit einem Präfix dagegen sind die XMM-Register Ziel der Aktivitäten. Das XMM-Analogon zu PSHUFW, PSHUFD, benutzt hierzu den operand size override prefix $66, während der Präfix $F3 (eigentlich REP) für das höherwertige Word (PSHUFHW) und der Präfix $F2 (eigentlich REPNE) für das niederwertige Word (PSHUFLW) in den gepackten Strukturen zuständig ist. Die benutzten Präfixe werden bei dem Opcode angegeben.

Die Real-SIMD-Befehle beziehen sich immer auf die XMM-Register (zu den 3D-Now!-Befehlen gibt es nicht nur aus diesem Grunde einen extra Abschnitt!). Hier muss somit die Unterscheidung der Datengröße anhand unterschiedlicher Mnemonics erfolgen, die jedoch in den gleichen Opcode übersetzt werden. Signalisiert ein Mnemonic, dass mit PackedSingles gearbeitet werden soll, wird der operand size override prefix nicht verwendet. Er wird dem Opcode jedoch vorangestellt, wenn mit PackedDoubles gearbeitet werden soll. Die folgende Tabelle zeigt ein Beispiel mit einer Integer und einer Real im SIMD-Format.

Alle Befehle des 3D-Now!-Instruktionssatzes spielen sich in den MMX-Registern ab (ShortPackedSingles). Daher gibt es keinerlei Notwendigkeit zu einem operand size override prefix. Dennoch zeichnet sich ein 3D-Now!-Opcode durch eine Besonderheit aus: Er hat immer den Zwei-Byte-Opcode $0F0F! Die Unterscheidung, welcher 3D-Now!-Befehl nun gewünscht wird, erfolgt durch ein Suffix-Byte, das in der Opcode-Spalte hinter dem Zeichen »]« angefügt wird.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	**XMM (128 Bit)**		**Dest**	**Src**
`0F FC /r`	`0F FC /r`	PADDB s, s/m	SIMD	SIMD / Mem
	`0F 58 /r`	ADDPS x, x/m ADDPD x, x/m	XMM	XMM / Mem
`0F 0F ]9E`		PFADD mm, mm/m	MMX	MMX/ Mem

In der Tabelle bedeuten in der Opcode-Spalte

xx[Der Opcode verwendet verschiedene Präfixe zur Unterscheidung einzelner Befehle. Der dem jeweiligen Befehl zugeord-

nete Präfix wird als xx mit Hexadezimalzeichen vor dem Präfix-Zeichen »[« angegeben. Als Präfixe kommen in Frage: der operand size override prefix $66 sowie die hier dem früheren, eigentlichen Zweck entfremdeten Repetier-Präfixe $F2 und $F3.

]xx Der Opcode besitzt ein Suffix-Byte als letztes Byte der Befehlssequenz. Daher handelt es sich bei dem Befehl um einen 3D-Now-Befehl und die beiden Opcode-Bytes lauten 0F 0F. Der dem Befehl zugeordnete Suffix wird als xx mit Hexadezimalzeichen hinter dem Suffix-Zeichen »]« angegeben.

/r Der Opcode besitzt ein *ModR/M*-Byte, da als Operanden sowohl ein Register als auch ein Register bzw. eine Speicherstelle angegeben werden müssen, die mittels der Felder *reg* und *r/m* kodiert werden.

/0 ... /7 Der Opcode besitzt ein *ModR/M*-Byte mit modifizierter Bedeutung: Das *reg*-Feld im *ModR/M*-Byte kann nur den spezifizierten Wert annehmen (z.B. 010_b bei /2) und dient der Erweiterung des Opcodes. Zur Kodierung von Operanden kann somit nur das *r/m*-Feld herangezogen werden. Das bedeutet entweder, dass es nur einen Operanden gibt, oder dass ein evtl. möglicher zweiter Operand nur eine Konstante sein kann, die nicht via *reg* kodiert werden muss.

+r Zu dem Byte des Opcodes wird ein Wert zwischen 0 und 7 addiert, der das Register kodiert, das im Befehl eine Rolle spielt:

Operandengröße	**r =**							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
64 Bit (MMX)	MM0	MM1	MM2	MM3	MM4	MM5	MM5	MM7
128 Bit (XMM)	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

Ferner bedeuten in der Mnemonic-Spalte:

c[x] Konstante der entsprechenden Operandengröße (8, 16 oder 32 Bit). Wird eine bestimmte Größe der Konstanten vorausgesetzt, wird sie als x angegeben, z.B. in c8.

r Allzweckregister der CPU

m[x] Speicherstelle, die ein Datum der entsprechenden Operandengröße (64 bzw. 128 Bit) aufnimmt. Wird eine bestimmte

Operandengröße vorausgesetzt, wird sie als x angegeben, z.B. in m64.

mm MMX-Register (64 Bit)

mm/m Der Operand kann entweder ein MMX-Register (siehe mm) oder eine Speicherstelle sein.

s SIMD-Register. Das kann je nach Operand ein MMX- (64 Bit) oder ein XMM-Register (128 Bit) sein.

s/m Der Operand kann entweder ein SIMD-Register (siehe s) oder eine Speicherstelle sein.

x XMM-Register (128 Bit)

x/m Der Operand kann entweder ein XMM-Register (siehe x) oder eine Speicherstelle sein.

Schließlich bedeuten in der Operanden-Spalte analog zur Mnemonic-Spalte:

Const[x] Konstante der entsprechenden Operandengröße (8, 16 oder 32 Bit). Wird eine bestimmte Größe der Konstanten vorausgesetzt, wird sie als x angegeben, z.B. in Const8.

Reg Allzweckregister der CPU. Es können allerdings nur 32-Bit-Register verwendet werden.

Mem[x] Speicherstelle, die ein Datum der entsprechenden Operandengröße (64 bzw. 128 Bit) aufnimmt. Wird eine bestimmte Operandengröße vorausgesetzt, wird sie als x angegeben, z.B. in Mem64, womit eine ShortPackedInt adressiert wird. In der Befehlssequenz stellt Mem eine effektive Adresse, also einen Offset dar, der als Konstante hinter den Opcode geschrieben wird. Die Größe dieses Offsets ist abhängig vom aktuellen Betriebsmodus, der aktuellen Umgebung und dem address size override prefix. Sie kann 16 oder 32 Bit betragen, sodass dem Opcode je nach Situation zwei oder vier Adressenbytes folgen. Falls sich der Offset auf ein anderes Segment bezieht, kann der Einsatz eines segment override prefix erforderlich werden.

MMX — MMX-Register (64 Bit). Gültige Argumente für die Mnemonics sind MM0 bis MM7 (ShortPackedSingles).

MMX/Mem — Der Operand kann entweder ein MMX-Register (siehe MMX) oder eine Speicherstelle für 64-Bit-Operanden sein.

SIMD — SIMD-Register. Je nach eingesetztem Register-Operanden handelt es sich um ein MMX- oder XMM-Register. Gültige Argumente für die Mnemonics sind: MM0 bis MM7 (MMX-Register; ShortPackedInteger) und XMM0 bis XMM7 (XMM-Register; PackedInteger)

SIMD/Mem — Der Operand kann entweder ein SIMD-Register (siehe SIMD) oder eine Speicherstelle für 64-Bit-Operanden sein.

XMM — XMM-Register (128 Bit). Gültige Argumente für die Mnemonics sind XMM0 bis XMM7 (PackedSingles und PackedDoubles).

XMM/Mem — Der Operand kann entweder ein XMM-Register (siehe XMM) oder eine Speicherstelle für 128-Bit-Operanden sein.

Exceptions (MMX)

Integer-SIMD-Instruktionen lösen wegen des verwendeten Datentyps (Integer) keine FPU-Exceptions aus, selbst wenn die Operationen in den MMX-Registern durchgeführt werden. Allerdings können Exceptions generiert werden, die die gleichen Gründe haben wie CPU-Instruktionen. Die bei dem entsprechenden Befehl möglichen Gründe werden tabellarisch in Abhängigkeit des Betriebsmodus des Prozessors angegeben:

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Exceptions (XMM)

Real-SIMD-Instruktionen dagegen können sehr wohl Real-spezifische Exceptions auslösen, die denen der FPU-Exceptions sehr ähnlich sind. Daher werden bei diesen Befehlen zusätzlich Angaben gemacht, wie sie auch unter den FPU-Befehlen erfolgten:

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	1	0	1	0	./:
#GP	3, 8	0	8	0	8
#PF	1	fault code	1	fault code	./.
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

Bitte beachten Sie, dass der Grund für die Auslösung einer #XM mindestens ein gesetztes, unmaskiertes XMM-Exception-Flag beim Eintritt in den Befehl ist! Die Exception-Flags für den aktuellen Befehl werden erst dann gesetzt, wenn die anhängige Exception behandelt wurde.

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen auch hier die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein Error-Code übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Die Gründe für die XMM-Exceptions sind analog zu denen der FPU zu sehen. Daher können Sie sie Tabelle 5.14 bis Tabelle 5.20 ab Seite 295 entnehmen.

Bemerkungen

Hier werden verschiedene Bemerkungen oder Hinweise gegeben. Dieser Abschnitt ist optional und nicht bei allen Befehlen vorhanden.

4.1 SIMD-Integer-Befehle

PADDB, PADDW, PADDD	**MMX (SSE2)**
PADDQ	**SSE2**
PADDUSB, PADDUSW	**MMX (SSE2)**
PADDSB, PADDSW	**MMX (SSE2)**
PSUBB, PSUBW, PSUBD	**MMX (SSE2)**
PSUBQ	**SSE2**
PSUBUSB, PSUBUSW	**MMX (SSE2)**
PSUBSB, PSUBSW	**MMX (SSE2)**

Funktion

Addiere oder subtrahiere ShortPackedIntegers, PackedIntegers und QuadWords entweder im wrap-around mode oder mit Hilfe signed oder unsigned saturation (»packed addition/subtraction of [[un]signed] bytes/words/doublewords/quadwords«).

Operation

```
IF (OperandSize = 128) THEN
   F := 2
ELSE                    ; (OperandSize = 64)
   F := 1
IF (ScalarSize(Operand) = Byte, ShortInt) THEN
   N := 8
   S := 8
ELSEIF (ScalarSize(Operand) = Word, SmallInt) THEN
   N := 4
   S := 16
ELSEIF (ScalarSize(Operand) = DoubleWord, LongInt) THEN
   N := 2
   S := 32
ELSE                    ; (ScalarType = QuadWord, QuadInt)
   N := 1
   S := 64
FOR (I = 0) TO (I = F * N - 1) DO
   U := (I + 1) * S – 1  ; höchstes Bit
   L := I * S            ; niederstes Bit
   IF (Operation = Addition) THEN
      Temp[U..L] := Dest[U..L] + Src[U..L]
   ELSE                  ; (Operation = Subtraktion)
      Temp[U..L] := Dest[U..L] – Src[U..L]
   IF (saturation = signed) THEN
      Dest[U..L] := SignedSaturation(Temp[U..L]
```

```
ELSEIF (saturation = unsigned) THEN
   Dest[U..L] := UnsignedSaturation(Temp[U..L]
ELSE                    ; (wrap-around)
   Dest := WrapAround(Temp)
```

Opcodes Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	**XMM (128 Bit)**		**Dest**	**Src**
0F FC /r	66[0F FC /r	PADDB s, s/m	SIMD	SIMD / Mem
0F FD /r	66[0F FD /r	PADDW s, s/m	SIMD	SIMD / Mem
0F FE /r	66[0F FD /r	PADDD s, s/m	SIMD	SIMD / Mem
0F D4 /r	66[0F D4 /r	PADDQ s, s/m	SIMD	SIMD / Mem
0F EC /r	66[0F EC /r	PADDSB s, s/m	SIMD	SIMD / Mem
0F ED /r	66[0F ED /r	PADDSW s, s/m	SIMD	SIMD / Mem
0F DC /r	66[0F DC /r	PADDUSB s, s/m	SIMD	SIMD / Mem
0F DD /r	66[0F DD /r	PADDUSW s, s/m	SIMD	SIMD / Mem
0F F8 /r	66[0F F8 /r	PSUBB s, s/m	SIMD	SIMD / Mem
0F F9 /r	66[0F F9 /r	PSUBW s, s/m	SIMD	SIMD / Mem
0F FA /r	66[0F FA /r	PSUBD s, s/m	SIMD	SIMD / Mem
0F FB /r	66[0F FB /r	PSUBQ s, s/m	SIMD	SIMD / Mem
0F E8 /r	66[0F E8 /r	PSUBSB s, s/m	SIMD	SIMD / Mem
0F E9 /r	66[0F E9 /r	PSUBSW s, s/m	SIMD	SIMD / Mem
0F D8 /r	66[0F D8 /r	PSUBUSB s, s/m	SIMD	SIMD / Mem
0F D9 /r	66[0F D9 /r	PSUBUSW s, s/m	SIMD	SIMD / Mem

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PMULHW, PMULLW — MMX (SSE2)
PMULHUW — SSE (SSE2)
PMULHRW — 3D-Now!
PMULUDQ — SSE2

Funktion

Multipliziere zwei Words aus gepackten Strukturen miteinander (vorzeichenlos oder vorzeichenbehaftet, ggf. mit Rundung) und speichere als Ergebnis die entstandenen Quadwords oder jeweils das höherwertige oder niedrigerwertige Word (»packed multiply unsigned words/signed words [with rounding] and store of high word/low word/quadword«).

