8 Die Floating-Point-Einheit

Die Floating-Point-Einheit (FPU), des 80486 ist mit ihren Vorgängern 8087, 80287 und 80387 **kompatibel** und ist das OnChip-**Äquivalent** des **80387**.

Dem Programmierer erscheint die FPU wie ein zusätzlicher Satz von Registern, Datentypen und Befehlen. Die CPU identifiziert die Befehle der FPU durch das Bitmuster 11011B, mit dem jeder Numerik-Befehl markiert ist. Alle so gekennzeichneten Befehle werden als ESCAPE-Befehle oder kurz als **ESC**-Befehle bezeichnet.

Das folgende Kapitel enthält eine komplette Einführung in die Architektur, das Zahlensystem und die Numerik-Mechanismen der FPU.

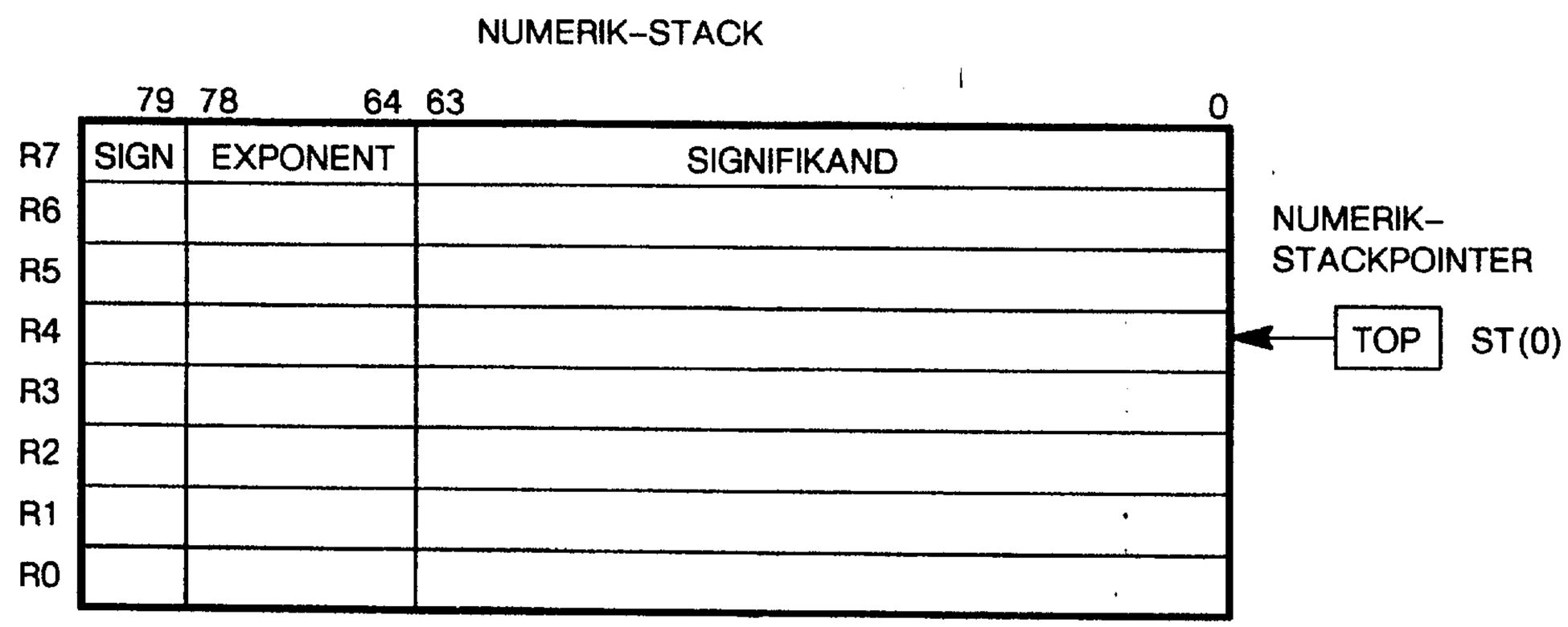
8.1 Die Numerik-Register

Die Numerik-Register des 80486 bestehen aus

- acht individuellen 80-Bit-Registern, die als Register-Stack organisiert sind,
- drei 16-Bit-Registern, die das FPU-Statuswort, das FPU-Kontrollwort und das FPU-Tagwort enthalten und aus
- Befehls- und Operanden-Pointern.

8.1.1 FPU-Register-Stack

Der FPU-Register-Stack ist im folgenden Bild zu sehen:

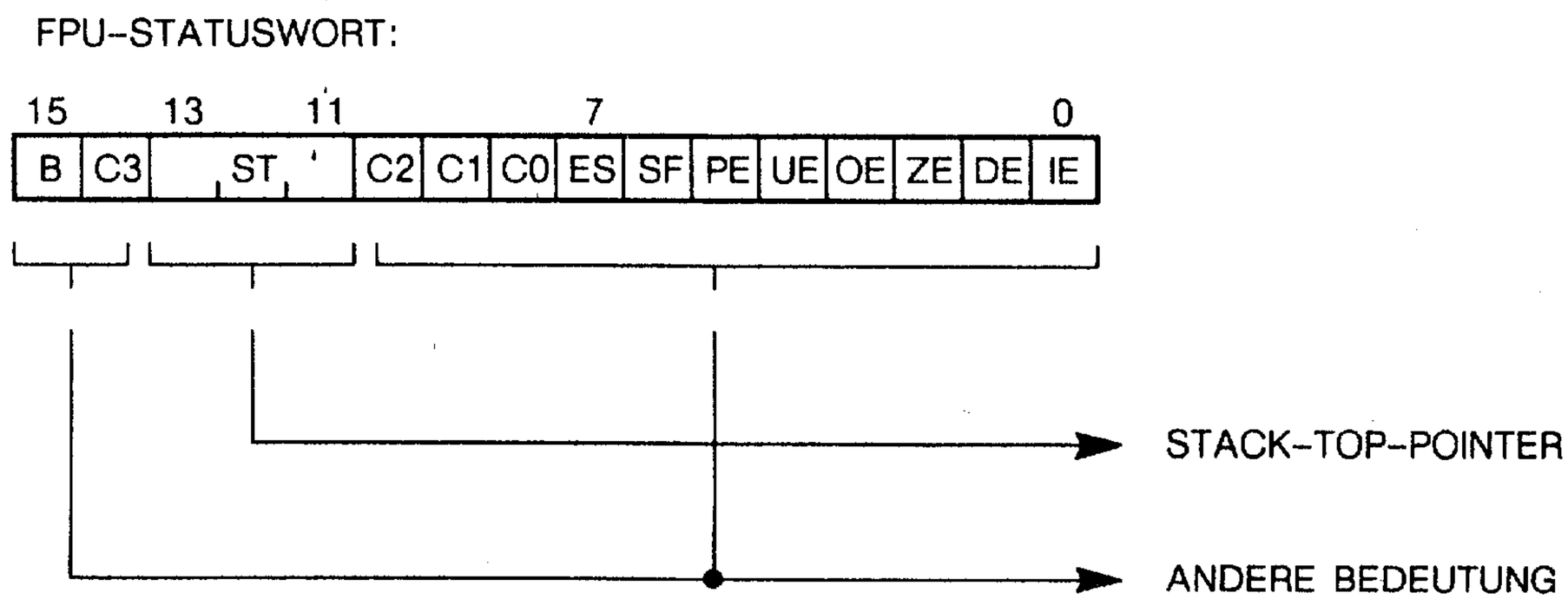


Jedes der acht Numerik-Stack-Register ist in drei Felder unterteilt und zwar in

- das Vorzeichen (Sign)-Feld (1 Bit),
- das Exponenten-Feld (15 Bits) und
- das Signifikanden-Feld (64 Bits).

Die Adressierung der individuellen Numerik-Stack-Register übernimmt der sogenannte Numerik-Stackpointer, der immer auf die augenblickliche Spitze (Top) des Numerik-Stack zeigt. Dabei kann **jedes** Numerik-Register Stack-Top-Register werden. ASM386/486 unterstützt diese Adressierungsart durch den Ausdruck ST(0) oder kurz ST. Eine **Schreib**operation (PUSH) dekrementiert zunächst den Numerik-Stackpointer und speichert danach den Floating-Point-Operanden im neuen Stack-Top-Register.

Eine **Lese**operation (POP) liest zunächst den Floating-Point-Operanden vom augenblicklichen Stack-Top-Register und inkrementiert anschließend den Numerik-Stackpointer. Verantwortlich für die Adressierung des Numerik-Stack ist das ST-Feld (Stack-Top) im FPU-Statuswort. Die Position dieses Feldes ist im folgenden Layout zu sehen:



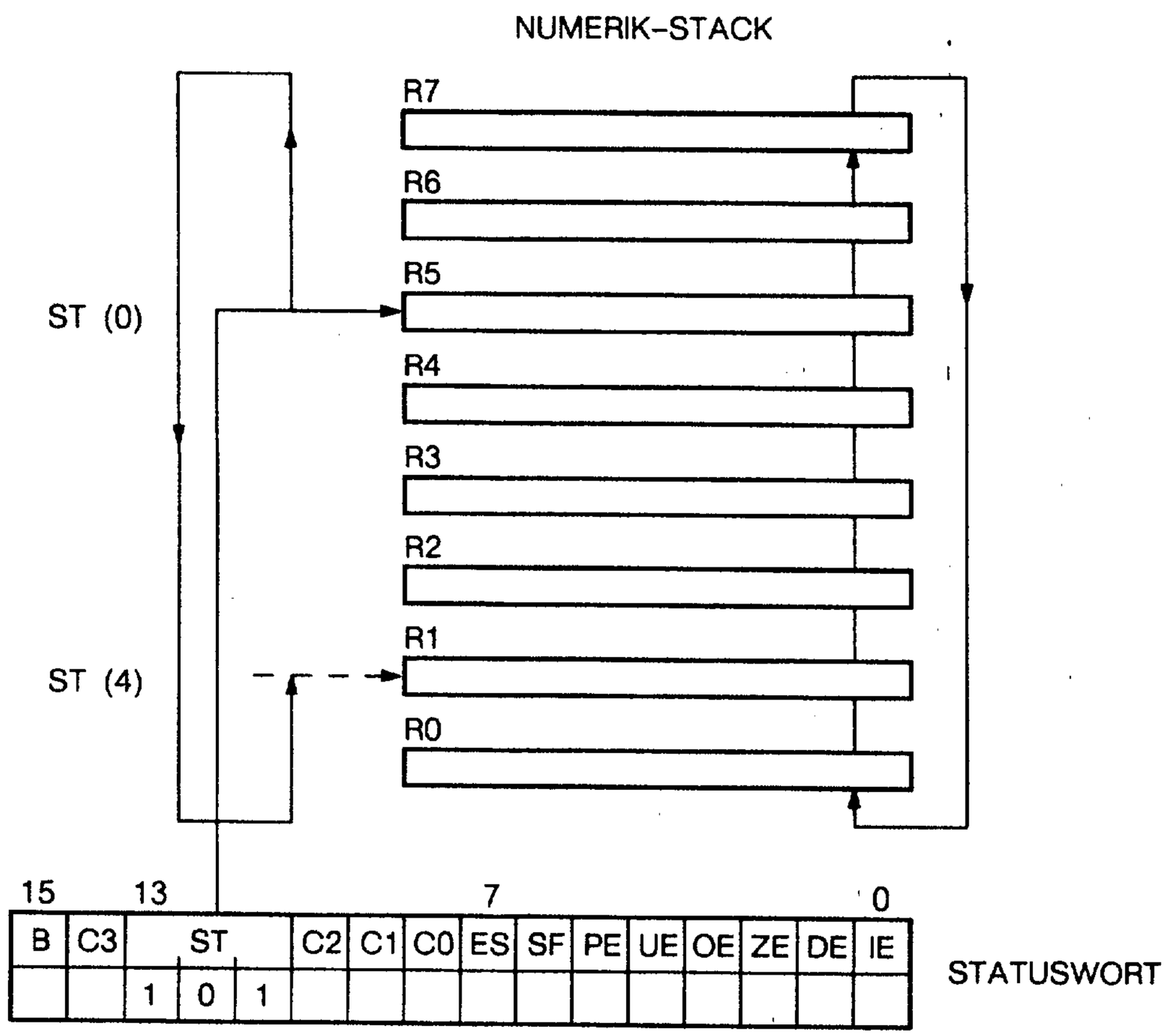
Das ST-Feld (Bits 13..11) enthält eine bestimmte Nummer im Bereich von 000..111 und kennzeichnet damit eines der acht Numerik-Stack-Register als Stack-Top-Register. Die restlichen Numerik-Stack-Register werden dann **relativ** zum augenblicklichen Stack-Top-Register adressiert und zwar auf folgende Weise:

ST(i) mit 0 <= i <= 7

↓

i-tes Numerik-Stack-Register nach dem augenblicklichen Stack-Top-Register adressieren

Beispiel: Numerik-Stack-Register adressieren



Mit den Bits

13	12	11
1	0	1

im FPU-Statuswort wird das Numerik-Stack-Register R5 adressiert. D.h. R5 ist das augenblickliche Stack-Top-Register ST(0).

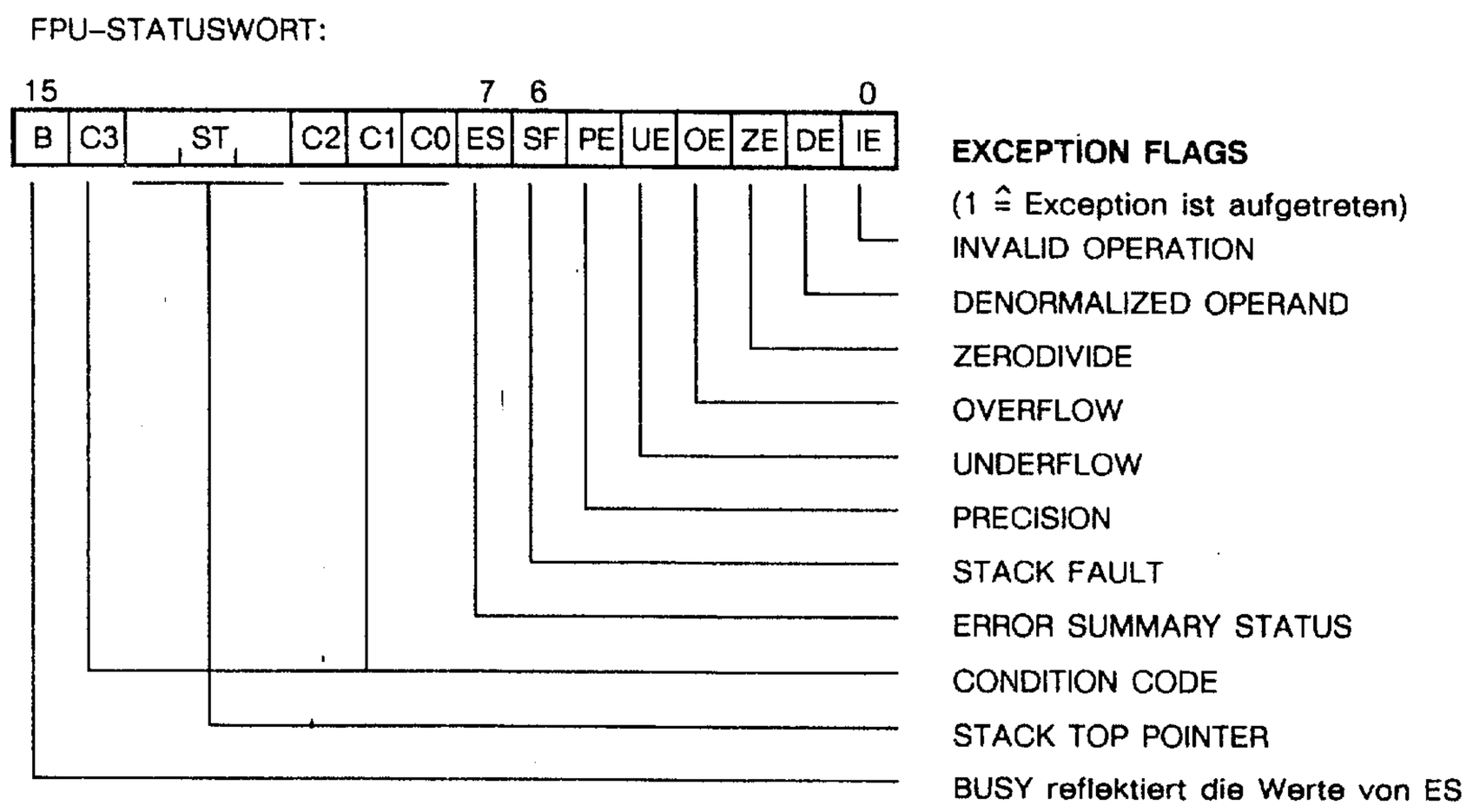
Mit ST(4) wird daraufhin das 4te Numerik-Stack-Register nach ST(0) bzw. nach R5 adressiert. Weil das Nachfolgeregister von R7 das Numerik-Stack-Register R0 ist, adressiert ST(4) das Numerik-Stack-Register R1.

Hinweis:
Nach der Initialisierung der FPU mit dem Befehl FINIT enthält das ST-Feld im FPU-Statuswort den Wert 000. Damit ist das Numerik-Stack-Register R0 als Stack-Top-Register spezifiziert.

8.1.2 FPU-Statuswort

Das 16-Bit-Statuswort gibt umfassend Auskunft über den augenblicklichen Betriebszustand der FPU. Es kann mit den Befehlen FSTSW/FNSTSW (Speichere Statusword), FSTENV/FNSTENV (Speichere Environment) und FSAVE/FNSAVE (Speichere Environment und Numerik-Stack) im Speicher oder mit den Befehlen FSTSW AX/FNSTSW AX im CPU-Register AX abgelegt werden. Durch passende Bitmaskierungen hat dann die CPU die Möglichkeit, den augenblicklichen FPU-Status festzustellen.

Das komplette Layout des FPU-Statusworts zeigt das folgende Bild:



- Das B-Bit (Busy) ist nur noch aus Kompatibilitätsgründen zum 8087 vorhanden. Es reflektiert hier die Inhalte vom ES-Bit (Error-Summary-Status = Bit 6).
- Die vier Condition-Bits C3..C0 verhalten sich ähnlich wie die Flags der CPU. Der 80486 aktualisiert diese Bits abhängig vom Ergebnis arithmetischer Operationen. Sie werden prinzipiell für bedingte Programmverzweigungen benutzt.
- Die Bits 13..11 (ST) zeigen an, welches der vier Numerik-Stack-Register das augenblickliche Top-Stack-Register ist.
- Die Bits 7..0 sind die sogenannten **Exception**-Flags. Sie werden gesetzt, wenn die FPU bei der Ausführung von Floating-Point-Operationen Fehler entdeckt.

1, **Precision**-Exception **PE** (Bit 5)
Die FPU läßt sich so programmieren, daß sie ihre Ergebnisse mit einer Signifikanden-Genauigkeit von 64 Bits im Temporary-Realformat (erweiterte Genauigkeit) oder 53 Bits im LONG-Realformat (doppelte Genauigkeit) oder 24 Bits im SHORT-Realformat (einfache Genauigkeit) liefert. Sollte aber die gewählte Signifikanden-Genauigkeit nicht ausreichen, um das Ergebnis einer Operation **exakt** darzustellen, dann wird das PE-Flag gesetzt (PE=1) und die FPU rundet das Ergebnis.