Operation

```
IF (Instruction = PMULHUW, PMULUDQ) THEN
  OP := unsigned multiplication
ELSEIF (Instruction = PMULHRW) THEN
  OP := signed multiplication with rounding
ELSE                    ; (Operation = PMULHW, PMULLW)
  OP := signed multiplication
IF (Instruction = PMULUDQ) THEN
   Dest[063..000] := Dest[031..000] OP Src[031..000]
   IF (OperandSize = 128) THEN
      Dest[127..064] := Dest[095..064] OP Src[095..064]
ELSE                    ; (Instruction ≠ PMULUDQ)
   Temp[031..000] := Dest[015..000] OP Src[015..000]
   Temp[063..032] := Dest[031..016] OP Src[031..016]
   Temp[095..064] := Dest[047..032] OP Src[047..032]
   Temp[127..096] := Dest[063..048] OP Src[063..048]
   IF (OperandSize = 128) THEN
      Temp[159..128] := Dest[079..064] OP Src[079..064]
      Temp[191..160] := Dest[095..080] OP Src[095..080]
      Temp[223..192] := Dest[111..096] OP Src[111..096]
      Temp[255..224] := Dest[127..112] OP Src[127..112]
   IF (Instruction = PMULHW, PMULHUW, PMULHRW) THEN
      Dest[015..000] := Temp[031..016]
```

```
      Dest[031..016] := Temp[063..048]
      Dest[047..032] := Temp[095..080]
      Dest[063..048] := Temp[127..112]
      IF (OperandSize = 128) THEN
         Dest[079..064] := Temp[159..144]
         Dest[095..080] := Temp[191..176]
         Dest[111..096] := Temp[223..208]
         Dest[127..112] := Temp[255..240]
   IF (Instruction = PMULLW) THEN
      Dest[015..000] := Temp[015..000]
      Dest[031..016] := Temp[047..032]
      Dest[047..032] := Temp[079..064]
      Dest[063..048] := Temp[111..096]
      IF (OperandSize = 128) THEN
         Dest[079..064] := Temp[143..128]
         Dest[095..080] := Temp[175..160]
         Dest[111..096] := Temp[207..192]
         Dest[127..112] := Temp[239..224]
```

Opcodes Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	**XMM (128 Bit)**		**Dest**	**Src**
0F E5 /r	66[0F E5 /r	PMULHW s, s/m	SIMD	SIMD / Mem
0F D5 /r	66[0F D5 /r	PMULLW s, s/m	SIMD	SIMD / Mem
0F E4 /r	66[0F E4 /r	PMULHUW s, s/m	SIMD	SIMD / Mem
0F 0F]B7		PMULHRW mm, mm/m	MMX	MMX / Mem64
0F F4 /r	66[0F F4 /r	PMULUDQ s, s/m	SIMD	SIMD / Mem

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PMADDWD — MMX (SSE2)

Funktion

Multipliziere zwei gepackte Integer miteinander und addiere paarweise (»packed multiplication of words to doublewords and addition«).

Operation

```
IF (OperandSize = 128) THEN
   Dest[127..096] := (Dest[127..112] * Src[127..112])
                   +  (Dest[111..096] * Src[111..096])
   Dest[095..064] := (Dest[095..080] * Src[095..080])
                   +  (Dest[080..064] * Src[080..064])
Dest[063..032] := (Dest[063..048] * Src[063..048])
                +  (Dest[047..032] * Src[047..032])
Dest[031..000] := (Dest[031..016] * Src[031..016])
                +  (Dest[015..000] * Src[015..000])
```

Opcodes

Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F F5 /r	66[0F F5 /r	PMADDWD s, s/m	SIMD	SIMD / Mem

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PMAXUB, PMAXSW — SSE (SSE2)
PMINUB, PMINSW — SSE (SSE2)
PAVGB, PAVGW — SSE (SSE2)
PAVGUSB — 3D-Now!
PSADBW — SSE (SSE2)

Funktion Bestimmung von Maxima, Minima, Durchschnittswerten und der Summe der Absoluten Differenzen in gepackten Daten (»packed maximum/minimum of [un]signed bytes/words«, »packed average of [unsigned] bytes/words«, »packed sum of absolute differences of bytes and store in word«).

Operation

```
IF (OperandSize = 128) THEN
   F := 2
ELSE                  ; (OperandSize = 64)
   F := 1
IF (ScalarSize(Operand) = Byte, ShortInt) THEN
   N := 8
   S := 8
ELSE                  ; (ScalarSize(Operand) = Word, SmallInt)
   N := 4
   S := 16
FOR (I = 0) TO (I = F * N - 1) DO
   U := (I + 1) * S – 1  ; höchstes Bit
   L := I * S            ; niederstes Bit
   IF (Operation = MAX) THEN
      IF (Dest[U..L] > Src[U..L] THEN
         ; ist bereits Maximum
      ELSE
         Dest[U..L] := Src[U..L]
   ELSEIF (Operation = Min) THEN
```

```
      IF (Dest[U..L] < Src[U..L] THEN
         ; ist bereits Minimum
      ELSE
         Dest[U..L] := Src[U..L]
   ELSEIF (Operation = Average) THEN
      Temp[U..L] := (Dest[U..L] + Src[U..L] + 1) DIV 2
      IF (saturation = unsigned) THEN
         Dest[U..L] := UnsignedSaturation(Temp[U..L])
      ELSE
         Dest[U..L] := Temp[U..L]
   ELSE              ; (Operation = PSADBW)
      Temp[U..L] := ABS(Dest[U..L] – Src[U..L])
IF (Operation = PSADBW) THEN
   Dest := 0
   FOR (I = 0) TO (I = 7) DO
      U := (I + 1) * 8 – 1
      L := I * 8
      Dest[15..00] := Dest[15..00] + ZeroExtend(Temp[U..L])
   IF (F > 1) THEN
   FOR (I = 0) TO (I = 7) DO
      U := 64 + (I + 1) * 8 – 1
      L := 64 + I * 8
      Dest[79..64] := Dest[79..64] + ZeroExtend(Temp[U..L])
```

Opcodes

Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe. PAVGUSB ist nur mit MMX-Registern möglich.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	**XMM (128 Bit)**		**Dest**	**Src**
0F DE /r	66[0F DE /r	PMAXUB s, s/m	SIMD	SIMD / Mem
0F EE /r	66[0F EE /r	PMAXSW s, s/m	SIMD	SIMD / Mem
0F DA /r	66[0F DA /r	PMINUB s, s/m	SIMD	SIMD / Mem
0F EA /r	66[0F EA /r	PMINSW s, s/m	SIMD	SIMD / Mem
0F E0 /r	66[0F E0 /r	PAVGB s, s/m	SIMD	SIMD / Mem
0F E3 /r	66[0F E3 /r	PAVGW s, s/m	SIMD	SIMD / Mem
0F 0F]BF		PAVGUSB mm, mm/m	MMX	MMX / Mem64
0F F6 /r	66[0F F6 /r	PSADBW s, s/m	SIMD	SIMD / Mem

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PAND MMX (SSE2)
PANDN MMX (SSE2)
POR MMX (SSE2)
PXOR MMX (SSE2)

Funktion Logische Operationen mit gepackten Datenstrukturen (»packed AND/AND NOT/OR/XOR«).

Operation
```
IF (OperandSize = 128) THEN
   N := 127
ELSE                    ; (OperandSize = 64)
   N := 63
FOR (I = 0) TO (I = N) DO
   IF (Instruction = PAND) THEN
     Dest[I] := Dest[I] AND Src[I]
ELSEIF (Instruction = PANDN) THEN
   Dest[I] := (NOT Dest[i]) AND Src[I]
ELSEIF (Instruction = POR) THEN
   Dest[I] := Dest[I] OR Src[I]
ELSE                    ; (Instruction = PXOR)
   Dest[I] := Dest[I] XOR Src[I]
```

Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe. *Opcodes*

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F DB /r	66[0F DB /r	PAND s, s/m	SIMD	SIMD / Mem
0F DF /r	66[0F DF /r	PANDN s, s/m	SIMD	SIMD / Mem
0F EB /r	66[0F EB /r	POR s, s/m	SIMD	SIMD / Mem
0F EF /r	66[0F EF /r	PXOR s, s/m	SIMD	SIMD / Mem

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PSLLW, PSLLD, PSLLQ — MMX (SSE2)
PSLLDQ — SSE2
PSRLW, PSRLD, PSRLQ — MMX (SSE2)
PSRLDQ — SSE2
PSRAW, PSRAD — MMX (SSE2)

Funktion

Verschiebe die Bits in gepackten Strukturen word-, doubleword-, quadword- oder doublequadwordweise arithmetisch oder logisch nach links oder rechts (»packed shift left/right arithmetically/logically words/doublewords/quadwords/ doublequadwords«).

Operation

```
IF (OperandSize = 128) THEN
   F := 2
ELSE                    ; (OperandSize = 64)
   F := 1
IF (ScalarSize(Operand) = Word) THEN
   N := 4
   S := 16
ELSEIF (ScalarSize(Operand) = DoubleWord) THEN
   N := 2
   S := 32
ELSE                    ; (ScalarType = QuadWord)
   N := 1
   S := 64
IF (Src ≥ S) THEN
   Dest := 0            ; alle Bits löschen
ELSE                    ; (Count < S)
   Count := Src
   FOR (I = 0) TO (I = F - 1) DO
      FOR (J = 0) TO (J = N - 1)
         B := I * N * S + J * S
         IF (Direction = left) THEN
            ; Bits shiften
            FOR (K = S - 1) DOWNTO (K = Count) DO
               Dest[B + K] := Dest[B + K - Count]
            ; Lücken auffüllen
            FOR (K = Count – 1) DOWNTO (K = 0) DO
               Dest[B + K] := 0
         ELSE           ; (Direction = right)
            IF (shift type = arithmetically) THEN
               Sign := Dest[B + Size – 1]
            ELSE        ; (shift type = logically)
               Sign := 0
            ; Bits shiften
            FOR (K = 0) TO (K = (Size - 1) – Count) DO
               Dest[B + K] := Dest[B + K + Count]
            ; Lücken auffüllen
            FOR (K = Size - Count) TO (K = Size - 1) DO
               Dest[B + K] := Sign
```

Opcodes Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F F1 /r	66[0F F1 /r	PSLLW s, s/m	SIMD	SIMD / Mem
0F 71 /6	66[0F 71 /6	PSLLW s, c8	SIMD	Const8
0F F2 /r	66[0F F2 /r	PSLLD s, s/m	SIMD	SIMD / Mem
0F 72 /6	66[0F 72 /6	PSLLD s, c8	SIMD	Const8
0F F3 /r	66[0F F3 /r	PSLLQ s, s/m	SIMD	SIMD / Mem
0F 73 /6	66[0F 73 /6	PSLLQ s, c8	SIMD	Const8
	66[0F 73 /7	PSLLDQ x, c8	XMM	Const8
0F D1 /r	66[0F D1 /r	PSRLW s, s/m	SIMD	SIMD / Mem
0F 71 /2	66[0F 71 /2	PSRLW s, c8	SIMD	Const8
0F D2 /r	66[0F D2 /r	PSRLD s, s/m	SIMD	SIMD / Mem
0F 72 /2	66[0F 72 /2	PSRLD s, c8	SIMD	Const8
0F D3 /r	66[0F D3 /r	PSRLQ s, s/m	SIMD	SIMD / Mem
0F 73 /2	66[0F 73 /2	PSRLQ s, c8	SIMD	Const8
	66[0F 73 /3	PSRLDQ x, c8	XMM	Const8
0F E1 /r	66[0F E1 /r	PSRAW s, s/m	SIMD	SIMD / Mem
0F 71 /4	66[0F 71 /4	PSRAW x, c8	SIMD	Const8
0F E2 /r	66[0F D2 /r	PSRAD s, s/m	SIMD	SIMD / Mem
0F 72 /4	66[0F 72 /4	PSRAD x, c8	SIMD	Const8

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PCMPEQB, PCMPEQW, PCMPEQD MMX (SSE2)
PCMPGTB, PCMPGTW, PCMPGTD MMX (SSE2)

Funktion Vergleiche zwei gepackte Strukturen byte-, word-, doublewordweise darauf, ob sie gleich sind oder eine größer als die andere ist (»packed compare if equal/greater byte wise/word wise/doubleword wise«).

Operation

```
IF (OperandSize = 128) THEN
   F := 2
ELSE             ;  (OperandSize = 64)
   F := 1
IF (ScalarSize(Operand) = Byte, ShortInt) THEN
   N := 8
   S := 8
ELSEIF (ScalarSize(Operand) = Word, SmallInt) THEN
   N := 4
   S := 16
ELSE             ; (ScalarSize(Operand) = DoubleWord, LongInt)
   N := 2
   S := 32
FOR (I = 0) TO (I = F * N - 1) DO
   U := (I + 1) * S – 1  ; höchstes Bit
   L := I * S            ; niederstes Bit
   IF (Vergleich = EQ) THEN
      IF (Dest[U..L] = Src[U..L]) THEN
         C := 1
      ELSE
         C := 0
      FOR J := L TO U DO
         Dest[J] := C
   ELSE                  ; (Vergleich = GT)
      IF (Dest[U..L] > Src[U..L]) THEN
         C := 1
      ELSE
         C := 0
      FOR J := L TO U DO
         Dest[J] := C
```

Opcodes Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 74 /r	66[0F 74 /r	PCMPEQB s, s/m	SIMD	SIMD / Mem
0F 75 /r	66[0F 75 /r	PCMPEQW s, s/m	SIMD	SIMD / Mem
0F 76 /6	66[0F 76 /r	PCMPEQD s, s/m	SIMD	SIMD / Mem
0F 64 /r	66[0F 64 /r	PCMPGTB s, s/m	SIMD	SIMD / Mem
0F 65 /r	66[0F 65 /r	PCMPGTW s, s/m	SIMD	SIMD / Mem
0F 66 /r	66[0F 66 /r	PCMPGTD s, s/m	SIMD	SIMD / Mem

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PACKUSWB — MMX (SSE2)
PACKSSWB, PACKSSDW — MMX (SSE2)
PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ — MMX (SSE2)
PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ — MMX (SSE2)

Funktion

Konvertiere vorzeichenbehaftete Integer zweier gepackter Strukturen in vorzeichenbehaftete oder vorzeichenlose Integers geringerer Datenbreite und packe sie in einer gepackten Struktur zusammen (»pack signed words/doublewords [un]signed to bytes/words«).

Mische abwechslungsweise die Integers zweier gepackter Strukturen aus den unteren oder oberen Hälften in einer gepackten Struktur (»packed unpack high/low bytes/words/doublewords in words/doublewords/quadwords«).

Operation

```
IF (OperandSize = 128) THEN
   F := 2
ELSE             ;  (OperandSize = 64)
   F := 1
IF (ScalarSize(Operand) = Byte, ShortInt) THEN
   N := 8
   S := 8
ELSEIF (ScalarSize(Operand) = Word, SmallInt) THEN
   N := 4
   S := 16
ELSE             ; (ScalarSize(Operand) = DoubleWord, LongInt)
   N := 2
   S := 32
 IF (Operation = PACK) THEN
   IF (DestFormat = signed) THEN
      OP := ConvertAndStaurateSigned
   ELSE          ; (DestFormat = unsigned)
      OP := ConvertAndSaturateUnsigned
   FOR (I = 0) TO (I = F * N - 1) DO
      U := (I + 1) * S – 1          ; high bit source operands
      L := I * S                    ; low bit source operands
      U1 := (I + 1) * (S DIV 2) – 1 ; high bit dest lower half
      L1 := I * (S DIV 2)           ; low bit dest lower half
      U2 := (F * N * S) DIV 2 + U1  ; high bit dest upper half
      L2 := (F * N * S) DIV 2 + L1  ; low bit dest upper half
      Dest[U1..L1] := OP(Dest[U..L])
      Dest[U2..L2] := OP(Src[U..L])
ELSE                   ; (Operation = UNPACK)
   IF (Source = lower half of structure) THEN
     B := 0
   ELSE                ; (source 0 upper half of structure)
     B := (F * N * S) DIV 2
   FOR (I = 0) TO (I = F * (N DIV 2) – 1) DO
      U := B + (I + 1) * S – 1      ; high bit source operands
      L := B + I * S                ; low bit source operands
      U1 := (I * S) + U             ; high bit dest lower half
      L1 := (I * S) + L             ; low bit dest lower half
      U2 := ((I + 1) * S) + U       ; high bit dest upper half
      L2 := ((I + 1) * S) + L       ; low bit dest upper half
   Dest[U1..L1] := Dest[U..L]
   Dest[U2..L2] := Src[U..L]
```

Opcodes

Als Zieloperand kommt ein MMX- oder XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- oder XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	**XMM (128 Bit)**		**Dest**	**Src**
`0F 67 /r`	`66[ 0F 67 /r`	PACKUSWB s, s/m	SIMD	SIMD / Mem
`0F 63/r`	`66[ 0F 63 /r`	PACKSSWB s, s/m	SIMD	SIMD / Mem
`0F 6B /r`	`66[ 0F 6B /r`	PACKSSDW s, s/m	SIMD	SIMD / Mem
`0F 68 /r`	`66[ 0F 68 /r`	PUNPCKHBW s, s/m	SIMD	SIMD / Mem
`0F 69 /r`	`66[ 0F 69 /r`	PUNPCKHWD s, s/m	SIMD	SIMD / Mem
`0F 6A /r`	`66[ 0F 6A /r`	PUNPCKHDQ s, s/m	SIMD	SIMD / Mem
`0F 60 /r`	`66[ 0F 60 /r`	PUNPCKLBW s, s/m	SIMD	SIMD / Mem
`0F 61 /r`	`66[ 0F 61 /r`	PUNPCKLWD s, s/m	SIMD	SIMD / Mem
`0F 62 /r`	`66[ 0F 62 /r`	PUNPCKLDQ s, s/m	SIMD	SIMD / Mem

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

MOVD, MOVQ	**MMX (SSE2)**
MOVDQA, MOVDQU	**SSE2**
MOVQ2DQ, MOVDQ2Q	**SSE2**
MOVNTQ	**SSE (SSE2)**
MOVNTDQ	**SSE2**

Funktion Kopiere ein DoubleWord aus einem MMX- bzw. XMM-Register in eine Speicherstelle oder ein Allzweckregister oder umgekehrt (»move doubleword«), ein QuadWord aus einem MMX- bzw. XMM-Register oder einer Speicherstelle in ein MMX- bzw. XMM-Register oder eine Speicherstelle (»move quadword«), oder ein DoubleQuadWord aus einem XMM-Register oder einer Speicherstelle in ein XMM-Register oder eine Speicherstelle (»mov doublequadword aligned«, »move doublequadword unaligned«).

Tausche ein Quadword zwischen MMX- und XMM-Register aus (»move quadword to doublequadword«, »mov doublequadword to quadword«).

Schreibe ein QuadWord oder ein OctelWord (DoubleQuadWord) aus dem MMX- oder XMM-Register unter Umgehung des Cache in den Speicher (»move using non-temporal hint (double)quadword«)

Operation

```
IF (Instruction = MOVD) THEN
   IF (Destination = MMX, XMM) THEN
      Dest := 0        ; Register loschen, nur low doubleword
      Dest[031..000] := Src[031..000]
   ELSE                ; (Source = MMX, XMM)
      Dest[031..000] := Src[031..000]
ELSEIF (Instruction = MOVQ) THEN
   IF (Destination = MMX, XMM) THEN
      IF (Source = Memory) THEN
         Dest := 0      ; Register nur löschen, wenn Src = Mem
      Dest[063..000] := Src[063..000]
   ELSE                ; (Source = MMX, XMM)
      Dest[063.000] := Src[063..000]
ELSEIF (Instruction = MOVDQU) THEN
   Dest[127..000] := Src[127..000]
ELSEIF (Instruction = MOVDQA) THEN
   IF (Src or Dest unaligned) THEN
      #GP(0)
   ELSE
      Dest[127..000] := Src[127..000]
```

```
ELSEIF (Instruction = MOVDQ2Q) THEN
   Dest[063..000] := Src[063..000]
ELSEIF (Instruction = MOVQ2DQ) THEN
   Dest[063..000] := Src[063..000]
   Dest[120..064] := 0
ELSE                    ; (Instruction = MOVNTQ, MOVNTDQ)
   Dest := Src
```

Opcodes

Neben dem Austausch von Daten zwischen MMX- bzw. XMM-Register mit anderen MMX- bzw. XMM-Registern oder einer Speicherstelle ist auch der Austausch zwischen MMX- und XMM-Register möglich.

Opcode			Operanden	
MMX (64 Bit)	**XMM (128 Bit)**	**Mnemonic**	**Dest**	**Src**
0F 6E /r	66[0F 6E /r	MOVD s, r/m	SIMD	Reg / Mem
0F 7E /r	66[0F 7E /r	MOVD r/m, s	Reg /Mem	SIMD
0F 6F /r		MOVQ mm, mm/m	MMX	MMX / Mem
0F 7F /r		MOVQ mm/m, mm	MMX / Mem	MMX
	F3[0F 7E /r	MOVQ x, x/m	XMM	XMM / Mem64
	66[0F D6 /r	MOVQ x/m, x	XMM / Mem64	XMM
	66[0F 6F /r	MOVDQA x, x/m	XMM	XMM / Mem
	66[0F 7F /r	MOVDQA x/m, x	XMM / Mem	XMM
	F3[0F 6F /r	MOVDQU x, x/m	XMM	XMM / Mem
	F3[0F 7F /r	MOVDQU x/m, x	XMM / Mem	XMM
	F3[0F D6 /r	MOVQ2DQ x, mm	XMM	MMX
	F2[0F D6 /r	MOVQ2DQ mm, x	MMX	XMM
0F E7 /r		MOVNTQ m, mm	Mem	MMX
	66[0F E7 /r	MOVNTDQ m, x	Mem	XMM

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 20, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

MASKMOVQ SSE (SSE2)
MASKMOVDQU SSE

Funktion Kopiere ausgewählte Komponenten einer gepackten Struktur von der Quelle in das Ziel. Eine Komponente gilt als ausgewählt, wenn in ihrer Maske das MSB (most significant bit; höchstwertige Bit) gesetzt ist (»move bytes from/to quadword/ doublequadword as directed by mask«).

Operation
```
IF (OperandSize = 128) THEN
   N := 16
ELSE                    ; (OperandSize = 64)
   N := 8
FOR (I = 0) TO (I = N – 1) DO
   U := (I + 1) * 8 – 1
   L := (I * 8)
   IF (Src2[U] = 1 THEN
      DS:(E)DI[U..L] := Src1[U..L]
```

Opcodes Die Befehle benutzen neben den explizit angegebenen Quelloperanden auch einen impliziten Zieloperanden, der die Adresse des Ziels des Datenstroms angibt.

Opcode		Mnemonic	Operanden		
MMX (64 Bit)	XMM (128 Bit)		Dest	Src	Scr2
0F F7 /r		MASKMOVQ mm, mm	DS:(E)DI	MMX	MMX
	66[0F F7 /r	MASKMOVDQU x, x	DS:(E)DI	XMM	XMM

Die grau unterlegten Operanden sind implizit an- und vorgegeben.

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. *Exceptions*

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	-
#GP	8	0	9	0	9
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 15	-	6, 13, 14	-	6, 13, 14

PMOVMSKB MMX (SSE2)

Erzeuge aus den Vorzeichen (höchstwertigen Bits, most significant bits, MSBs) der Komponenten einer gepackten Struktur ein Maskenbyte (»packed move mask byte«). *Funktion*

```
Dest := 0                                                    Operation
IF (OperandSize = 128) THEN
   N := 16
ELSE                    ; (OperandSize = 16)
   N := 8
FOR (I = 0) TO (I = N - 1) DO
  Dest[I] := Src[(I + 1) * 8 – 1]
```

Als Zieloperand kommt ein Allzweck-Register in Frage, die Quelle ist ein MMX- oder XMM-Register. *Opcodes*

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F D7 /r	66[0F D7 /r	PMOVMSKB r, s	Reg	SIMD

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen *Exceptions*

Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#UD	6, 13, 14, 15	-	6, 13, 14	-	6, 13, 14

PEXTRW SSE (SES2)
PINSRW SSE (SSE2)

Funktion Kopiere ein spezifiziertes Word aus dem Quell- in den Zieloperanden (»packed extract word to register«, »packed insert word form register«).

Operation

```
IF (OperandSize = 128) THEN
   W := Src2 AND $07
ELSE                    ; (OperandSize = 64)
   W := Src2 AND $03
U := (W + 1) * 16 - 1
L := W * 16
IF (Instruction = PEXTRW) THEN
   Dest[15..00] := Src[U..L]
   Dest[31..16] := 0
ELSE                    ; (Instruction = PINSRW)
   Dest[U..L] := Src[15..00]
```

Opcodes Neben einem MMX- oder XMM-Register ist auch ein Allzweckregister (oder ggf. eine Speicherstelle) sowie eine 8-Bit-Konstante als Operand vorgesehen.

Opcode		Mnemonic	Operanden		
MMX (64 Bit)	XMM (128 Bit)		Dest	Src	Src2
0F C5 /r	66[0F C5 /r	PEXTRW r, s, c8	Reg	SIMD	Const8
0F C4 /r	66[0F C4 /r	PINSRW s, r/m, c8	SIMD	Reg / Mem	Const8

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PSHUFW SSE
PSHUFD SSE2
PSHUFLW, PSHUFHW SSE2

Funktion

Kopiere ausgewählte Daten aus dem Quell- in den Zieloperanden (»packed shuffle word/doubleword/low word/high word«).

Operation

```
IF (OperandSize = 128) AND (Instruction ≠ PSHFHW,PSHUFLW) THEN
   F := 2
ELSE               ; (OperandSize = 64 oder PSHUFHW, PSHUFLW)
   F := 1
IF (ScalarSize(Operand) = Word, SmallInt) THEN
   N := 4
   S := 16
ELSE               ; ScalarSize(Operand) = DoubleWord,LongInt)
   N := 2
   S := 32
IF (Instruction = PSHUFHW) THEN
   B := 64
ELSEIF             ; (Instruction = PSHUFW, PSHUFD, PSHUFLW)
   B := 0
FOR (I = 0) TO (I = F * N - 1) DO
   ; Bitbereich des Zieloperanden
   U := B + (I + 1) * S – 1  ; höchstes Bit
   L := B + (I * S)            ; niederstes Bit
   ; Bitbereich des Index
   UI := (I + 1) * 2 – 1
   LI := I * 2
   Index := Src2[UI..LI]
   ; Bitbereich des Quelloperanden
   US := B + (Index + 1) * S – 1
   LS := B + (Index * S)
   Dest[U..L] := Src1[US..LS]
```

Opcodes Als Zieloperand kommt bei PSHUFW ein MMX-, bei den anderen Befehlen ein XMM-Register in Frage, die Quelle ist ebenfalls ein MMX- bzw. XMM-Register oder eine Speicherstelle der entsprechenden Größe sowie eine 8-Bit-Konstante.

Opcode		Mnemonic	Operanden		
MMX (64 Bit)	XMM (128 Bit)		Dest	Src	Src2
0F 70 /r		PSHUFW mm, mm/m, c8	MMX	MMX / Mem	Const8
	66[0F 70 /r	PSHUFD x, x/m, c8	XMM	XMM / Mem	Const8
	F3[0F 70 /r	PSHUFHW x, x/m, c8	XMM	XMM / Mem	Const8
	F2[0F 70 /r	PSHUFLW x, x/m, c8	XMM	XMM / Mem	Const8

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9, 68	0	9, 68
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

PSWAPD — 3D-Now!-X

Funktion Kopiere zwei DoubleWords aus einem QuadWord in ein anderes QuadWord und vertausche die Reihenfolge (»packed swap doublewords«).

Operation
```
Temp := Src            ; erforderlich, wenn Dest = Src!
Dest[31..00] := Temp[63..32]
Dest[63..00] := Temp[31..00]
```

Opcodes Als Zieloperand kommt lediglich ein MMX-Register in Frage, die Quelle ist ebenfalls ein MMX-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 0F]BB		PSWAPD mm, mm/m	MMX	MMX / Mem64

Der Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6	-	6	-	6

4.2 SIMD-Real-Befehle

ADDPS, ADDSS **SSE**
ADDPD, ADDSD **SSE2**

SUBPS, SUBSS **SSE**
SUBPD, SUBSD **SSE2**

DIVPS, DIVSS **SSE**
DIVPD, DIVSD **SSE2**

MULPS, MULSS **SSE**
MULPD, MULSD **SSE2**

Funktion

Addiere, subtrahiere, dividiere oder multipliziere ScalarSingles, PackedSingles, ScalarDoubles oder PackedDoubles (»add/subtract/multiply/divide scalar/packed single/double«).

Operation

```
IF (DataType = Scalar) THEN
   N := 1
   IF (OperandSize = Single) THEN
      S := 32
```

```
   ELSE                ; (OperandSize = Double)
      S := 64
ELSE                   ; (DataType = Packed)
   IF (OperandSize = Single) THEN
      N := 4
      S := 32
   ELSE                ; (OperandSize = Double)
      N := 2
      S := 64
FOR (I = 0) TO (I = N – 1) DO
   U := (I + 1) * S – 1
   L := I * S
   IF (Operation = Addition) THEN
      Dest[U..L] := Dest[U..L] + Src[U..L]
   ELSEIF (Operation = Subtraction) THEN
      Dest[U..L] := Dest[U..L] - Src[U..L]
   ELSEIF (Operation = Multiplication) THEN
      Dest[U..L] := Dest[U..L] * Src[U..L]
   ELSE                ; (Operation = Division) THEN
      Dest[U..L] := Dest[U..L] / Src[U..L]
```

Opcodes Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	**XMM (128 Bit)**		**Dest**	**Src**
	0F 58 /r	ADDPS x, x/m	XMM	XMM / Mem128
	F3[0F 58 /r	ADDSS x, x/m	XMM	XMM / Mem32
	66[0F 58 /r	ADDPD x, x/m	XMM	XMM / Mem128
	F2[0F 58 /r	ADDSD x, x/m	XMM	XMM / Mem64
	0F 5C /r	SUBPS x, x/m	XMM	XMM / Mem128
	F3[0F 5C /r	SUBSS x, x/m	XMM	XMM / Mem32
	66[0F 5C /r	SUBPD x, x/m	XMM	XMM / Mem128
	F2[0F 5C /r	SUBSD x, x/m	XMM	XMM / Mem64
	0F 5E /r	DIVPS x, x/m	XMM	XMM / Mem128
	F3[0F 5E /r	DIVSS x, x/m	XMM	XMM / Mem32
	66[0F 5E /r	DIVPD x, x/m	XMM	XMM / Mem128
	F2[0F 5E /r	DIVSD x, x/m	XMM	XMM / Mem64

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	0F 59 /r	MULPS x, x/m	XMM	XMM / Mem128
	F3[0F 59 /r	MULSS x, x/m	XMM	XMM / Mem32
	66[0F 59 /r	MULPD x, x/m	XMM	XMM / Mem128
	F2[0F 59 /r	MULSD x, x/m	XMM	XMM / Mem64

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Tabelle 5.14 bis Tabelle 5.20 ab Seite 295 listen die Gründe für die XMM-Exceptions auf.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1, 7	-	1	1	1	-

PFADD — 3D-Now!
PFSUB, PFSUBR — 3D-Now!
PFMUL — 3D-Now!