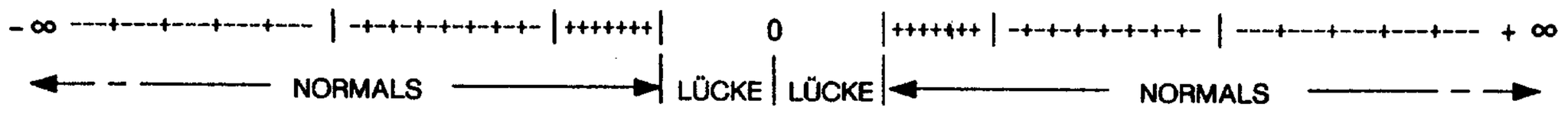
2, **Underflow**-Exception **UE**
(Bit 4) Dieses Flag wird gesetzt (UE=1), wenn nach einer Operation der Exponent des Ergebnisses **zu klein** ist, um im gewünschten Real-Format dargestellt werden zu können.

3, **Overflow**-Exception **OE**
(Bit 3) Dieses Flag wird gesetzt (OE=1), wenn nach einer Operation der Exponent des Ergebnisses **zu groß** ist, um im gewünschten Real-Format dargestellt werden zu können.

4, **Zerodivide**-Exception **ZE** (Bit 2)
Dieses Flag wird gesetzt (ZE=1), wenn versucht wird, einen endlichen Nicht-Null-Operanden durch **0** zu teilen.

5, **Denormalized Operand**-Exception **DE** (Bit 1)
Dieses Flag wird gesetzt (DE=1), wenn versucht wird, einen sogenannten Denormal-Operanden zu verarbeiten. Was ist darunter zu verstehen?
In einem Floating-Point-Zahlensystem ist die Dichte der darstellbaren Zahlen in der Nähe von **Null** größer als in der Nähe der größten darstellbaren Zahl. Daher leiden Maschinen, die **keine** Denormals unterstützen unter einer großen Lücke zwischen Null und der nächsten darstellbaren Zahl (Normal). Wie das folgende Bild verdeutlicht, gibt es zwei solcher Lücken und zwar eine auf der positiven Seite von Null und eine auf der negativen Seite von Null.

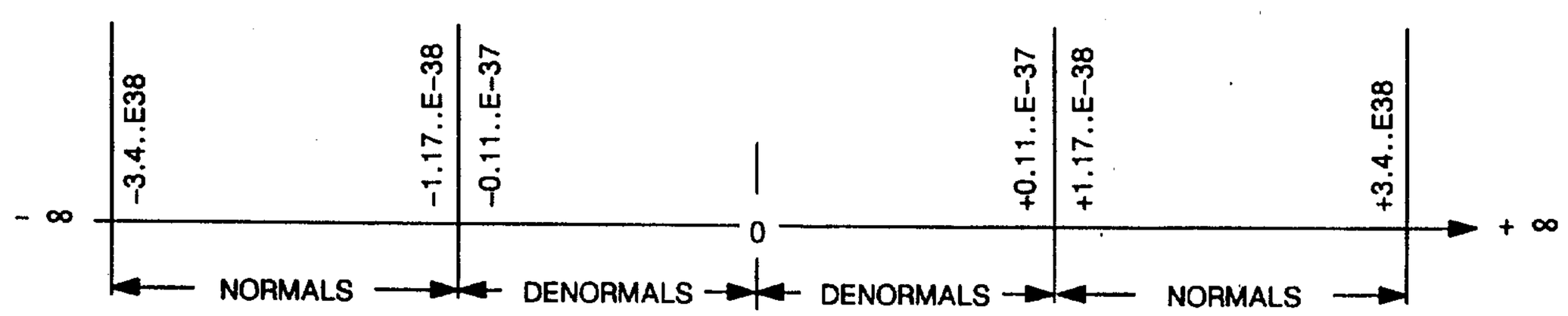
Floating-Point-Zahlensystem **ohne** Denormals:



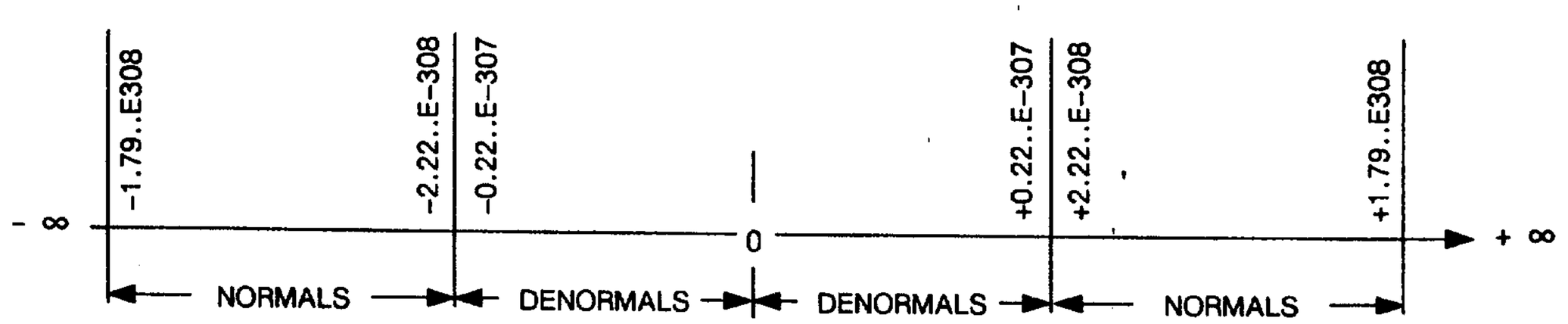
Die Lücke zwischen Null und der nächsten darstellbaren Zahl unterscheidet sich von der Lücke zwischen dieser Zahl und der nächst folgenden darstellbaren Zahl ungefähr um den Faktor 8.4×10^6 beim **SHORT**-Realformat (32 Bits), 4.5×10^{15} beim **LONG**-Realformat (64 Bits) und 9.2×10^{18} beim **Temporary**-Realformat (80 Bits). Die Folge davon ist, daß sich Fehleranalysen erheblich verkomplizieren.

Um dies zu vermeiden, benutzt die FPU diese Lücken für die Denormals. Wie unter Berücksichtigung der Denormals das Floating-Point-Zahlensystem im SHORT-Realformat, im LONG-Realformat und im Temporary-Realformat aussieht, illustrieren die folgenden Bilder:

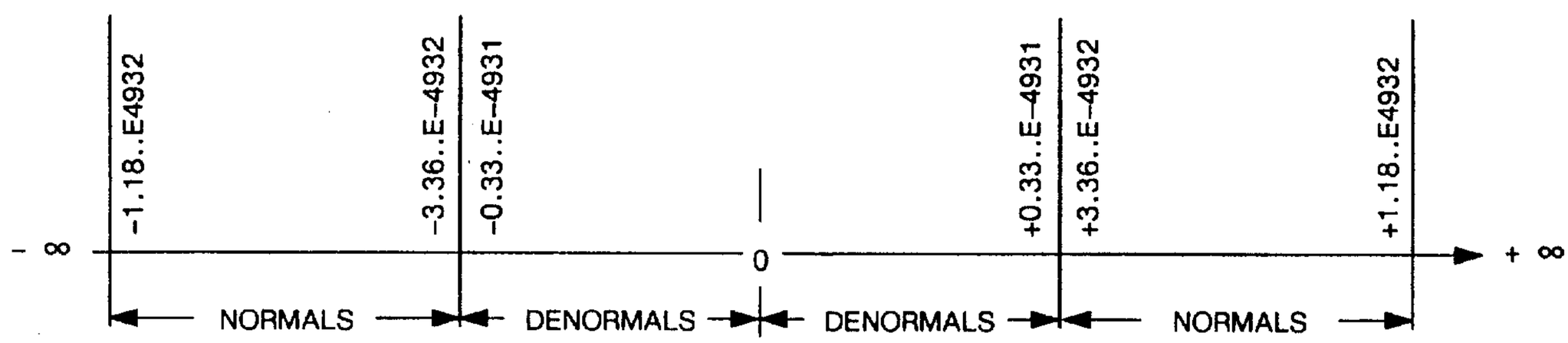
Floating-Point-Zahlensystem **mit** Denormals
SHORT-Realformat:



Floating-Point-Zahlensystem **mit** Denormals
Long-Realformat:



Floating-Point-Zahlensystem **mit** Denormals
Temporary-Realformat:



Wie wird nun ein Denormal erzeugt? Tritt nach einer Floating-Point-Operation ein **Underflow** auf, dann kann der wahre Exponent des Ergebnisses (weil zu klein) nicht im gewünschten Realformat dargestellt werden. In diesem Fall bildet die FPU aus dem wahren Operations-Ergebnis ein wertgleiches **Denormal**. Jedes Denormal ist durch den Bit-String 0000 0000 binär in seinem Exponenten und durch einen Nicht-Null-Signifikanden identifiziert. Betrachten wir dazu die Wertebereiche der darstellbaren Exponenten in den **drei** Real-Zahlenformaten.

- Im SHORT-Realformat liegt der Wertebereich der darstellbaren Exponenten zwischen –126 dezimal und +127 dezimal,
- im LONG-Realformat liegt er zwischen –1022 dezimal und +1023 dezimal und
- im Temporary-Realformat liegt er zwischen –16382 dezimal und +16383 dezimal.

Ist z. B. nach einer Floating-Point-Operation der wahre Exponent des Ergebnisses –129 dezimal, dann kann dieser Exponent zwar im LONG-Realformat oder im TEMPORARY-Realformat dargestellt werden, **nicht** aber im SHORT-Realformat.

Wenn aber das Ergebnis im SHORT-Realformat dargestellt werden soll, dann leitet die FPU den sogenannten **Denormalisierungs-Prozess** ein. Dabei heißt denormalisieren, den zu kleinen Exponenten in den darstellbaren Bereich des gewünschten Realformats bringen. Auf welche Weise dies geschieht, illustriert die folgende Tabelle:

Operation	Vorzeichen	Exponent	Signifikand
Wahres Ergebnis	0	–129	1▲01011100 00
Denormalisieren	0	–128	0▲101011100 00
Denormalisieren	0	–127	0▲0101011100 . . . 00
Denormalisieren	0	–126	0▲00101011100 . . 00
Denormalisiertes Ergebnis	0	–126	0▲00101011100 . . 00

Der Exponent, beginnend bei –129 dezimal wird solange inkrementiert, bis er den im SHORT-Realformat darstellbaren Wert von –126 erreicht hat. Bei jedem Inkrementiervorgang wird **gleichzeitig** der Signifikand um eine Binärstelle nach rechts verschoben und die führende Stelle mit einer Null besetzt. So hat das denormalisierte Operationsergebnis den gleichen Wert wie das wahre Operationsergebnis.