Funktion

Addiere, subtrahiere oder multipliziere ShortPackedSingles (»packed floating-point add/subtract/multiply [reverse]«).

Operation

```
IF (Instruction = FPADD) THEN
   Dest[31..00] := Dest[31..00] + Src[31..00]
   Dest[63..32] := Dest[63..32] + Src[63..32]
```

```
ELSEIF (Instruction = FPSUB) THEN
   Dest[31..00] := Dest[31..00] - Src[31..00]
   Dest[63..32] := Dest[63..32] - Src[63..32]
ELSEIF (Instruction = FPSUBR) THEN
   Dest[31..00] := Src[31..00] - Dest[31..00]
   Dest[63..32] := Src[63..32] - Dest[63..32]
ELSE                    ; (Instruction = FPMUL)
   Dest[31..00] := Dest[31..00] * Src[31..00]
   Dest[63..32] := Dest[63..32] * Src[63..32]
```

Opcodes Als Zieloperand kommt nur ein MMX-Register in Frage, die Quelle ist ebenfalls ein MMX-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 0F]9E		PFADD mm, mm/m	MMX	MMX / Mem64
0F 0F]9A		PFSUB mm, mm/m	MMX	MMX / Mem64
0F 0F]AA		PFSUBR mm, mm/m	MMX	MMX / Mem64
0F 0F]B4		PFMUL mm, mm/m	MMX	MMX / Mem64

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6	-	6	-	6

MAXPS, MAXSS — SSE
MAXPD, MAXSD — SSE2
MINPS, MINSS — SSE
MINPD, MINSD — SSE2

Funktion

Bilde die Maxima bzw. Minima aus skalaren oder gepackten SingleReals oder DoubleReals (»maximum/minimum of scalar/packed singles/doubles«).

Operation

```
IF (DataType = Scalar) THEN
   N := 1
   IF (OperandSize = Single) THEN
      S := 32
   ELSE                ; (OperandSize = Double)
      S := 64
ELSE                   ; (DataType = Packed)
   IF (OperandSize = Single) THEN
      N := 4
      S := 32
   ELSE                ; (OperandSize = Double)
      N := 2
      S := 64
FOR (I = 0) TO (I = N – 1) DO
   U := (I + 1) * S – 1
   L := I * S
   IF (Operation = Maximum) THEN
      IF ((Dest[U..L] = 0.0) AND (Src[U..L] = 0.0))
      OR (Dest[U..L] = sNaN)
      OR (Src[U..L] = sNaN)
      OR (Dest[U..L] ≤ Src[U..L]) THEN
         Dest[U..L] := Src[U..L]
      ELSE
         Dest[U..L] := Dest[U..L]
   ELSE                ; (Operation = Minimum) THEN
      IF ((Dest[U..L] = 0.0) AND (Src[U..L] = 0.0))
      OR (Dest[U..L] = sNaN)
      OR (Src[U..L] = sNaN)
      OR (Dest[U..L] ≥ Src[U..L]) THEN
         Dest[U..L] := Src[U..L]
      ELSE
         Dest[U..L] := Dest[U..L]
```

Opcodes Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	0F 5F /r	MAXPS x, x/m	XMM	XMM / Mem128
	F3[0F 5F /r	MAXSS x, x/m	XMM	XMM / Mem32
	66[0F 5F /r	MAXPD x, x/m	XMM	XMM / Mem128
	F2[0F 5F /r	MAXSD x, x/m	XMM	XMM / Mem64
	0F 50 /r	MINPS x, x/m	XMM	XMM / Mem128
	F3[0F 50 /r	MINSS x, x/m	XMM	XMM / Mem32
	66[0F 50 /r	MINPD x, x/m	XMM	XMM / Mem128
	F2[0F 50 /r	MINSD x, x/m	XMM	XMM / Mem64

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Tabelle 5.14 bis Tabelle 5.20 ab Seite 295 listen die Gründe für die XMM-Exceptions auf.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	3	9	-	-	-	-	-

PFMAX 3D-Now!
PFMIN 3D-Now!

Funktion

Bilde die Maxima bzw. Minima aus ShortPackedSingles (»packed floating-point maximum/minimum«).

Operation

```
IF (Instruction = PFMAX) THEN
   IF (Dest[31..00] < Src[31..00]) THEN
      Dest[031..00] := Src[31..00]
   ELSE
      Dest[31..00] := Dest[31..00]
   IF (Dest[63..32] < Src[63..32]) THEN
      Dest[063..32] := Src[63..32]
   ELSE
      Dest[63..32] := Dest[63..32]
ELSE                    ; (Instruction = PFMIN)
   IF (Dest[31..00] > Src[31..00]) THEN
      Dest[031..00] := Src[31..00]
   ELSE
      Dest[31..00] := Dest[31..00]
   IF (Dest[63..32] > Src[63..32]) THEN
      Dest[063..32] := Src[63..32]
   ELSE
      Dest[63..32] := Dest[63..32]
```

Opcodes

Als Zieloperand kommt nur ein MMX-Register in Frage, die Quelle ist ebenfalls ein MMX-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 0F]A4		PFMAX mm, mm/m	MMX	MMX / Mem64
0F 0F]94		PFMIN mm, mm/m	MMX	MMX / Mem64

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen

Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6	-	6	-	6

SQRTPS, SQRTSS SSE
SQRTPD, SQRTSD SSE2

Funktion Bilde die Quadratwurzel aus skalaren oder gepackten SingleReals oder DoubleReals (»square root of scalar/packed singles/doubles«).

Operation

```
IF (DataType = Scalar) THEN
   N := 1
   IF (OperandSize = Single) THEN
      S := 32
   ELSE                ; (OperandSize = Double)
      S := 64
ELSE                   ; (DataType = Packed)
   IF (OperandSize = Single) THEN
      N := 4
      S := 32
   ELSE                ; (OperandSize = Double)
      N := 2
      S := 64
FOR (I = 0) TO (I = N – 1) DO
   U := (I + 1) * S – 1
   L := I * S
Dest[U..L] := SQRT(Src[U..L])
```

Opcodes Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	0F 51 /r	SQRTPS x, x/m	XMM	XMM / Mem128
	F3[0F 51 /r	SQRTSS x, x/m	XMM	XMM / Mem32
	66[0F 51 /r	SQRTPD x, x/m	XMM	XMM / Mem128
	F2[0F 51 /r	SQRTSD x, x/m	XMM	XMM / Mem64

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Tabelle 5.14 bis Tabelle 5.20 ab Seite 295 listen die Gründe für die XMM-Exceptions auf.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	2	1, 3	-	-	1	-	-

RCPPS, RCPSS SSE
RSQRTPS, RSQRTSS SSE

Funktion

Bilde den Kehrwert bzw. die reziproke Quadratwurzel aus skalaren oder gepackten SingleReals (»reciprocal [square root] of scalar/packed singles«).

Operation

```
IF (DataType = Scalar) THEN
   N := 1
   IF (OperandSize = Single) THEN
      S := 32
   ELSE                  ; (OperandSize = Double)
      S := 64
ELSE                     ; (DataType = Packed)
   IF (OperandSize = Single) THEN
      N := 4
      S := 32
   ELSE                  ; (OperandSize = Double)
      N := 2
      S := 64
FOR (I = 0) TO (I = N – 1) DO
   U := (I + 1) * S – 1
   L := I * S
   IF (Operation = Reciprocal) THEN
      Dest[U..L] := Approximate(1.0 / Src[U..L])
   ELSE                  ; (Operation = recip. square root) THEN
      Dest[U..L] := Approximate(1.0 / SQRT(Src[U..L]))
```

Opcodes Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	0F 53 /r	RCPPS x, x/m	XMM	XMM / Mem128
	F3[0F 53 /r	RCPSS x, x/m	XMM	XMM / Mem64
	0F 52 /r	RSQRTPS x, x/m	XMM	XMM / Mem128
	F3[0F 52 /r	RSQRTSS x, x/m	XMM	XMM / Mem64

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

PFRCP, PFRCPIT1, PFRCPIT2 3D-Now!
PFRSQRT, PFRSQIT1 3D-Now!

Funktion

Bilde den Kehrwert bzw. die reziproke Quadratwurzel von Short-PackedSingles (»packed floating-point reciprocal [square root] «).

Operation

```
PFRCP:
Operation: Division mittels ROM-basierter Divisionstabelle
Dest[31..00] := Dest[31..00] / Src[31..00]
Dest[63..32] := Dest[31..00] / Src[31..00] ; kein Druckfehler!
Genauigkeit: 14 – 15 Bits

PFRCPIT1:
Operation: Erster Schritt einer Iteration nach Newton-Raphson
Formel:    Z_{i+1} := Z_i · (2 – b · Z_i)
Bedingung: Startwert aus PFRCP, Ausgabe für PFRCPIT2

PFRSQRT:
Operation: Schätzwertbildung für die reziproke Quadratwurzel
Dest[31..00] := Dest[31..00] / Src[31..00]
Dest[63..32] := Dest[31..00] / Src[31..00] ; kein Druckfehler!
Genauigkeit: 14 – 15 Bits

PFRSQIT1:
Operation: Erster Schritt einer Iteration nach Newton-Raphson
Formel:    Z_{i+1} := 0.5 · Z_i · (3 – b · Z_i^2)
Bedingung: Startwert aus PFRSQRT, Ausgabe für PFRCPIT2
```

```
PFRCPIT2:
Operation: Abschluss der Iterationen nach Newton-Raphson
Genauigkeit: 24 Bits
```

Opcodes Als Zieloperand kommt nur ein MMX-Register in Frage, die Quelle ist ebenfalls ein MMX-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	**XMM (128 Bit)**		**Dest**	**Src**
0F 0F]96		PFRCP mm, mm/m	MMX	MMX / Mem64
0F 0F]A6		PFRCPIT1 mm, mm/m	MMX	MMX / Mem64
0F 0F]B6		PFRCPIT2 mm, mm/m	MMX	MMX / Mem64
0F 0F]97		PFRSQRT mm, mm/m	MMX	MMX / Mem64
0F 0F]A7		PFRSQIT1 mm, mm/m	MMX	MMX / Mem64

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	**ErrorCode**	**Grund**	**ErrorCode**	**Grund**
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6	-	6	-	6

PFACC 3D-Now!
PFNACC 3D-Now!-X
PFPNACC 3D-Now!-X

Funktion

Akkumuliere ShortPackedSingles (»packed floating-point [positive] [negative] accumulation«).

Operation

```
Temp := Src
IF (Instruction = PFACC) THEN
   Dest[31..00] := Dest[31..00] + Dest[63..32]
   Dest[63..32] := Temp[31..00] + Temp[63..00]
ELSEIF (Instruction = PFNACC) THEN
   Dest[31..00] := Dest[31..00] - Dest[63..32]
   Dest[63..32] := Temp[31..00] - Temp[63..00]
ELSE                    ; (Instruction = PFPNACC)
   Dest[31..00] := Dest[31..00] - Dest[63..32]
   Dest[63..32] := Temp[31..00] + Temp[63..00]
```

Opcodes

Als Zieloperand kommt nur ein MMX-Register in Frage, die Quelle ist ebenfalls ein MMX-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 0F]AE		PFACC mm, mm/m	MMX	MMX / Mem64
0F 0F]8A		PFNACC mm, mm/m	MMX	MMX / Mem64
0F 0F]8E		PFPNACC mm, mm/m	MMX	MMX / Mem64

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#MF	2	-	2	-	2

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6	-	6	-	6

CMPPS, CMPSS SSE
CMPPD, CMPSD SSE2

Funktion Vergleiche zwei skalare oder gepackte SingleReals oder DoubleReals miteinander (»compare scalar/packed singles/doubles«).

Operation

```
IF (DataType = Scalar) THEN
   N := 1
   IF (OperandSize = Single) THEN
      S := 32
   ELSE                ; (OperandSize = Double)
      S := 64
ELSE                   ; (DataType = Packed)
   IF (OperandSize = Single) THEN
      N := 4
      S := 32
   ELSE                ; (OperandSize = Double)
      N := 2
      S := 64
CASE Src2 OF
   0 : OP := equal
   1 : OP := lower than
   2 : OP := lower than or equal
   3 : OP := unordered
   4 : OP := not equal
   5 : OP := not lower than
   6 : OP := not (lower than or equal)
   7 : OP := ordered
FOR (I = 0) TO (I = N – 1) DO
   U := (I + 1) * S – 1
   L := I * S
   IF (Dest[U..L] OP Src[U..L]) THEN
      FOR (J = L) TO (J = U) DO
         Dest[J] := 1
   ELSE                ; (comparison = FALSE)
      FOR (J = L) TO (J = U) DO
         Dest[J] := 0
```

Opcodes

Als Zieloperand kommt nur ein XMM-Register in Frage, Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe und eine 8-Bit-Konstante.

Opcode		Mnemonic	Operanden		
MMX (64 Bit)	XMM (128 Bit)		Dest	Src	Src2
	`0F C2 /r`	CMPPS x, x/m, c	XMM	XMM / Mem64	Const8
	`F3[ 0F C2 /r`	CMPSS x, x/m, c	XMM	XMM / Mem64	Const8
	`66[ 0F C2 /r`	CMPPD x, x/m, c	XMM	XMM / Mem64	Const8
	`F2[ 0F C2 /r`	CMPSD x, x/m, c	XMM	XMM / Mem64	Const8

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Tabelle 5.14 bis Tabelle 5.20 ab Seite 295 listen die Gründe für die XMM-Exceptions auf.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	3	14	-	-	-	-	-

Bemerkung

Bitte beachten Sie, dass das Mnemonic CMPSD auch bei den String-Befehlen verwendet wird. Dort wird es als »Erweiterung« des Befehls CMPS mit DoubleWord-Operanden verwendet. Einen tatsächlichen Konflikt bei der Nutzung der Mnemonics gibt es jedoch nicht, da der Programmierer aufgrund des Kontextes weiß, welcher Befehl gemeint ist, und der Assembler die korrekte Befehlssequenz aufgrund der Operanden feststellen kann.