Bevor aber das denormalisierte Operationsergebnis gespeichert wird, rundet die FPU den Signifikanden auf 24 Bits (einfache Genauigkeit im SHORT-Realformat) und addiert zum Exponenten +126 dezimal. Es entsteht ein Denormal, das durch einen Nicht-Null-Signifikanden und dem typischen Bit-String 0000 0000 binär identifiziert ist.

7, **Invalid Operation**-Exception **IE** (Bit 0)

Diese Ausnahme kann bei **zwei** verschiedenen Klassen von Floating-Point-Operationen vorkommen, nämlich bei

- Numerik-Stack-Operationen oder bei
- arithmetischen Operationen.

Welche der beiden Klassen den Numerik-Fehler verursacht hat, zeigt das **Stack Fault**-Flag **SF** (Bit 6) im Statuswort an.

- Wenn **SF=1** ist, dann hat die Floating-Point-Operation entweder zu einem Numerik-Stack-Overflow oder zu einem Numerik-Stack-Underflow geführt.
- Wenn **SF=0** ist, dann wurde der Numerik-Fehler von einer ungültigen arithmetischen Operation verursacht, wie z.B. die Multiplikation **Unendlich x 0** oder die Division **0/0** usw.

8, **Stack Fault**-Exception **SF** (Bit 6)
Diese Flag wird gesetzt (SF=1), wenn es bei einer Floating-Point-Operation entweder zu einem Numerik-Stack-**Overflow** oder zu einem Numerik-Stack-**Underflow** gekommen ist. Die Unterscheidung trifft das Condition Bit C1 im Statuswort. Es hat hier die Bedeutung von Overflow/Underflow* (O/U*)

- Wenn **C1** (O/U*) = **1** ist, dann wird ein Numerik-Stack- **Overflow** angezeigt. Dies bedeutet, daß ein Befehl versucht hat, in ein **nicht-leeres** Numerik-Stack-Register zu schreiben.
- Wenn **C1** (O/U*) = **0** ist, dann wird ein Numerik-Stack-**Underflow** angezeigt. Dies bedeutet, daß ein Befehl versucht hat, ein **leeres** Numerik-Stack-Register zu lesen.

9, **Error Summary Status ES** (Bit 7)
Jedes der sechs Exception-Bits PE/UE/OE/ZE/DE/IE im FPU-Statuswort korrespondiert individuell mit einem bestimmten Exception-**Masken**-Bit PM/UM/OM/ZM/DM/IM im FPU-Kontrollwort. So korrespondiert

- PE mit PM (Precision-Mask), UE mit UM (Underflow-Mask),
- OE mit OM (Overflow-Mask), ZE mit ZM (Zerodivide-Mask),
- DE mit DM (Denormalized Operand-Mask) und
- IE mit IM (Invalid Operation-Mask).

Treten bei Floating-Point-Operationen Exceptions auf, dann bestimmen die Masken-Bits im FPU-Kontrollwort das weitere Verhalten der FPU.

- Ist eine Numerik-Exception **maskiert**, d.h. ist das korrespondierende Masken-Bit im Kontrollwort = **1**, dann produziert der Prozessor ein angemessenes Ergebnis und erlaubt der FPU die ungestörte Fortsetzung des Numerik-Programms. Führt z.B. die FPU die ungültige arithmetische Operation **0/0** aus, dann liefert sie als Ergebnis den

Spezialwert **QNaN Indefinite** (Quiet Not-a-Number Indefinite). Dieser Spezialwert hat im SHORT-Realformat den Code FF000000H, im LONG-Realformat den Code FFF8000000000000H und im TEMPORARY-Realformat den Code FFFFC000000000000000H.

- Ist eine Numerik-Exception **nicht maskiert**, d.h ist das korrespondierende Masken-Bit im Kontrollwort = **0**, dann führt die Exception zum Aufruf des Software-Exception-Handlers. Dies kann entweder vom OnChip-**Interrupt-Vektor 16** veranlaßt werden oder von einem **externen Interrupt**.

  Was hat dies alles mit dem Error-Summary-Status-Flag zu tun? Wenn es zu Exceptions kommt, dann wird das ES-Flaggesetzt (ES=1), wenn die Exceptions **nicht maskiert** sind. Sind dagegen die Exceptions **maskiert**, dann wird das ES-Flag gelöscht (ES=0).

8.1.2.1 Liste der Exceptions

Die folgende Tabelle enthält die Befehls-Mnemoniks der FPU in alphabetischer Reihenfolge. Für jede Mnemonik sind diejenigen Exceptions aufgelistet, die bei dem betreffenden Befehl vorkommen können.

Mnemonik	SF	IE	DE	ZE	OE	UE	PE
F2XM1	X	X	X			X	X
FABS	X						
FADD(P)	X	X	X		X	X	X
FBLD	X						
FBSTP	X	X					X
FCHS	X						
FCLEX							
FCOM(P)(P)	X	X	X				
FCOS	X	X	X			X	X
FDECSTP							
FDIV(R)(P)	X	X	X	X	X	X	X
FFREE							
FIADD	X	X	X		X	X	X
FICOM(P)	X	X	X				
FIDIV	X	X	X	X		X	X
FIDIVR	X	X	X	X	X	X	X
FILD	X						
FIMUL	X	X	X		X	X	X
FINCSTP							

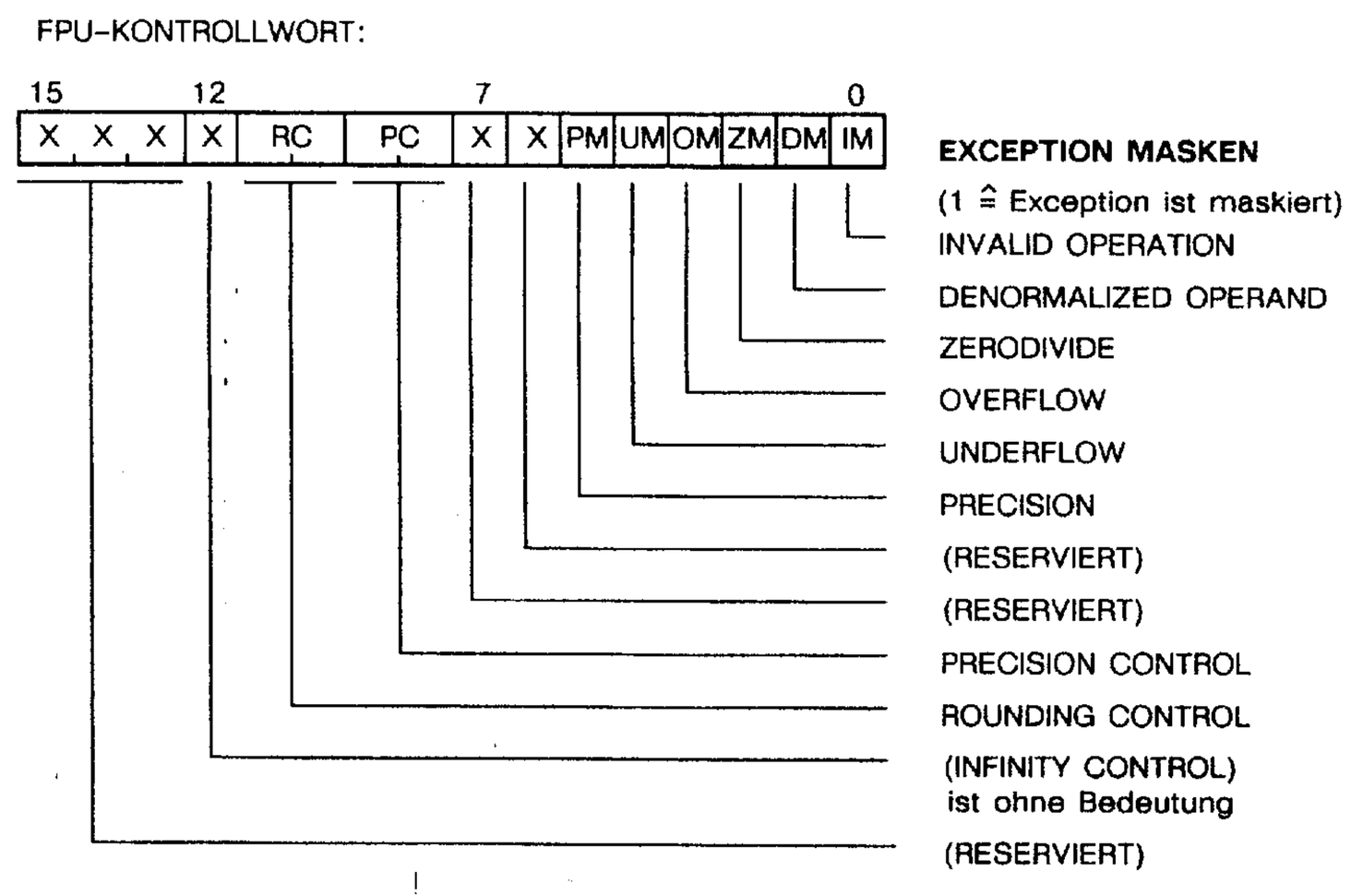
Mnemonik	SF	IE	DE	ZE	0E	UE	PE
FINIT							
FIST(P)	X	X					X
FISUB(R)	X	X	X		X	X	X
FLD temporary oder Stack	X						
FLD short oder long	X	X	X				
FLD1	X						
FLDCW	X	X	X	X	X	X	X
FLDENV	X	X	X	X	X	X	X
FLDL2E	X						
FLDL2T	X						
FLDLG2	X						
FLDLN2	X						
FLDPI	X						
FLDZ	X						
FMUL(P)	X	X	X		X	X	X
FNOP							
FPATAN	X	X	X			X	X
FPREM	X	X	X			X	
FPREM1	X	X	X			X	
FPTAN	X	X	X			X	X
FRNDINT	X	X	X				X
FRSTOR	X	X	X	X	X	X	X
FSAVE							
FSCALE	X	X	X		X	X	X
FSIN	X	X	X			X	X
FSINCOS	X	X	X			X	X
FSQRT	X	X	X				X
FST(P) temporary oder Stack	X						
FST(P) short oder long	X	X	X		X	X	X
FSTCW							
FSTENV							