PFCMPEQ 3D-Now!
PFCMPGE 3D-Now!
PFCMPGT 3D-Now!

Funktion Vergleiche zwei ShortPackedSingles miteinander (»packed floating-point compare equal/greater or equal/ greater«).

Operation

```
IF (Instruction = PFCMPEQ) THEN
   IF (Dest[31..00] = Src[31..00]) THEN
      Dest[31..00] := $FFFF_FFFF
   ELSE
      Dest[31.00] := $0000_0000
   IF (Dest[63..32] = Src[63..32]) THEN
      Dest[63..32] := $FFFF_FFFF
   ELSE
      Dest[63..32] := $0000_0000
ELSEIF (Instruction = PFCMPGE) THEN
   IF (Dest[31..00] ≥ Src[31..00]) THEN
      Dest[31..00] := $FFFF_FFFF
   ELSE
      Dest[31.00] := $0000_0000
   IF (Dest[63..32] ≥ Src[63..32]) THEN
      Dest[63..32] := $FFFF_FFFF
   ELSE
      Dest[63..32] := $0000_0000
ELSE                    ; (Instruction = PFCMPGT)
   IF (Dest[31..00] > Src[31..00]) THEN
      Dest[31..00] := $FFFF_FFFF
   ELSE
      Dest[31.00] := $0000_0000
   IF (Dest[63..32] > Src[63..32]) THEN
      Dest[63..32] := $FFFF_FFFF
   ELSE
      Dest[63..32] := $0000_0000
```

Opcodes Als Zieloperand kommt nur ein MMX-Register in Frage, die Quelle ist ebenfalls ein MMX-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 0F]B0		PFCMPEQ mm, mm/m	MMX	MMX / Mem64
0F 0F]90		PFCMPGE mm, mm/m	MMX	MMX / Mem64
0F 0F]A0		PFCMPGT mm, mm/m	MMX	MMX / Mem64

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6	-	6	-	6

COMISS SSE
COMISD SSE2
UCOMISS SSE
UCOMISD SSE2

Funktion

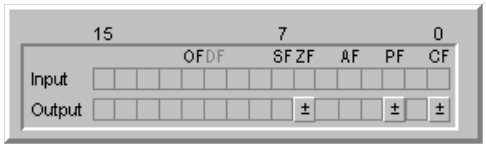
Vergleiche zwei skalare SingleReals oder DoubleReals gleicher oder verschiedener Ordnung miteinander (»[unordered] compare scalar singles/doubles and store result in integer flags«).

Operation

```
IF (OperandSize = Single) THEN
   S := 32
ELSE                ; (OperandSize = Double)
   S := 64
U := S – 1
L := 0
```

```
IF (Src1[U..L] > Src2[U..L]) THEN
   ZF := 0
   PF := 0
   CF := 0
ELSEIF (Src1[U..L] < Src2[U..L]) THEN
   ZF := 0
   PF := 0
   CF := 1
ELSE                              ; (Src1[U..L] = Src2[U..L])
   ZF := 1
   PF := 0
   CF := 0
IF (Src1[U..L] = sNaN,qNaN)
OR (Src2[U..L] = sNaN,qNaN) THEN
   ZF := 1
   PF := 1
   CF := 1
```

CPU-Statusflags



Die Ergebnisse des Vergleichs werden bei diesen Befehlen in den Statusflags der CPU zurückgegeben! Hierbei werden trotz der Verwendung vorzeichenbehafteter Realzahlen in den XMM-Registern die Flags für vorzeichenlose Integer verwendet (also CF anstelle von OF und SF!)

Bedingung	ZF	PF	CF
Src1 > Src2	0	0	0
Src1 < Src2	0	0	1
Src1 = Scr2	1	0	0
»unordered«, nicht vergleichbar	1	1	1

Opcodes

Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Src1	Src2
	0F 2F /r	COMISS x, x/m	XMM	XMM / Mem32
	66[0F 2F /r	COMISD x, x/m	XMM	XMM / Mem32
	0F 2E /r	UCOMISS x, x/m	XMM	XMM / Mem32
	66[0F 2E /r	UCOMISD x, x/m	XMM	XMM / Mem32

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben. Tabelle 5.14 bis Tabelle 5.20 ab Seite 295 listen die Gründe für die XMM-Exceptions auf.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	3	9	-	-	-	-	-

ANDPS	**SSE**
ANDPD	**SSE2**
ANDNPS	**SSE**
ANDNPD	**SSE2**
ORPS	**SSE**
ORPD	**SSE2**
XORPS	**SSE**
XORPD	**SSE2**

Funktion Bilde eine logische UND-, UND-NICHT-, ODER bzw. Exclusiv-ODER-Verknüpfung mit zwei gepackten SingleReals oder DoubleReals (»and/and not/or/xor packed singles/doubles«).

Operation
```
IF (OperandSize = Single) THEN
   N := 4
   S := 32
ELSE                ; (OperandSize = Double)
   N := 2
   S := 64
FOR (I = 0) TO (I = N – 1) DO
   U := (I + 1) * S – 1
   L := I * S
   FOR (J = L) TO (J = U) DO
      IF (Operation = AND) THEN
         Dest[J] := Dest[J] AND Src[J]
      ELSEIF (Operation = AND NOT) THEN
         Dest[J] := NOT(Dest[J]) AND Src[J]
      ELSEIF (Operation = OR) THEN
         Dest[J] := Dest[J] OR Src[J]
      ELSE               ; (Operation = XOR) THEN
         Dest[J] := Dest[J] XOR Src[J]
```

Opcodes Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	0F 54 /r	ANDPS x, x/m	XMM	XMM / Mem128
	66[0F 54 /r	ANDPD x, x/m	XMM	XMM / Mem128
	0F 55 /r	ANDNPS x, x/m	XMM	XMM / Mem128
	66[0F 55 /r	ANDNPD x, x/m	XMM	XMM / Mem128
	0F 56 /r	ORPS x, x/m	XMM	XMM / Mem128
	66[0F 56 /r	ORPD x, x/m	XMM	XMM / Mem128
	0F 57 /r	XORPS x, x/m	XMM	XMM / Mem128
	66[0F 57 /r	XORSD x, x/m	XMM	XMM / Mem128

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

CVTPS2PD, CVTSS2SD **SSE**
CVTPD2PS, CVTSD2SS **SSE**

CVTPS2PI, CVTSS2SI, CVTPD2PI, CVTSD2SI **SSE2**
CVTTPS2PI, CVTTSS2SI, CVTTPD2PI, CVTTSD2SI **SSE2**
CVTPI2PS, CVTSI2SS, CVTPI2PD, CVTSI2SD **SSE2**

CVTPS2DQ, CVTPD2DQ **SSE2**
CVTTPS2DQ, CVTTPD2DQ **SSE2**
CVTDQ2PS, CVTDQ2PD **SSE2**

Funktion Konvertiere PackedSingles in PackedDoubles und umgekehrt bzw. ScalarSingles in ScalarDoubles und umgekehrt (»convert packed/scalar singles/doubles to packed/scalar doubles/singles«). Hierzu gibt es folgende Beziehungen:

Real (XMM-Register)	Richtung	Real	
		ScalarDouble (XMM-Register)	PackedDouble (XMM-Register)
ScalarSingle	→	CVTSS2SD	-
	←	CVTSD2SS	-
PackedSingle	→	-	CVTPS2PD
	←	-	CVTPD2PS

Konvertiere PackedSingles/PackedDoubles in ShortPackedLongInts und umgekehrt bzw. ScalarSingles/ScalarDoubles in ScalarIntegers und umgekehrt (»convert [by truncation] packed/scalar doubles/singles/integers to packed/scalar integers/doubles/singles«).

Konvertiere PackedSingles/PackedDoubles in PackedLongInts und umgekehrt (»convert [by truncation] packed doubles/singles/doublequads to packed doublequads/singles/doubles«).

Dies ergibt folgende Beziehungen:

Real (XMM-Register)	Richtung	Real		
		CPU-Integer (Allzweckregister)	ShortPackedInteger (MMX-Register)	PackedInteger (XMM-Register)
ScalarSingle	→	CVTSS2SI CVTTSS2SI	-	-
	←	CVTSI2SS	-	-
PackedSingle	→	-	CVTPS2PI CVTTPS2PI	CVTPS2DQ CVTTPS2DQ
	←	-	CVTPI2PS	CVTDQ2PS
ScalarDouble	→	CVTSD2SI CVTTSD2SI	-	-
	←	CVTSI2SD	-	-
PackedDouble	→	-	CVTPD2PI CVTTPD2PI	CVTPD2DQ CVTTPD2DQ
	←	-	CVPTPI2PD	CVTDQ2PD

Operation

```
IF (Conversion = SingleToDouble) THEN
   Dest[063..000] := ConvertSingle2Double(Src[031..000])
   IF (DataType = packed) THEN
      Dest[127..064] := ConvertSingle2Double(Src[063..032])
   ELSE                ; (DataType = scalar)
      ; Dest[127..064] bleibt unverändert
ELSEIF (Conversion = DoubleToSingle) THEN
   Dest[031..000] := ConvertDouble2Single(Src[063..000])
   IF (DataType = packed) THEN
      Dest[063..032] := ConvertDouble2Single(Src[127..000])
      Dest[127..064] := 0
   ELSE                ; (DataType = scalar)
      ; Dest[127..032] bleibt unverändert
ELSEIF (Conversion = SingleToLongInt(MMX)) THEN
   IF (Truncate) THEN
      OP := ConvertSingle2LongIntAndTruncate
   ELSE
      OP := ConvertSingleToDWord
   Dest[031..000] := OP(Src[031..000])
   IF (DataType = packed) THEN
      Dest[063..032] := OP(Src[031..032])
ELSEIF (Conversion = DoubleToLongInt(MMX)) THEN
   IF (Truncate) THEN
      OP := ConvertDoubleLongIntAndTruncate
```

```
   ELSE
      OP := ConvertDoubleToLongInt
   Dest[031..000] := OP(Src[063..000])
   IF (DataType = packed) THEN
      Dest[063..032] := OP(Src[127..064])
ELSEIF (Conversion = LongInt(MMX)ToSingle) THEN
   Dest[031..000] := ConvertLongInt2Single(Src[031..000])
   IF (DataType = packed) THEN
      Dest[063..032] := ConvertLongInt2Single(Src[063..032])
      Dest[127..064] bleibt unverändert
   ELSE                ; (DataType = scalar)
      Dest[127..032] bleibt unverändert
ELSEIF (Conversion = LongInt(MMX)ToDouble) THEN
   Dest[063..000] := ConvertLongInt2Double(Src[031..000])
   IF (DataType = packed) THEN
      Dest[127..032] := ConvertLongInt2Double(Src[063..032])
   ELSE                ; (DataType = scalar)
      Dest[127..064] := bleibt unverändert
ELSEIF (Conversion = SingleToLongInt(XMM)) THEN
   IF (Truncate) THEN
      OP := ConvertSingle2LongIntAndTruncate
   ELSE
      OP := ConvertSingleToLongInt
   Dest[031..000] := OP(Src[031..000])
   IF (DataType = packed) THEN
      Dest[063..032] := OP(Src[031..032])
      Dest[095..064] := OP(Src[095..064])
      Dest[127..096] := OP(Src[127..096])
ELSEIF (Conversion = DoubleToLongInt(XMM)) THEN
   IF (Truncate) THEN
      OP := ConvertDouble2LongIntAndTruncate
   ELSE
      OP := ConvertDoubleToLongInt
   Dest[031..000] := OP(Src[063..000])
   Dest[063..032] := OP(Src[127..064])
   Dest[127..064] bleibt unverändert
ELSEIF (Conversion = LongInt(XMM)ToSingle) THEN
   Dest[031..000] := ConvertLongInt2Single(Src[031..000])
   Dest[063..032] := ConvertLongInt2Single(Src[063..032])
   Dest[095..064] := ConvertLongInt2Single(Src[095..064])
   Dest[127..096] := ConvertLongInt2Single(Src[127..096])
ELSEIF (Conversion = LongInt(XMM)ToDouble) THEN
   Dest[063..000] := ConvertLongIntDouble(Src[031..000])
   Dest[127..064] := ConvertLongIntDouble(Src[063..032])
```

Opcodes

Bei den Konvertierungsbefehlen können fast alle möglichen Registerkombinationen als Quelle und Ziel angegeben werden: Allzweck-, XMM- und MMX-Register bzw. Speicherstellen der entsprechenden Größe.

Opcode			Operanden	
MMX (64 Bit)	XMM 128 Bit)	Mnemonic	Dest	Src
	0F 5A /r	CVTPS2PD x, x/m	XMM	XMM / Mem64
	F3[0F 5A /r	CVTSS2SD x, x/m	XMM	XMM / Mem32
	66[0F 5A /r	CVTPD2PS x, x/m	XMM	XMM / Mem128
	F2[0F 5A /r	CVTSD2SS x, x/m	XMM	XMM / Mem64
	0F 2A /r	CVTPI2PS x, mm/m	XMM	MMX / Mem64
	F3[0F 2A /r	CVTSI2SS x, r/m	XMM	Reg / Mem32
	66[0F 2A /r	CVTPI2PD x, mm/m	XMM	MMX / Mem64
	F2[0F 2A /r	CVTSI2SD x, r/m	XMM	Reg / Mem32
	0F 2D /r	CVTPS2PI mm, x/m	MMX	XMM / Mem64
	F3[0F 2D /r	CVTSS2SI r, x/m	Reg	XMM / Mem32
	66[0F 2D /r	CVTPD2PI mm, x/m	MMX	XMM / Mem64
	F2[0F 2D /r	CVTSD2SI r, x/m	Reg	XMM / Mem64
	0F 2C /r	CVTTPS2PI mm, x/m	MMX	XMM / Mem64
	F3[0F 2C /r	CVTTSS2SI r, x/m	Reg	XMM / Mem32
	66[0F 2C /r	CVTTPD2PI mm, x/m	MMX	XMM / Mem64
	F2[0F 2C /r	CVTTSD2SI r, x/m	Reg	XMM / Mem64
	66[0F 5B /r	CVTPS2DQ x, x/m	XMM	XMM / Mem128
	F3[0F 5B /r	CVTTPS2DQ x, x/m	XMM	XMM / Mem128
	0F 5B /r	CVTDQ2PS x, x/m	XMM	XMM / Mem128
	F2[0F E6 /r	CVTPD2DQ x, x/m	XMM	XMM / Mem128
	66[0F E6 /r	CVTTPD2DQ x, x/m	XMM	XMM / Mem128
	F3[0F E6 /r	CVTDQ2PD x, x/m	XMM	XMM / Mem64

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

PF2ID, PI2FD — 3D-Now!
PF2IW, PI2FW — 3D-Now!-X

Funktion Konvertiere ShortPackedSingles in ShortPackedIntegers und umgekehrt. Benutze signed saturation, um sie als Word oder DoubleWord darzustellen (»packed floating-point to integer conversion as word/doubleword«, »packed integer to floating-point conversion as words/doublewords«).