Mnemonik	SF	IE	DE	ZE	0E	UE	PE
FSTSW (AX)							
FSUB(R)(P)	X	X	X		X	X	X
FTST	X	X	X				
FUCOM(P)(P)	X	X	X				
FWAIT							
FXAM							
FXCH	X						
FXTRACT	X	X	X	X			
FYL2X	X	X	X	X	X	X	X
FYL2XP1	X	X	X			X	X

Schlüssel: SF – Ungültige Operation durch Numerik-Stack-Overflow/Underflow
IE – Andere ungültige Operation
DE – Denormal-Operand
ZE – Division durch Null
OE – Overflow
UE – Underflow
PE – Ungenaues Ergebnis

8.1.3 FPU-Kontrollwort

Über das Kontrollwort läßt sich das **Betriebsverhalten** der FPU bestimmen. Zu diesem Zweck wird das Kontrollwort mit einem passenden 16-Bit-Wert (Befehl FLDCW source) geladen. Das folgende Bild zeigt das Format des FPU-Kontrollworts:



Im **Low**-Byte sind die sechs Exception-Masken-Bits PM/UM/OM/ZM/DM/IM enthalten, die das Verhalten der FPU bei Numerik-Exceptions bestimmen und im **High**-Byte gibt es zwei Felder, mit deren Hilfe

- der Rundungsmodus (Rounding Control RC) und
- die Signifikanden-Genauigkeit (Precision Control PC)

festgelegt werden kann.

1, **Rounding Control RC** (Bits 11,10)
Wenn nach einer Floating-Point-Operation das **wahre** Operations-ergebnis **nicht** exakt im gewünschten Realformat dargestellt werden kann, dann muß das Operationsergebnis gerundet werden. Zu diesem Zweck stellt die FPU vier verschiedene Rundungsarten zur Verfügung und zwar

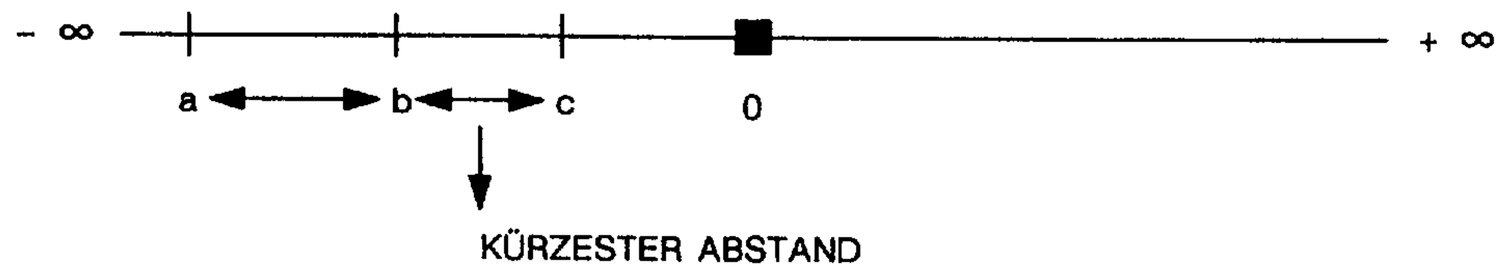
- Runden zum nähesten Wert ;RC=00
- Abrunden in Richtung –Unendlich ;RC=01
- Aufrunden in Richtung +Unendlich ;RC=10
- Abschneiden in Richtung 0 (Chop) ;RC=11

Nehmen wir an, das wahre Operationsergebnis ist b. Wenn b nicht dargestellt werden kann, dann liefert die FPU zwei Realzahlen a und c, die das wahre Operationsergebnis im kürzesten Abstand umschließen:

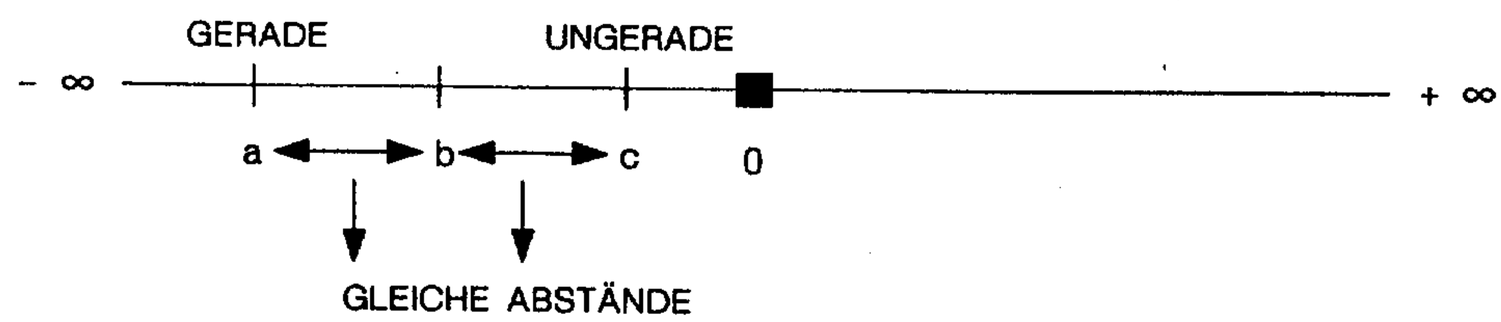
$$(a < b < c)$$

Abhängig von der gewählten Rundungsart, liefert die FPU als Ergebnis dann entweder a oder c.

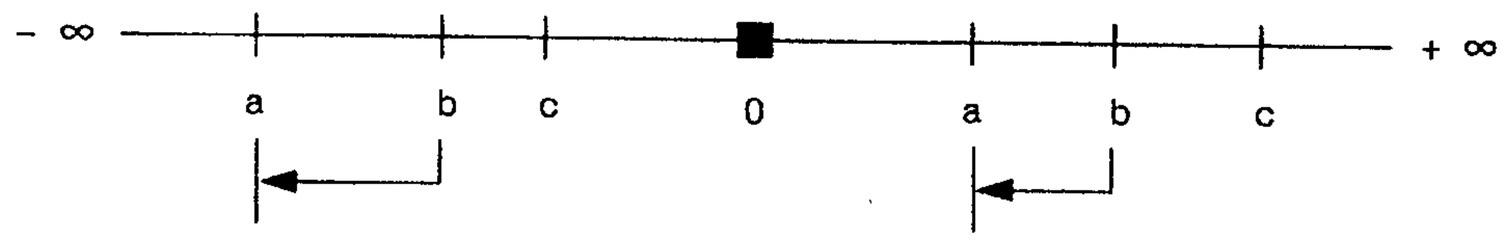
- **Runden zum nähesten Wert**
Bei dieser Rundungsart wird das wahre Operationsergebnis b auf denjenigen Wert auf- oder abgerundet, der b am nähesten liegt. Im folgenden Beispiel ist dies c.



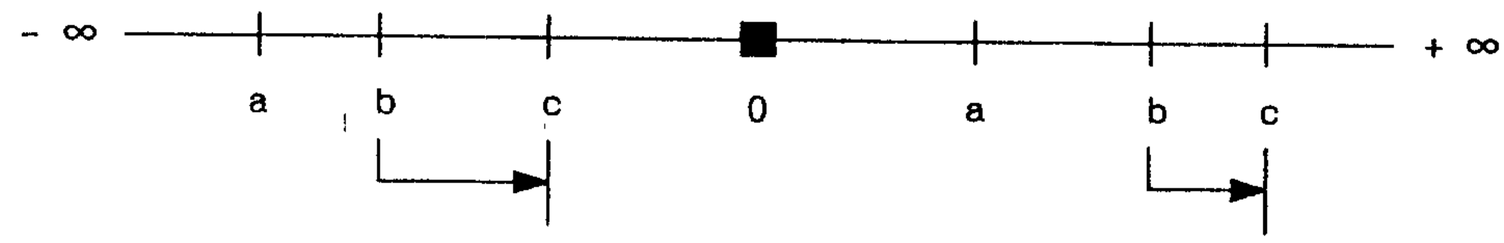
Liegen die Realzahlen a und c vom wahren Operationsergebnis b **gleichweit** entfernt, dann wird b auf den **geraden** Wert auf- oder abgerundet. Im folgenden Beispiel ist dies a.



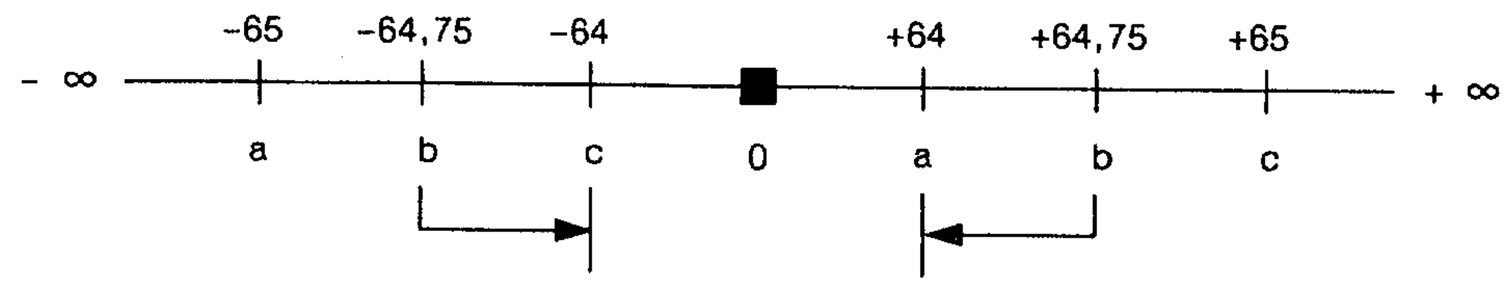
– **Abrunden**
Bei dieser Rundungsart wird immer in Richtung **-Unendlich** abgerundet. Im folgenden Bild ist dies illustriert. Hier liefert die FPU als Operationsergebnis den Rundungswert a.