Operation

```
IF (Instruction = PF2ID)
   IF (Src[31..00] >= 2^31) THEN
      Dest[31..00] := $7FFF_FFFF
   ELSEIF (Src[31..00] <= –2^31) THEN
      Dest[31..00] := $8000_0000
   ELSE
      Dest[31..00] := TruncateToInteger(Src[31..00])
   IF (Src[63..32] >= 2^31) THEN
      Dest[63..32] := $7FFF_FFFF
   ELSEIF (Src[63..32] <= –2^31) THEN
     Dest[63..32] := $8000_0000
   ELSE
     Dest[63..32] := TruncateToInteger(Src[63..32])
IF (Instruction = PF2IW)
   IF (Src[31..00] >= 2^15) THEN
      Dest[31..00] := $0000_7FFF
   ELSEIF (Src[31..00] <= –2^15) THEN
      Dest[31..00] := $FFFF_8000
   ELSE
      Dest[31..00] := TruncateToInteger(Src[31..00])
```

```
   IF (Src[63..32] >= 2^15) THEN
      Dest[63..32] := $0000_7FFF
   ELSEIF (Src[63..32] <= –2^15) THEN
     Dest[63..32] := $FFFF_8000
   ELSE
     Dest[63..32] := TruncateToInteger(Src[63..32])
IF (Instruction = PI2IFD)
   Dest[31..00] := ConvertToFloat(Src[31..00])
   Dest[63..32] := ConvertToFloat(Src[63..32])
IF (Instruction = PI2IFW)
   Dest[31..00] := ConvertToFloat(Src[15..00])
   Dest[63..32] := ConvertToFloat(Src[47..32])
```

Opcodes

Als Zieloperand kommt nur ein MMX-Register in Frage, die Quelle ist ebenfalls ein MMX-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 0F]1D		PF2ID mm, mm/m	MMX	MMX / Mem64
0F 0F]0D		PI2FD mm, mm/m	MMX	MMX / Mem64
0F 0F]1C		PF2IW mm, mm/m	MMX	MMX / Mem64
0F 0F]0C		PI2FW mm, mm/m	MMX	MMX / Mem64

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8	0	9	0	9
#MF	2	-	2	-	2
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6	-	6	-	6

MOVSS **SSE**
MOVSD **SSE2**

MOVAPS, MOVUPS **SSE**
MOVAPD, MOVUPD **SSE2**

MOVHPS, MOVLPS, MOVHLPS, MOVLHPS **SSE**
MOVHPD, MOVLPD **SSE2**

MOVNTPS **SSE**
MOVNTPD **SSE2**

Funktion Kopiere eine skalare Single oder Double aus einem XMM-Register oder einer Speicherstelle in ein XMM-Register oder eine Speicherstelle (»move scalar single/double«).

Kopiere eine PackedSingle oder PackedDouble aus einem XMM-Register oder einer Speicherstelle in ein XMM-Register oder eine Speicherstelle (»move aligned/unaligned packed single/double«).

Kopiere eine »halbe« PackedSingle oder PackedReal zwischen einem XMM-Register und einer Speicherstelle oder umgekehrt, dem »oberen« bzw. »unteren« Teil eines XMM-Registers in eine Speicherstelle oder umgekehrt (»move high/low packed single/double«) bzw. (»move high-to-low/low-to-high packed single«).

Schreibe eine PackedSingle oder PackedDouble aus dem XMM-Register unter Umgehung des Cache in den Speicher (»move using non-temporal hint packed single/double«).

Operation

```
IF (Instruction = MOVSS) THEN
   Dest[031..000] := Src[031..000]
   IF (Source = Memory) THEN
      Dest[127..000] := 0
   ELSE
      ; Dest[127..032] bleibt unverändert
ELSEIF (Instruction = MOVSD) THEN
   Dest[063..000] := Src[063..000]
   IF (Source = Memory) THEN
      Dest[127..064] := 0
   ELSE
      ; Dest[127..064] bleibt unverändert
ELSEIF (Instruction = MOVAPS, MOVAPD) THEN
   IF (Src or Dest unaligned) THEN
      #GP(0)
```

```
   ELSE
      Dest[127..000] := Src[127..000]
ELSEIF (Instruction = MOVUPS, MOVUPD) THEN
   Dest[127..000] := Src[127..000]
ELSEIF (Instruction = MOVLPS, MOVLPD) THEN
   IF (Destination = Memory) THEN
      Dest := Src[063..000]
   ELSE                ; (Source = Memory)
      Dest[063..000] := Src
      ; Dest[127..064] bleibt unverändert
ELSEIF (Instruction = MOVHPS, MOVHPD) THEN
   IF (Destination = Memory) THEN
      Dest := Src[127..064]
   ELSE                ; (Source = Memory)
      Dest[127..064] := Src
      ; Dest[063..000] bleibt unverändert
ELSEIF (Instruction = MOVHLPS) THEN
   Dest[063..000] := Src[127..064]
   ; Dest[127..064] bleibt unverändert
ELSEIF (Instruction = MOVLHPS) THEN
   Dest[127..064] := Src[063..000]
   ; Dest[063..000] bleibt unverändert
ELSE                   ; (Instruction = MOVNTPS, MOVNTPD)
   Dest := Src
```

Opcodes

Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode			Operanden	
MMX (64 Bit)	**XMM (128 Bit)**	**Mnemonic**	**Dest**	**Src**
	F3[0F 10 /r	MOVSS x, x/m	XMM	XMM / Mem32
	F3[0F 11 /r	MOVSS x/m x	XMM / Mem32	XMM
	F2[0F 10 /r	MOVSD x, x/m	XMM	XMM / Mem164
	F2[0F 11 /r	MOVSD x/m x	XMM / Mem64	XMM
	0F 28 /r	MOVAPS x, x/m	XMM	XMM / Mem128
	0F 29 /r	MOVAPS x/m x	XMM / Mem128	XMM
	0F 10 /r	MOVUPS x, x/m	XMM	XMM / Mem128
	0F 11 /r	MOVUPS x/m x	XMM / Mem128	XMM
	66[0F 28 /r	MOVAPD x, x/m	XMM	XMM / Mem128
	66[0F 29 /r	MOVAPD x/m x	XMM / Mem128	XMM

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	66[0F 10 /r	MOVUPD x, x/m	XMM	XMM / Mem128
	66[0F 11 /r	MOVUPD x/m x	XMM / Mem128	XMM
	0F 12 /r	MOVLPS x, m	XMM	Mem64
	0F 12 /r	MOVHLPS x, x	XMM	XMM
	0F 13 /r	MOVLPS m, x	Mem64	XMM
	0F 16 /r	MOVHPS x, m	XMM	Mem64
	0F 16 /r	MOVLHPS x, x	XMM	XMM
	0F 17 /r	MOVHPS m, x	Mem64	XMM
	66[0F 12 /r	MOVLPD x, x/m	XMM	Mem64
	66[0F 13 /r	MOVLPD x/m x	Mem64	XMM
	66[0F 16 /r	MOVHPD x, x/m	XMM	Mem64
	66[0F 17 /r	MOVHPD x/m x	Mem64	XMM
	0F 2B /r	MOVNTPS m, x	Mem128	XMM
	66[0F 2B /r	MOVNTPD m, x	Mem128	XMM

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

Bemerkungen

Die Opcodes für MOVLPS und MOVHLPS bzw. MOVHPS und MOVLHPS sind jeweils gleich, da MOVLPS und MOVHPS nur den Austausch zwischen Speicher und XMM-Register durchführen, nicht aber zwischen zwei XMM-Registern. Dies ist die Aufgabe der Befehle MOVHLPS bzw. MOVLHPS. Daher können die Befehle trotz paarweise gleichem Opcode durch das folgende ModR/M-Byte unterschieden werden.

Bitte beachten Sie auch, dass das Mnemonic MOVSD auch bei den String-Befehlen verwendet wird. Dort wird es als »Erweiterung« des Befehls MOVS mit DoubleWord-Operanden verwendet. Einen tatsächlichen Konflikt bei der Nutzung der Mnemonics gibt es jedoch nicht, da der Programmierer aufgrund des Kontextes weiß, welcher Befehl gemeint ist, und der Assembler die korrekte Befehlssequenz aufgrund der Operanden feststellen kann.

MOVMSKPS SSE
MOVMSKPD SSE2

Funktion

Erzeuge aus den Vorzeichen der Komponenten einer gepackten Struktur ein Maskenbyte (»move mask of packed single/double«).

Operation

```
Dest := 0
IF (OperandSize = Double) THEN
   N := 2
   S := 64
ELSE                    ; (OperandSize = Single)
   N := 4
   S := 32
FOR (I = 0) TO (I = N - 1) DO
  Dest[I] := Src[(I + 1) * S – 1]
```

Opcodes

Zielregister ist immer ein Allzweckregister, Quelle ein XMM-Register.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	0F 50 /r	MOVMSKPS r, x	Reg	XMM
	66[0F 50 /r	MOVMSKPD r, x	Reg	XMM

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebe-

nen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#NM	1	-	1	-	1
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1
Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

SHUFPS SSE
SHUFPD SSE2

Funktion Kopiere ausgewählte Daten aus dem Quell- in den Zieloperanden (»shuffle packed single/double«).

Operation

```
IF (OperandSize = Double) THEN
   N := 2
   S := 64
   X := 1
ELSE               ; (OperandSize = Single)
   N := 4
   S := 32
   X := 2
FOR (I = 0) TO (I = N - 1) DO
   ; Bitbereich des Zieloperanden
   U := (I + 1) * S – 1    ; höchstes Bit
   L := (I * S)            ; niedrigstes Bit
   ; Bitbereich des Index
   UI := (I + 1) * X – 1
   LI := I * X
   Index := Src2[UI..LI]
   ; Bitbereich des Quelloperanden
   US := (Index + 1) * S – 1
   LS := (Index * S)
   Dest[U..L] := Src1[US..LS]
```

Opcodes Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe sowie eine 8-Bit-Konstante.

Opcode		Mnemonic	Operanden		
MMX (64 Bit)	XMM (128 Bit)		Dest	Src	Src2
	0F C6 /r	SHUFPS x, x/m, c	XMM	XMM / Mem128	Const8
	66[0F C6 /r	SHUFPD x, x/m, c	XMM	XMM / Mem128	Const8

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

UNPCKHPS, UNPCKLPS — SSE
UNPCKHPD, UNPCKLPD — SSE2

Funktion

Mische abwechslungsweise die SingleReals oder DoubleReals aus den unteren oder oberen Hälften in einer gepackten Struktur (»unpack high/low packed singles/doubles«).

Operation

```
IF (OperandSize = Single) THEN
   N := 4
   S := 32
ELSE             ;  (OperandSize = Double)
   N := 2
   S := 64
IF (Source = lower half of structure) THEN
   B := 0
```

```
ELSE                ; (source = upper half of structure)
   B := (N * S) DIV 2
FOR (I = 0) TO (I = (N DIV 2) – 1) DO
   U := B + (I + 1) * S – 1      ; high bit source operands
   L := B + I * S                ; low bit source operands
   U1 := (I * S) + U             ; high bit dest lower half
   L1 := (I * S) + L             ; low bit dest lower half
   U2 := ((I + 1) * S) + U       ; high bit dest upper half
   L2 := ((I + 1) * S) + L       ; low bit dest upper half
   Dest[U1..L1] := Dest[U..L]
   Dest[U2..L2] := Src[U..L]
```

Opcodes Als Zieloperand kommt nur ein XMM-Register in Frage, die Quelle ist ebenfalls ein XMM-Register oder eine Speicherstelle der entsprechenden Größe.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
	0F 14 /r	UNPCKLPS x, x/m	XMM	XMM / Mem128
	0F 15 /r	UNPCKHPS x, x/m	XMM	XMM / Mem128
	66[0F 14 /r	UNPCKLPD x, x/m	XMM	XMM / Mem128
	66[0F 15 /r	UNPCKHPD x, x/m	XMM	XMM / Mwm128

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden CPU-Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	./.
#GP	8, 68	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	./.
#SS	1	0	-	0	-
#UD	6, 13, 14, 16	-	6, 13, 14, 16	-	6, 13, 14, 16
#XM	1	-	1	-	1

Exception-Flags	#D	#IA	#IS	#O	#P	#U	#Z
	-	-	-	-	-	-	-

4.3 SIMD-Befehle für Verwaltungs- und Kontrollaufgaben

EMMS **MMX**
FEMMS **3D-Now!**

Funktion

Mache die MMX-Register für die FPU nutzbar, indem die Tag-Werte aller Register auf »empty« gesetzt werden (»empty MMX state«, »fast empty MMX state«).

Operation

```
[TagRegister] := $FFFF
```

Opcodes

Beide Befehle besitzen keine Operanden.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM (128 Bit)		Dest	Src
0F 77		EMMS		
0F 0E		FEMMS		

Der grau unterlegte Befehl ist nicht kompatibel, da er nur auf AMD-Prozessoren verfügbar ist.

Exceptions

Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#MF	1	-	1	-	1
#NM	1	-	1	-	1
#UD	6	-	6	-	6

LDMXCSR **SSE**
STMXCSR **SSE**

Funktion

Lade oder sichere das MXCS-Register (»load/store MXCSR«).

Operation

```
IF (Instruction = LDMXCSR) THEN
   [MXCSR] := Src
ELSE                    ; (Instruction = STMXCST)
   Dest := [MXCSR]
```

Opcodes Beide Befehle erwarten einen Operanden, der den Inhalt des MXCS-Registers darstellt.

Opcode		Mnemonic	Operanden	
MMX (64 Bit)	XMM 128 Bit)		Dest	Src
	0F AE /2	LDMXCSR m		Mem32
	0F AE /3	STMXCSR m	Mem32	

Exceptions Tabelle 5.1 bis Tabelle 5.13 ab Seite 287 listen die Gründe für die entsprechenden Exceptions anhand der in der Spalte »Grund« angegebenen Codezahlen auf. Wenn dem Exception-Handler ein ErrorCode übergeben wird, wird er in der Spalte »ErrorCode« angegeben.

Typ	Protected Mode		Virtual 8086 Mode		Real Mode
	Grund	ErrorCode	Grund	ErrorCode	Grund
#AC	2	0	2	0	-
#GP	8	0	9	0	9
#NM	1	-	1	-	1
#PF	1	fault code	1	fault code	-
#SS	1	0	-	0	-
#UD	6, 13, 14	-	6, 13, 14	-	6, 13, 14

5 Exceptiongründe

5.1 CPU-Exceptions

#**AC**	
Code	**Bedeutung**
1	Der Alignment-Check ist aktiviert (CR0[AM] = 1, EFLAGS[AC] = 1), der CPL ist 3, und es wird ein nicht ausgerichtetes Datum vorgefunden.
MMX / XMM:	
2	Der Alignment-Check ist aktiviert (CR0[AM] = 1, EFLAGS[AC] = 1), der CPL ist 3, und es wird ein nicht ausgerichtetes Datum vorgefunden (nur MMX!).
3	Der Alignment-Check ist aktiviert (CR0[AM] = 1, EFLAGS[AC] = 1), der CPL ist 3, und es wird ein nicht an 16-Byte-(»Paragraphen«)-Grenzen ausgerichtetes Datum vorgefunden. Allerdings kann bei CPL < 3 bzw. implementationsabhängig oder wenn der Alignment-Check deaktiviert wurde eine #GP mit dem Grund 69 anstelle dieser exception ausgelöst werden.

Tabelle 5.1: Gründe für eine Exception des Typs Alignment Check #AC

#**BR**	
Code	**Bedeutung**
1	Das Testergebnis liegt außerhalb der im Operanden vorgegebenen Grenzen.

Tabelle 5.2: Gründe für eine Exception des Typs Bound Range exceeded #BR

#**DB**	
Code	**Bedeutung**
1	Das GD-Flag in Debugregister DR7 ist gesetzt, und es wird auf ein Debugregister zugegriffen.

Tabelle 5.3: Gründe für eine Exception des Typs Debug #DB

#**DE**	
Code	**Bedeutung**
1	Der Operand hat den Wert $00.
2	Der Divisor (Quelloperand) hat den Wert $00.
3	Das Ergebnis ist im Format des Zieloperanden nicht darstellbar.

Tabelle 5.4: Gründe für eine Exception des Typs Divide #DE

#**GP**	
Code	**Bedeutung**
Allgemein:	
1	Der Protected-Mode ist nicht aktiv.
2	Der Befehl kann im Virtual-8086-Mode nicht ausgeführt werden.
Nullselektoren	
3	Auf das DS-, ES-, FS- oder GS-Register wird zugegriffen, der enthaltene Selektor ist aber ein Nullselektor.
4	Der im Zieloperanden, Interrupt-Gate, Trap-Gate, Call-Gate, Task-Gate oder Task-State-Segment verzeichnete Selektor ist ein Nullselektor.
5	Der im Gate verzeichnete Selektor für das Codesegment ist ein Nullselektor.
6	Der in das SS-Register zu ladende Selektor ist ein Nullselektor.
7	Die Ziel-/Rücksprungadresse ist Null, oder der Selektor für das Stacksegment ist ein Nullselektor.
Grenzüberschreitungen	
8	Die verwendete Speicheradresse liegt außerhalb des durch den Selektor in CS, DS, ES, FS oder GS ausgewählten Segments.
9	Die verwendete Speicheradresse liegt außerhalb des adressierbaren Bereichs von $0000 bis $FFFF.
10	Die Rücksprungadresse bzw. die Zieladresse im Operanden, Call-Gate oder Task-Switch-State liegt nicht im anzuspringenden Codesegment.
11	Die Zieladresse im Codesegment liegt außerhalb des adressierbaren Bereichs von $0000 bis $FFFF für das Codesegment, weil ein ADRSIZE-Präfix verwendet wurde.
12	Der Selektor für das Codesegment, das Gate oder den Task-Switch-State zeigt nicht in eine Deskriptortabelle.
13	Der Selektor für das Codesegment oder Stacksegment für den Rücksprung zeigt nicht in eine Deskriptortabelle.
14	Der Selektor zeigt auf einen Eintrag außerhalb der verwendeten Tabelle.
15	Der Selektor zeigt auf eine LDT oder eine Stelle außerhalb der GDT.
16	Eine Interrupt-Nummer zeigt auf einen Vektor außerhalb der IDT.

Tabelle 5.5: Gründe für eine Exception des Typs General Protection #GP

#GP	
Code	**Bedeutung**
17	Die Einsprungadresse in der IDT, im Interrupt-Gate, Trap-Gate oder Task-Gate liegt nicht in einem Codesegment.
18	Das EBP-Register zeigt auf eine Adresse, die außerhalb des Stack-Segments liegt.
19	Das EBP-Register zeigt auf eine Adresse, die außerhalb des adressierbaren Bereichs von $0000 bis $FFFF liegt.
Falsche Segmente	
20	Das Ziel der Operation liegt in einem als non-writable markierten Segment.
21	Der Selektor, der in das SS-Register geladen werden soll, zeigt auf ein als non-writable markiertes Segment.
22	Der Selektor, der in das DS-, ES-, FS- oder GS-Register geladen werden soll, zeigt nicht auf ein als readable markiertes Code- oder Datensegment.
23	Der Selektor im Operanden zeigt nicht auf ein conforming oder non-conforming Codesegment, auf ein Call-Gate, Task-Gate oder ein Task-Switch segment.
24	Der Selektor eines Call-Gates, Interrupt-Gates oder Trap-Gates zeigt nicht auf ein Codesegment.
25	Der Selektor im Operanden zeigt nicht auf einen GDT-Eintrag, oder der selektierte GDT-Eintrag zeigt nicht auf eine LDT.
26	Der Selektor in einem Task-Gate zeigt auf ein Task-Switch segment, das als not present markiert ist.
27	Das Stacksegment ist kein als writable markiertes Datensegment.
28	Der Deskriptor eines Codesegments enthält nicht die Typinformation für ein Codesegment.
29	Der Deskriptor, auf den der Selektor der Rücksprungadresse zeigt, beschreibt kein Codesegment.
30	Der Deskriptor in der IDT beschreibt keine Interrupt-Gate, Trap-Gate oder Task-Gate.
Privilegverletzungen	
31	Der IOPL ist kleiner als 3, oder der DPL des Interrupt-, Trap- oder Task-Gate-Deskriptors ist nicht gleich 3.
32	Der CPL ist größer als 0.
33	Der CPL ist größer als der IOPL (I/O-Privileg-Level) des aktuellen Programms oder der aktuellen Routine
34	Der CPL ist größer als der IOPL (I/O-Privileg-Level), und irgendeines der Permission-Bits im TSS für den angesprochenen Port ist gesetzt.
35	Der DPL im anzuspringenden non-conforming Codesegment ist nicht gleich dem CPL, oder der RPL des Selektors in der Ziel- oder Rücksprungadresse ist größer als der CPL.

Tabelle 5.5: Gründe für eine Exception des Typs General Protection #GP (Forts.)

#GP	
Code	**Bedeutung**
36	Der DPL im anzuspringenden conforming Codesegment ist größer als der CPL.
37	Der DPL im anzuspringenden conforming Codesegment ist größer als der RPL im Selektor der Rücksprungadresse.
38	Der DPL des Codesegments, auf das ein Call-Gate zeigt, ist größer als der CPL.
39	Der DPL des Stacksegments ist nicht gleich dem RPL im Selektor der Rücksprungadresse.
40	Der DPL eines Call-Gates, Task-Gates oder eines Task-Switch-State ist kleiner als der CPL oder als der RPL des betreffenden Selektors.
41	Der DPL ist kleiner als der CPL und der RPL des Selektors, der in das DS-, ES-FS- oder GS-Register geladen wird. Dies ist eine Schutzverletzung, obwohl der Selektor auf ein Daten- oder non-conforming Codesegment zeigt.
42	Der DPL des Interrupt-Gates, Trap-Gates oder Task-Gates, das durch Auslösen des INT n-Befehls (auch INT3 und INTO) verwendet werden soll, ist kleiner als der CPL.
43	Der RPL des Selektors in der Ziel- oder Rücksprungadresse ist größer als der CPL.
44	Der RPL des Selektors der Rücksprungadresse ist kleiner als das CPL.
45	Der RPL im Selektor des Stacksegments und der Rücksprungadresse sind verschieden.
46	Das SS-Register wird mit einem Selektor geladen, dessen RPL nicht gleich dem DPL des Segments und dem CPL ist.
Falsche Bitstellungen	
47	Das Global/Local-Flag im Selektor eines Task-Switch-States steht auf »local«.
48	Ein Permission-Bit im Task-Switch-State für den angesprochenen Port ist gesetzt.
49	Versuch, eine ungültige Bitkombination in CR0 zu schreiben (z.B. PG = 1 und PE = 0 oder CD = 0 und NE = 1).
50	Versuch, ein Bit im Kontrollregister CR4 zu setzen.
51	Versuch, in der Pointer-Tabelle eines Page-Directory-Eintrages reservierte Bits für die Nutzung der Physical-Address-Extension zu setzen, wenn PAE in Kontrollregister CR4 und PG in Kontrollregister CR0 gesetzt sind.
52	Der CPL ist größer als 0, und das PCE-Flag in Kontrollregister CR4 ist gelöscht.
53	Das PCE-Flag in Kontrollregister CR4 ist gelöscht.
54	Der CPL ist größer als 0, und das TSD-Flag in Kontrollregister CR4 ist gesetzt.
55	Das TSD-Flag in Kontrollregister CR4 ist gesetzt.

Tabelle 5.5: Gründe für eine Exception des Typs General Protection #GP (Forts.)

#GP	
Code	**Bedeutung**
Ungültige Operanden/Befehle	
56	Auf die Debugregister kann in dieser Betriebsart nicht zugegriffen werden.
57	POPF/POPFD wurde mit einem OPSIZE-Präfix versehen.
58	Das SP- bzw. ESP-Register enthält den Wert 7, 9, 11, 13 oder 15.
59	Der Inhalt von ECX kodiert ein reserviertes oder nicht implementiertes modellspezifisches Register (MSR).
60	Der Wert in ECX ist nicht 0 oder 1.
61	Das modellspezifische Register (MSR) SYSENTER_CS_MSR enthält den Wert 0.
Inkonsistenzen	
62	Beim Laden des SS-Registers wird eine Fehlerursache vorgefunden: Der Selektor zeigt nicht in die GDT, der RPL des Selektors ist nicht gleich dem CPL, das als Stack verwendete Datensegment ist als non-writable markiert, oder der DPL des Segments ist nicht gleich dem CPL.
63	Beim Laden des DS-, ES-, FS- oder GS-Registers wird folgende Fehlerursache vorgefunden: Der Selektor ist ein Nullselektor oder zeigt nicht in die GDT, das Segment ist kein Daten- oder readable Codesegment, oder das Segment ist zwar ein Daten- oder non-conforming Codesegment, aber sowohl der RPL als auch der CPL sind größer als der DPL.
64	Der Deskriptor weist seinen Task-Switch-State als busy oder nicht verfügbar aus.
65	Der Selektor im Quelloperand zeigt auf ein Segment, das kein Task-Switch-State (TSS) ist, oder aber auf ein TSS, das zu einem gerade aktiven Task gehört.
66	Das Codesegment der Rücksprungadresse ist non-conforming, und sein DPL ist nicht gleich dem RPL des Selektors für das Codesegment.
67	Das Codesegment der Rücksprungadresse ist conforming, und sein DPL ist größer als der RPL des Selektors für das Codesegment.
MMX / XMM	
68	Einer der Operanden ist nicht an einer Paragraphengrenze (16-Bit-Grenze) ausgerichtet (nur XMM, nicht MMX!)
69	Der Speicheroperand ist unabhängig vom Segment nicht an einer 16-Byte-Grenze ausgerichtet.

Tabelle 5.5: Gründe für eine Exception des Typs General Protection #GP (Forts.)

#**MF**

Code	Bedeutung
1	Eine unmaskierte FPU-Exception wartet auf die Behandlung. Welche das ist, kann aus dem Statuscord der FPU ermittelt werden. Mögliche FPU-Exceptions sind #D. #IA, #IS, #P, #O, #U und #Z.
MMX / XMM:	
2	Eine unmaskierte FPU-Exception wartet auf die Behandlung (nur MMX!).

Tabelle 5.6: Gründe für eine Exception des Typs Math Fault #MF

#NM

Code	Bedeutung
1	Das TS-Flag in Kontrollregister CR0 ist gesetzt.
2	Das EM-Flag oder das TS-Flag in Kontrollregister CR0 ist gesetzt.
4	Das MP-Flag und das TS-Flag in Kontrollregister CR0 sind gesetzt.

Tabelle 5.7: Gründe für eine Exception des Typs Device Not Available #NM

#NP

Code	Bedeutung
1	Ein Codesegment, Datensegment, Stacksegment oder ein Call-Gate, Task-Gate oder Task-State-Segment (TSS) sind als not present markiert.
2	Der LDT-Deskriptor in der GDT ist als not present markiert.
3	Das Task-State-Segment ist als not present markiert.
4	Der nach DS, ES, FS oder GS zu ladende Selektor zeigt auf ein Segment, das als not present markiert ist.

Tabelle 5.8: Gründe für eine Exception des Typs Segment Not Present #NP

#PF

Code	Bedeutung
1	Beim Versuch, auf eine Page zuzugreifen, wurde ein Fehler entdeckt.

Tabelle 5.9: Gründe für eine Exception des Typs Page Fault #PF

#SS	
Code	**Bedeutung**
1	Die verwendete Speicheradresse liegt außerhalb des durch den Selektor in SS ausgewählten Segments.
2	Der neu geladene Wert im SS- oder ESP-Register weist auf Adressen, die außerhalb der Stacksegmentgrenzen liegen.
3	Das Ablegen von Daten auf dem Stack sprengt die Grenzen des Stacksegments.
4	Das Ablegen einer Rücksprungadresse und/oder Flags und/oder eines Fehlercodes sprengt die Grenzen des Stacksegments.
5	Das Ablegen einer Rücksprungadresse und/oder Parameter und/oder Stackpointer auf den Stack, ohne dass ein Task-Switch stattfindet, führt zum Überschreiten der Grenzen des Stacksegments.
6	Das Ablegen einer Rücksprungadresse und/oder Parameter und/oder Stackpointer auf den Stack bei einem Task-Switch führt zum Überschreiten der Grenzen des Stacksegments.
7	Das SS-Register wird mit einem Selektor geladen, dessen dazugehöriges Segment als not present markiert ist.
8	Das SS-Register wird im Rahmen eines Task-Switchs mit einem Selektor geladen, dessen dazugehöriges Segment als not present markiert ist.
9	Die Stackspitze liegt außerhalb der Segmentgrenzen des Stacksegments.
10	Stackspitze und Stackbasis liegen außerhalb der Segmentgrenzen des Stacksegments.

Tabelle 5.10: Gründe für eine Exception des Typs Stack Fault #SS

#TS	
Code	**Bedeutung**
1	Der Selektor für das Stacksegment ist ein Nullselektor.
2	SS:ESP liegt außerhalb des TSS.
3	Der RPL des im TSS verzeichneten Selektors für das Stacksegment ist nicht gleich dem DPL des Codesegments, auf das umgeschaltet wird.
4	Der DPL des neuen Stacksegments, auf das der Selektor im TSS zeigt, ist nicht gleich dem DPL des neuen Codesegments.
5	Das neue Stacksegment ist kein als writable markiertes Datensegment.
6	Der Selektor für ein Stacksegment zeigt nicht in eine Deskriptortabelle.