– **Aufrunden**
Bei dieser Rundungsart wird immer in Richtung **+Unendlich** aufgerundet. Im folgenden Bild ist dies illustriert. Hier liefert die FPU als Operationsergebnis den Rundungswert c.



– **Abschneiden** (Chop)
Diese Rundungsart wird bevorzugt in der INTEGER-Arithmetik benutzt. Der gebrochene Anteil der Realzahlen wird „abgeschnitten", so daß sich ganze Zahlen ergeben, die in Richtung 0 liegen. Im folgenden Beispiel sind dies die Zahlen c und a.



2, **Precision Control PC** (Bits 9,8)
Die FPU kann ihre internen Berechnungen mit einer Signifikanden-Genauigkeit von 64 Bits, 53 Bits oder 24 Bits durchführen. Mit welcher Genauigkeit die FPU arbeitet, bestimmt das PC-Feld in Übereinstimmung mit der folgenden Tabelle:

PC	Signifikanden-Genauigkeit	Format
00	24 Bits	SHORT-Real
01	Reserviert	
10	53 Bits	LONG-Real
11	64 Bits	Temporary-Real

Nach der Initialisierung (Befehl FINIT) arbeitet die FPU mit einer Signifikanden-Genauigkeit von 64 Bits (PC=11). Wenn aber eine reduzierte Genauigkeit spezifiziert ist, dann setzt die FPU die nicht-benutzten Bits des Signifikanden auf Null.

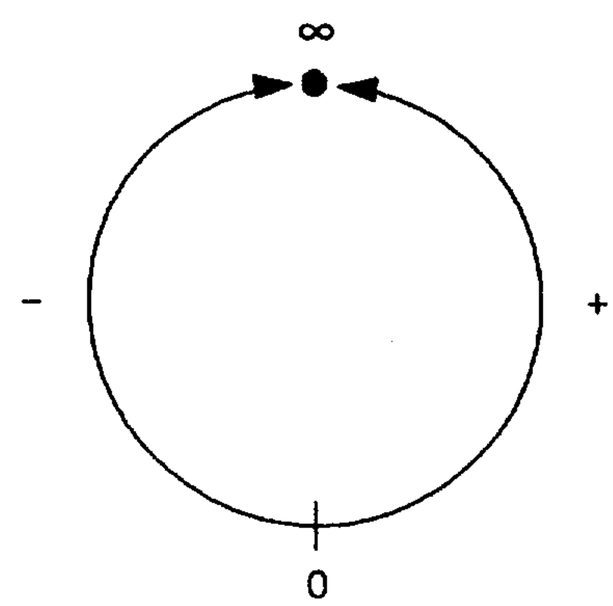
Hinweis:
Das PC-Feld beeinflußt nur die Befehle FADD (Addition), FSUB (Subtraktion), FMUL (Multiplikation), FDIV (Division) und FSQRT (Quadratwurzel).

3, **Infinity Control IC** (Bit 12)

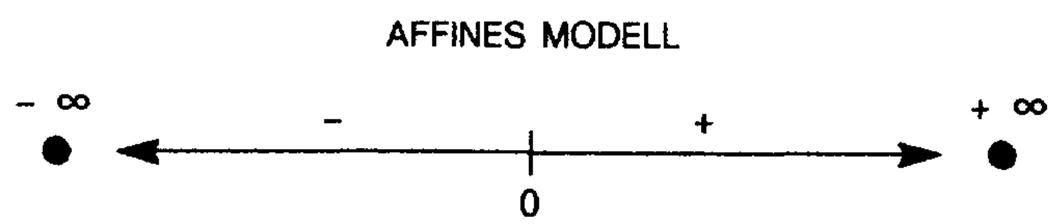
Bei den Numerik-Prozessoren 8087 und 80287 bestimmt dieses Bit, wie ein System von Realzahlen im Bereich von **–Unendlich** bis **+Unendlich** geschlossen dargestellt werden soll. Zwei Modelle stehen dort zur Auswahl, nämlich

- das **projektive** Modell (IC=0) und
- das **affine** Modell (IC=1).

Beim projektiven Modell behandeln die Numerik-Prozessoren 8087/80287 die Spezialwerte –Unendlich und +Unendlich als **vorzeichenlose** Unendlichkeiten. Im folgenden Bild ist dieses Modell illustriert:



Im Gegensatz dazu werden beim affinen Modell die **Vorzeichen** der Spezialgrößen –Unendlich und +Unendlich von den Numerik-Prozessoren 8087/80287 **berücksichtigt.** Im folgenden Bild ist dieses Modell illustriert:



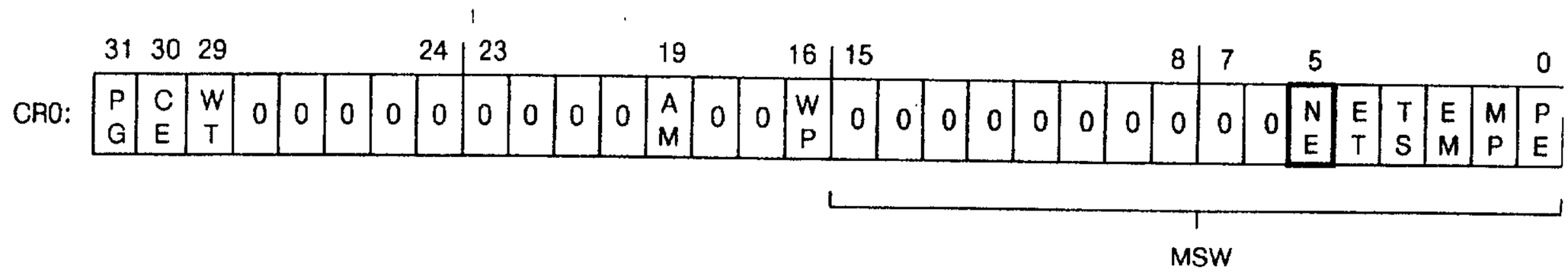
Die **FPU** des 80486 unterstützt genauso wie der 80387 nur noch das **affine** Modell. So kann zwar das IC-Bit programmiert werden, es hat aber **keinen** Einfluß auf die FPU-Operationen.

4, Die **Masken**-Bits **PM/UM/OM/ZM/DM/IM** (Bits 5..0)
Wenn während der Ausführung eines Numerik-Programms Exceptions auftreten, dann kann der Prozessor durch zwei unterschiedliche Aktionen darauf reagieren:

- Sind die Exceptions **maskiert** (PM/UM/OM/ZM/DM/IM=**1**), dann behandelt die FPU diese Exceptions selbst. D.h. sie produziert ein angemessenes Ergebnis und führt das Numerik-Programm ungehindert fort.

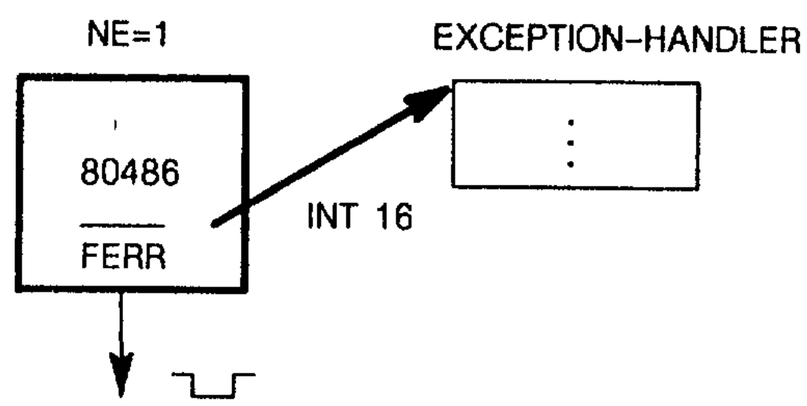
– Sind dagegen die Exceptions **nicht maskiert** (PM/UM/OM/ZM/DM/IM=**0**), dann behandelt die FPU diese Exceptions **nicht** selbst.
  Statt dessen wird ein Exception-Handler aufgerufen, der diese Aufgabe übernimmt.

Wenn es zu **nicht-maskierten** Exceptions kommt, dann kann der Aufruf des Exception-Handlers entweder vom OnChip-**Interrupt-Vektor 16** oder von einem **externen Interrupt** veranlaßt werden. Die Entscheidung darüber trifft das **NE**-Bit (Numeric Exception) im Control-Register CR0. Die Position dieses Bits ist im CR0-Layout zu sehen:

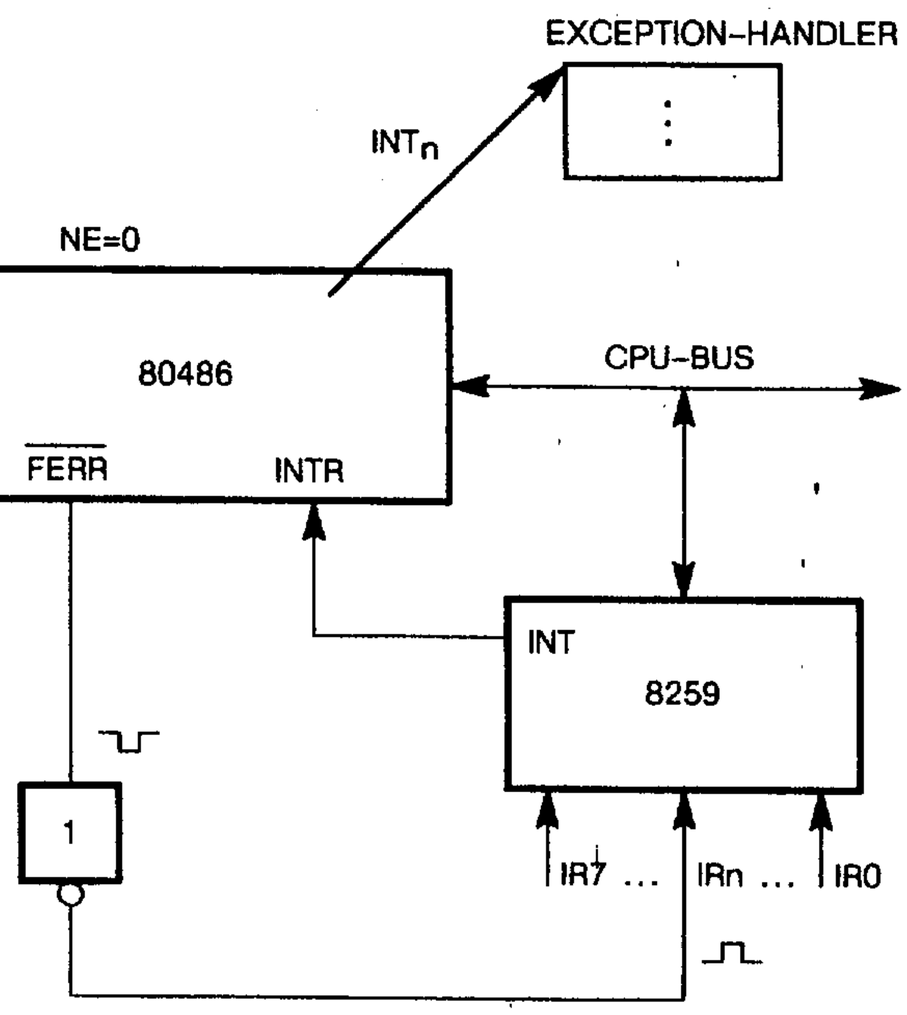


Ungeachtet vom NE-Wert zeigt der Prozessor jeden unmaskierten Floating-Point-Fehler durch ein LOW-Signal an seinem **FERR*** -Ausgang an (Floating-Point Error, * bedeutet LOW-aktiv).

– Wenn **NE = 1** ist, dann wird der Exception-Handler vom Interrupt 16, dem sogenannten Floating-Point-Error-Interrupt, aufgerufen. FERR* ist hier ohne Bedeutung. Dies ist im folgenden Bild illustriert:

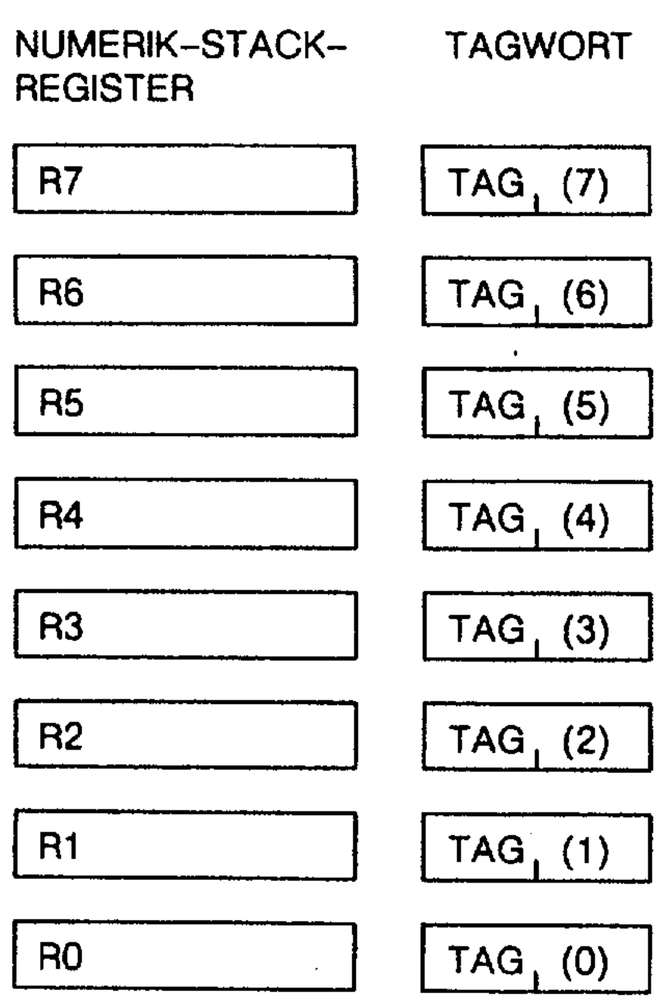


– Wenn **NE = 0** ist, dann wird die **DOS-kompatible** Art der Fehlerbehandlung unterstützt. Hier „friert" ein unmaskierter Floating-Point-Fehler den Prozessor ein und er wartet, bis eine externe Interruptanforderung an seinem INTR-Eingang (Interupt Request) eintrifft. Diese Anforderung kommt vom Interrupt-Controller 8259. Wie das folgende Bild zeigt, wird FERR* der CPU als Interrupt-Anforderungssignal benutzt und ist daher über einen Inverter mit dem Interrupt-Request-Eingang IRn (n = 7..0) des Interrupt-Controllers verbunden. Trifft wegen FERR* = LOW an IRn eine Interrupt-Anforderung ein, dann leitet der Interrupt-Controller diese Anforderung an den INTR-Eingang der CPU weiter. Die CPU holt sich daraufhin vom ICW2-Register des Interrupt-Controllers die zu IRn gehörige Vektornummer n und verzweigt mit INT(erupt) n über die Interrupt-Deskriptor-Tabelle (IDT) zum Exception-Handler.



8.1.4 Das FPU-Tagwort

Wie das folgende Bild zeigt, ist jedem Numerik-Stack-Register ein 2-Bit-**Tagfeld** zugeordnet, das den Inhalt des Numerik-Register **charakterisiert.**



Tag-Wert	Numerik-Register-Inhalt
00	gültig
01	Null
10	spezielle Werte: NaN (Not-a-Number) nicht-unterstützte Bitmuster Unendlich Denormal
11	leer

An den Tag-Werten hat die FPU ein anderes Interesse als der Programmierer. Der Programmierer benutzt die Tag-Informationen, um **exakt** die Numerik-Register-Inhalte festzustellen. Dagegen benutzt die FPU diese Tag-Informationen, um zwischen **leeren** und **nicht-leeren** Numerik-Stack-Registern zu unterscheiden.

- Eine leeres Numerik-Stack-Register ist durch den Tag-Wert 11 gekennzeichnet.
- Ein nicht-leeres Numerik-Stack-Register ist durch die Tag-Werte 00 oder 01 oder 10 gekennzeichnet.

Der FPU reicht diese grobe Unterscheidung, denn sie muß lediglich bei **ungültigen** Operationen feststellen, ob es sich um einen Stack-**Overflow** oder um einen Stack-**Underflow** handelt. Der Programmierer kann die Tag-Informationen über die Befehle F(N)SAVE (Speichere Numerik-Status) oder F(N)STENV (Speichere Numerik-Environment) bekommen. **Bevor** diese Befehle ausgeführt werden, sorgen sie dafür, daß die Tag-Felder in Übereinstimmung mit den aktuellen Inhalten der Numerik-Stack-Register die richtigen Werte enthalten.

8.1.5 Befehls- und Operanden-Pointer

Die Befehls- und Operanden-Pointer sind besonders für den Exception-Handler von Interesse. Sie sind im Prozessor enthalten und sind durch die Befehle FLDENV (Lade Environment), FSTENV (Speichere Environment), FSAVE (Speichere Status) und FRSTOR (Lade Status) auch der FPU zugänglich.

Immer wenn die CPU einen ESC-Befehl dekodiert, **rettet** sie den **Befehls-Pointer**, den **Operanden-Pointer** und den **Operationscode** des betreffenden Numerikbefehls in den Chip.

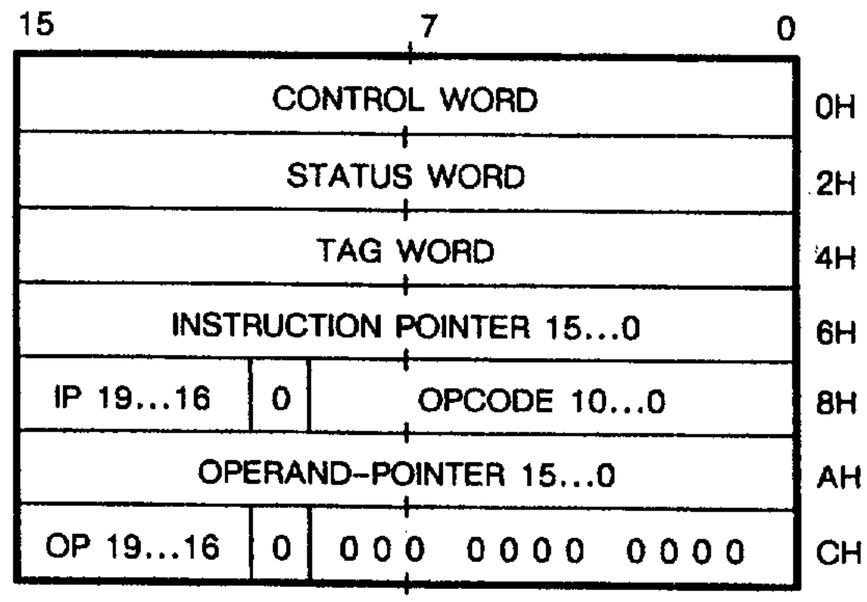
Wenn das sogenannte „Environment" der FPU mit FSTENV im Speicher hinterlegt wird, erscheinen die Befehls- und Operanden-Pointer in einem von **vier** Speicher-Formaten. Diese Formate sind abhängig

- von der Betriebsart des 80486 und
- vom **USE**-Attribut des Segments, in dem das „Environment" aufbewahrt werden soll.