Tabelle 5.11: Gründe für eine Exception des Typs Invalid TSS #TS

#UD	
Code	**Bedeutung**
1	Der Befehl ist in diesem Modus nicht verfügbar.
2	Der Quelloperand ist keine Speicherstelle.
3	Der Zieloperand ist kein Register.
4	Der Zieloperand ist ein Register.
5	Der Zieloperand ist keine Speicherstelle
6	Das EM-Flag in Kontrollregister CR0 ist gesetzt.
7	Das Präfix kann nicht in Verbindung mit dem sich anschließenden Befehl eingesetzt werden.
8	Das CS-Register soll geladen werden.
9	Das DE-Flag in Kontrollregister CR4 ist gesetzt, und ein Operand bezeichnet die Debugregister DR4 oder DR5.
10	Der Prozessor ist bei Ausführung des Befehls nicht im System-Management-Mode (SMM).
11	Die Exception wird mit Sicherheit ausgeführt.
12	Das System Call Extension Bit (SCE) im modellspezifischen Register (MSR) $C0000080 (Extended Feature Flags Register) ist nicht gesetzt
MMX / XMM:	
13	Das Flag OSFXCSR in Kontrollregister CR 4 ist gelöscht (nur XMM!).
14	Das Flag SSE2 in den feature flags des CPUID-Befehls ist gelöscht (nur XMM!).
15	Im ModR/M-Byte ist der Wert des Feldes mod nicht 11_b!
16	Das Flag OSXMMEXCEPT ist gelöscht und eine unmaskierte XMM-Exception wartet auf die Behandlung (nur XMM!).
17	Das EM flag in Kontrollregister CR0 ist gesetzt.
18	Das Flag FXSR in den feature flags des CPUID-Befehls ist gelöscht.
19	Dem Befehl geht das Präfix LOCK voran.
20	Das Flag CLFLSH in den feature flags des CPUID-Befehls ist gelöscht.

Tabelle 5.12: Gründe für eine Exception des Typs Invalid Opcode #UD

#XM	
Code	**Bedeutung**
1	Das Flag OSXMMEXCEPT in Kontrollregister CR4 ist gesetzt und eine unmaskierte XMM-Exception wartet auf die Behandlung (nur XMM!).

Tabelle 5.13: Gründe für eine Exception des Typs Math Fault #MF

5.2 FPU- und XMM-Exceptions

#D	
Code	**Bedeutung**
1	Der Quelloperand ist denormalisiert.
2	Der Zieloperand ist denormalisiert.
3	Ein oder beide Operanden sind denormalisiert.

Tabelle 5.14: Gründe für eine Exception des Typs Denormalized Operand #D

#IS	
Code	**Bedeutung**
1	Es erfolgte ein Stack-Unterlauf.
2	Es erfolgte ein Stack-Überlauf.

Tabelle 5.15: Gründe für eine Exception des Typs FPU Invalid Operation #I (speziell Stack Fault #IS)

#IA	
Code	**Bedeutung**
1	Der Quelloperand ist eine sNaN oder ein nicht unterstütztes Format.
2	Der Quelloperand ist leer, enthält eine NaN, ±∞, ein nicht unterstütztes Format oder einen Wert, der nicht mit 18 BCD-Ziffern dargestellt werden kann.
3	Der Quelloperand ist negativ (Ausnahme: -0).
4	Der Quelloperand ist zu groß für das Zielformat.
5	Der Operand ist ∞.
6	Die Operanden sind Unendlichkeiten mit gleichem Vorzeichen.
7	Die Operanden sind Unendlichkeiten mit ungleichem Vorzeichen.
8	Der eine Operand ist ±0, der andere ±∞.
9	Einer der oder beide Operanden sind NaNs oder weisen ein nicht unterstütztes Format auf.
10	Einer der oder beide Operanden sind sNaNs (keine qNaNs!) oder weisen ein nicht unterstütztes Format auf.
11	Der Quelloperand ist eine sNaN, der Divisor ist 0, der Dividend ∞ oder ein nicht unterstütztes Format.
12	Das Register ist als empty markiert.

Tabelle 5.16: Gründe für eine Exception des Typs FPU Invalid Operation #I (speziell Arithmetic Fault #IA)

#IA

Code	Bedeutung
13	Division von ±∞ durch ±∞ oder ±0 durch ±0.
14	Einer der oder beide Operanden sind qNaNs mit einem Vergleichsprädikat, das nicht auf Gleichheit, »unordered«, Nicht-Gleichheit oder »ordered« prüft, oder einer der oder beide Operanden sind sNaNs.

Tabelle 5.16: Gründe für eine Exception des Typs FPU Invalid Operation #I (speziell Arithmetic Fault #IA) (Forts.)

#O

Code	Bedeutung
1	Das Ergebnis ist zu groß für das Zielformat.

Tabelle 5.17: Gründe für eine Exception des Typs FPU Overflow #O

#P

Code	Bedeutung
1	Das Ergebnis kann im Zielformat nicht korrekt dargestellt werden.

Tabelle 5.18: Gründe für eine Exception des Typs FPU Precision #P

#U

Code	Bedeutung
1	Das Ergebnis ist zu klein für das Zielformat.

Tabelle 5.19: Gründe für eine Exception des Typs FPU Underflow #U

#Z

Code	Bedeutung
1	Division des Zieloperanden durch ±0, wenn der Zieloperand nicht den Wert ±0 hat.
2	Division des Quelloperanden durch ±0, wenn der Quelloperand nicht den Wert ±0 hat.
3	Der TOS (=ST(0)) enthält den Wert ±0.

Tabelle 5.20: Gründe für eine Exception des Typs FPU Zero Divide #Z

6 Befehls-Dekodierung

In diesem Kapitel sind die Tabellen angegeben, die Sie benötigen, wenn Sie einen Befehl anhand seiner Befehlssequenz dekodieren möchten. Dies erfolgt in vier Schritten:

- Dekodierung der Präfixe
- Dekodierung des Opcodes
- Dekodierung des eventuell vorhandenen ModR/M- und ggf. SIB-Bytes
- Dekodierung einer eventuell vorhandenen Adresse oder Konstante.

6.1 Dekodierung des/der Präfixe(s)

Die Präfixe sind Ein-Byte-Codes, die Sie in der Tabelle für Ein-Byte-Opcodes (Tabelle 6.1) finden.

6.2 Dekodierung des Opcodes

Generell gibt es Ein-, Zwei- und Drei-Byte-Opcodes. Ein-Byte-Opcodes sind in Tabelle 6.1 für CPU- und SIMD-Befehle und in Tabelle 6.7 für FPU-Befehle dargestellt. Die Zwei-Byte-Opcodes sind im Falle von CPU- und FPU-Befehlen in Tabelle 6.2, im Falle von FPU-Befehlen in Tabelle 6.8 dargestellt. Einige SIMD-Befehle verwenden Präfixe zur weiteren Kodierung. Solche »Drei-Byte-Opcodes« finden Sie in Tabelle 6.3 bis Tabelle 6.5.

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 kodiert. In Tabelle 6.1 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

kodiert 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 Kodierung herangezogen werden. In diesen Fällen spricht man vom *Spec*-Feld (»special«) des ModR/M-Bytes. Solche Befehle sind in Tabelle 6.1 ebenfalls grau unterlegt und mit »group x« bezeichnet. Sie werden in Tabelle 6.6 dargestellt.

6.2.1 Dekodierung eines ModR/M- und ggf. eines SIB-Byte

Zur Dekodierung eines eventuellen ModR/M und ggf. vorhandenen SIB-Bytes gibt es ein Flussdiagramm, das die Dekodierung vereinfacht, sobald ein Speicheroperand im ModR/M-Byte kodiert ist. Dieses Flussdiagramm finden Sie in Abbildung 6.1. Falls als Operand keine Speicherstelle involviert ist, ist die Dekodierung des ModR/M-Bytes einfach: mod hat dann den Wert 11b und reg und R/M kodieren jeweils ein Register.

6.2.2 Dekodierung 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 Dekodierung 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 kodiert 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 Kodierung 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 6.1: 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	syscall	clts	sysret	invd	wbinvd	reserved	ud2	reserved	prefetch *)	femms *)	*)
1_	movups[3]	movups[4]	**)	movlps[4]	unpcklps	unpckhps	***)	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[2,4]	shld[2,4,5]	shld[2,4,6]	reserved	reserved	push gs	pop gs	rsm	bts[2,4]	shrd[2,4,5]	shrd[2,4,6]	group 15	imul[2]
B_	cmpxchg[1]	cmpxchg[2]	lss[3]	btr[2,4]	lfs[3]	lgs[3]	movzx[3,C]	movzx[3,D]	reserved	group 10	group 8	btc[2,4]	bsf[2,3]	bsr[2,3]	movsx[3,C]	movsx[3,D]
C_	xadd[1]	xadd[2]	cmpps[3,5]	movnti[H]	pinsrw[E,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]	paddq	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]	pmuludq	pmaddwd[3]	psadbw[3]	maskmovq[J]	psubb[3]	psubw[3]	psubd[3]	psubq	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 Kodierung 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.

**) Ist als Quelloperand eine Speicherstelle beteiligt, heißt der Befehl movlps[3], ansonsten movhlps.

***) Ist als Quelloperand eine Speicherstelle beteiligt, heißt der Befehl movhps[3], ansonsten movlhps.

Tabelle 6.2: 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]														
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 Kodierung 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 6.3: »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]	ccttpd2pi[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]	punpckhbw[1]	punpckhwd[1]	punpckhdq[1]	packsswd[1]			movd[A]	
7_	pshufw[1,9]	group 12	group 13	group 14	pcmpeqb[1]	pcmpeqw[1]	pcmpeqd[1]								movd[B]	
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]	paddq[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	pmuludq[1]	pmaddwd[1]	psadbw[1]	maskovdqu[7]	psubb[1]	psubw[1]	psubd[1]	psubq[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 Kodierung 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, A: Ziel ist XMM-Register, Quelle 32-Bit-R/M, B: Ziel ist 32-Bit-R/M, Quelle XMM-Register.

Tabelle 6.4: »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 Kodierung 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 6.5: »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; F7: xxx r/m bzw. text r/m8, c8; test 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 reg, c8
13	reserved	reserved	psrld	reserved	psrad	reserved	pslld	reserved	ja	nein	0F 72: xxx reg, c8
14	reserved	reserved	psrlq	psrldq[1]	reserved	reserved	psllq	pslldq[1]	ja	nein	0F 73: xxx reg, 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.

[1]: mit Präfix $66; *) 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 6.6: 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 6.7: 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 6.8: 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 kodiert 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 6.9: »Drei-Byte«-Opcodes für die 3DNow!-Befehle der AMD-Prozessoren

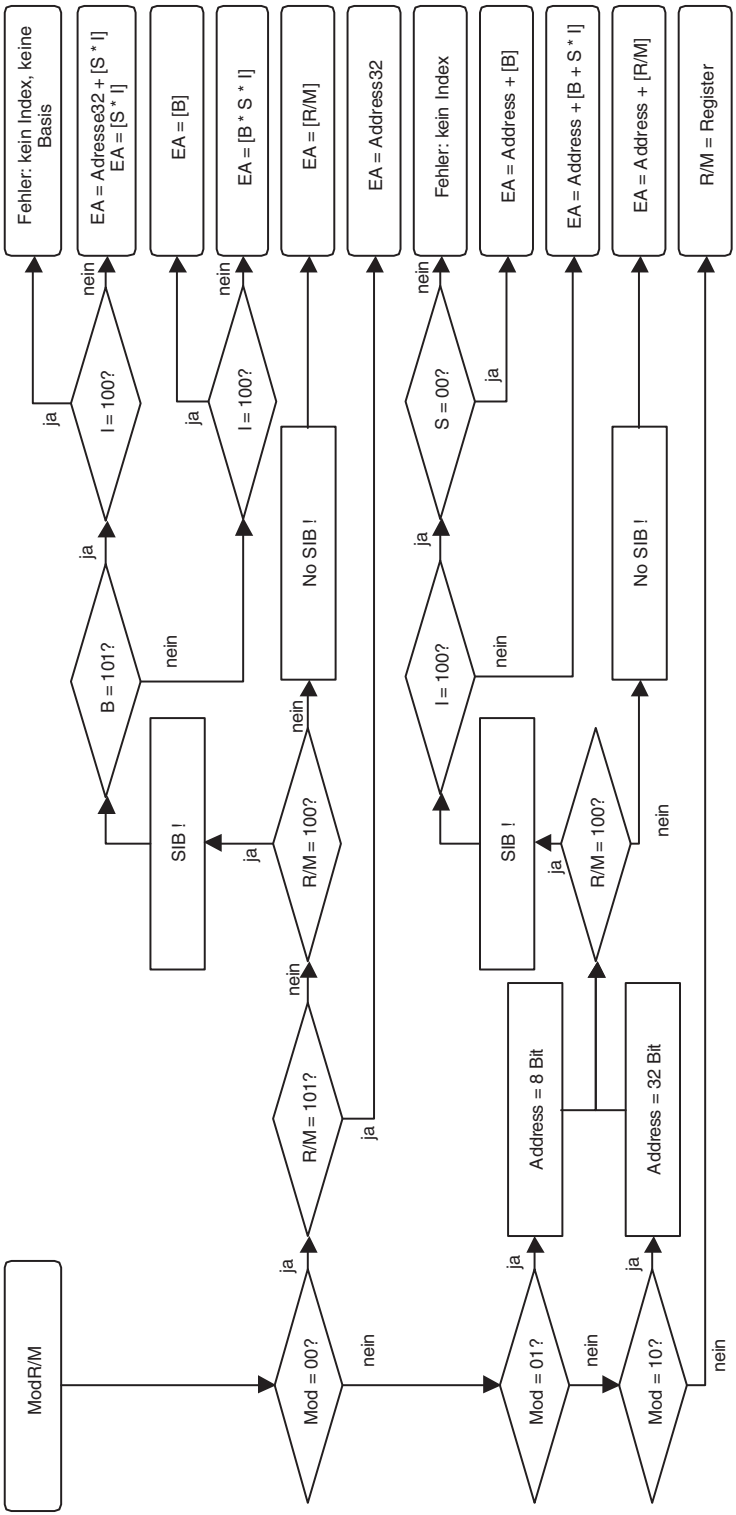


Abbildung 6.1: Flussdiagramm zur Dekodierung des ModR/M- und SIB-Bytes einer Befehlssequenz

Stichwortverzeichnis