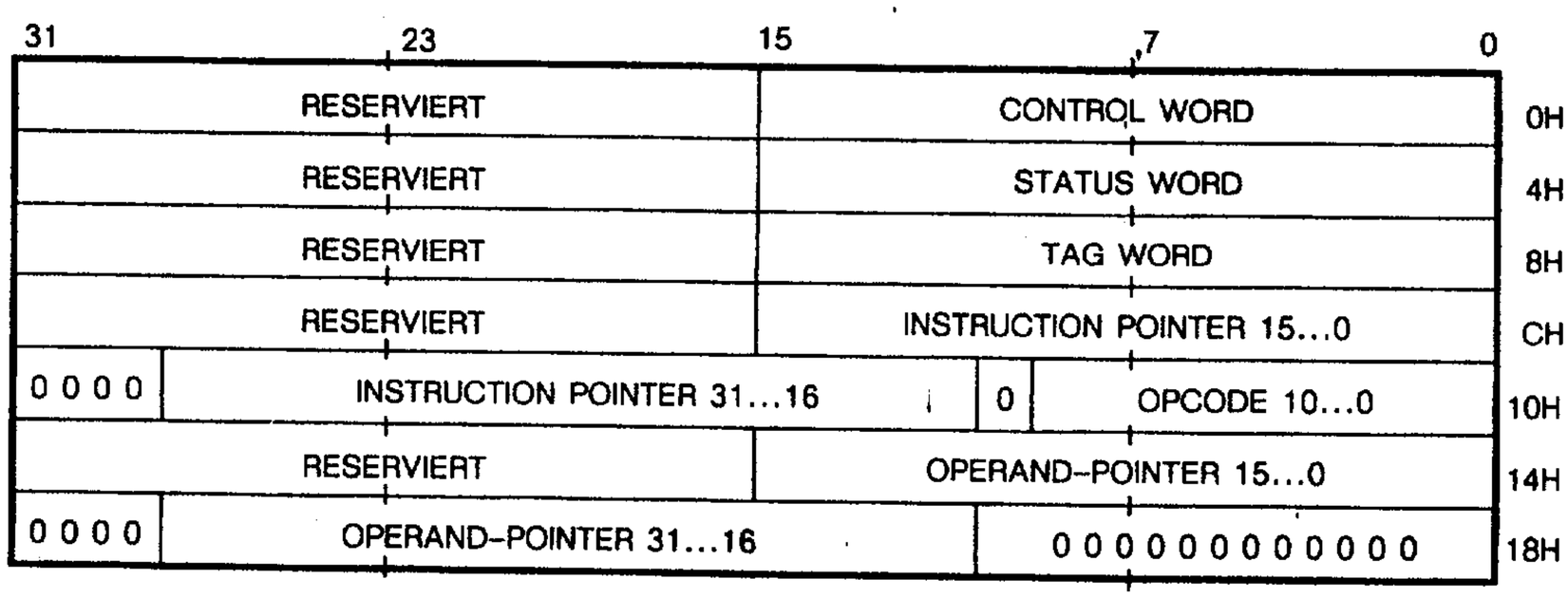
Sowohl im „Real Address-Mode“ als auch im „Protected-Mode“ des 80486 kann das „Environment“ entweder in einem **USE16**- oder in einem **USE32**-Segment gespeichert werden. Dabei benutzt die CPU in USE16-Segmenten 16-Bit-Operanden und 16-Bit-Adreßpointer und in USE32-Segmenten 32-Bit-Operanden und 32-Bit-Adreßpointer. Wenn der 80486 im virtuellen 8086-Mode arbeitet, erscheinen die Befehls- und Operanden-Pointer im Speicher-Format des 16-Bit-“Real Address-Modes“.

Die folgenden Bilder zeigen die Lage der Befehls- und Operanden-Pointer in allen vier 80486-Speicher-Formaten, nachdem der Befehl FSTENV ausgeführt worden ist:

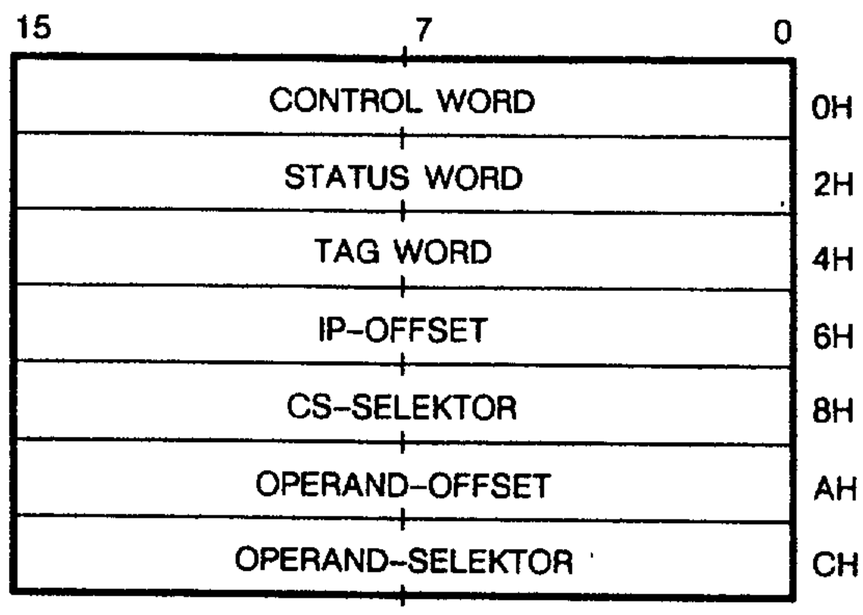
FORMAT IM 16-BIT-"REAL ADDRESS-MODE" UND VIRTUELLEM 8086-MODE
(USE16-SEGMENTE)



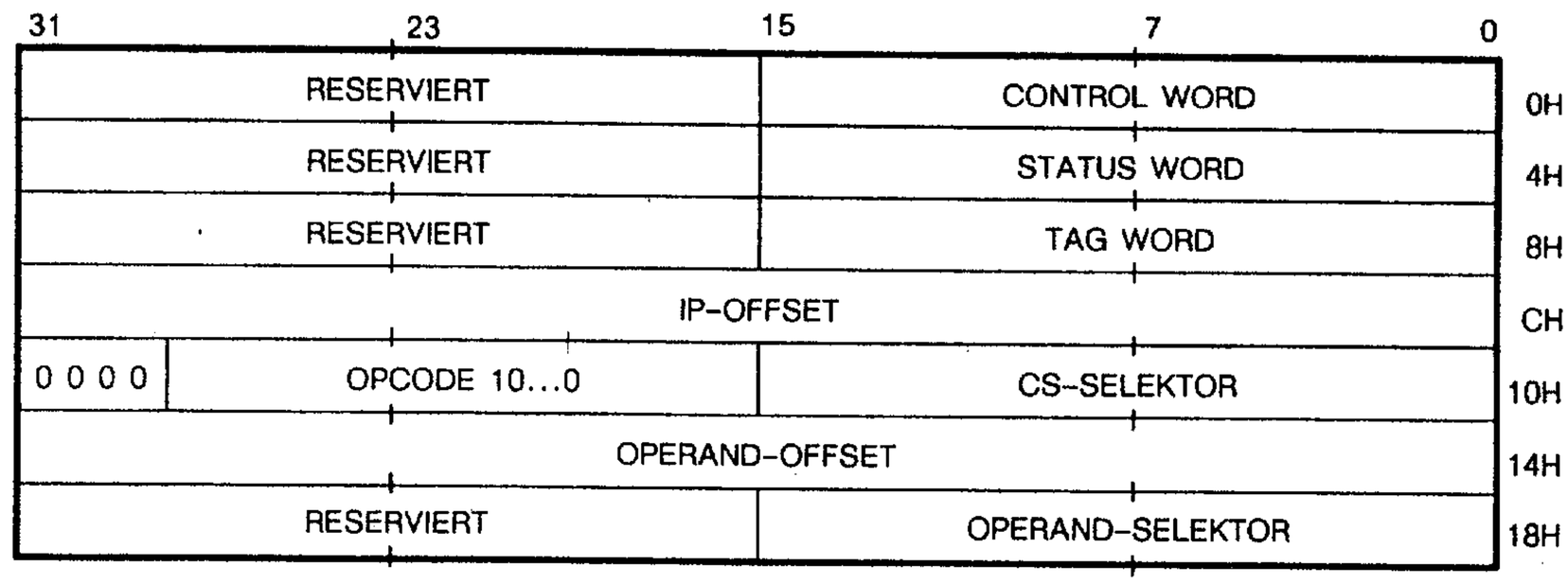
FORMAT IM 32-BIT-"REAL ADDRESS-MODE"
(USE32-SEGMENTE)



FORMAT IM 16-BIT-"PROTECTED-MODE"
(USE16-SEGMENTE)



FORMAT IM 32-BIT-"PROTECTED-MODE"
(USE32-SEGMENTE)



Die Befehls-Pointer (Instruction Pointer) zeigen immer auf das **führende** „Prefix"-Byte (wenn vorhanden) des ESC-Befehls. Führt der Prozessor die Steuerbefehle FINIT, FLDCW, FSTCW, FSTSW, FCLEX, FSTENV, FLDENV, FSAVE, FRSTOR und FWAIT aus, dann werden im Gegensatz zu allen anderen Befehlen die Operanden-Pointer **nicht** beeinflußt. Sollte aber der vorangegangene ESC-Befehl **keinen** Speicheroperanden haben, ist bei allen Befehlen, mit Ausnahme der oben aufgelisteten, der Wert des Operanden-Pointers **undefiniert**.

8.2 Datentypen und Formate

Die FPU kennt **sieben** verschiedene Datentypen, die in **drei** Klassen eingeteilt sind:

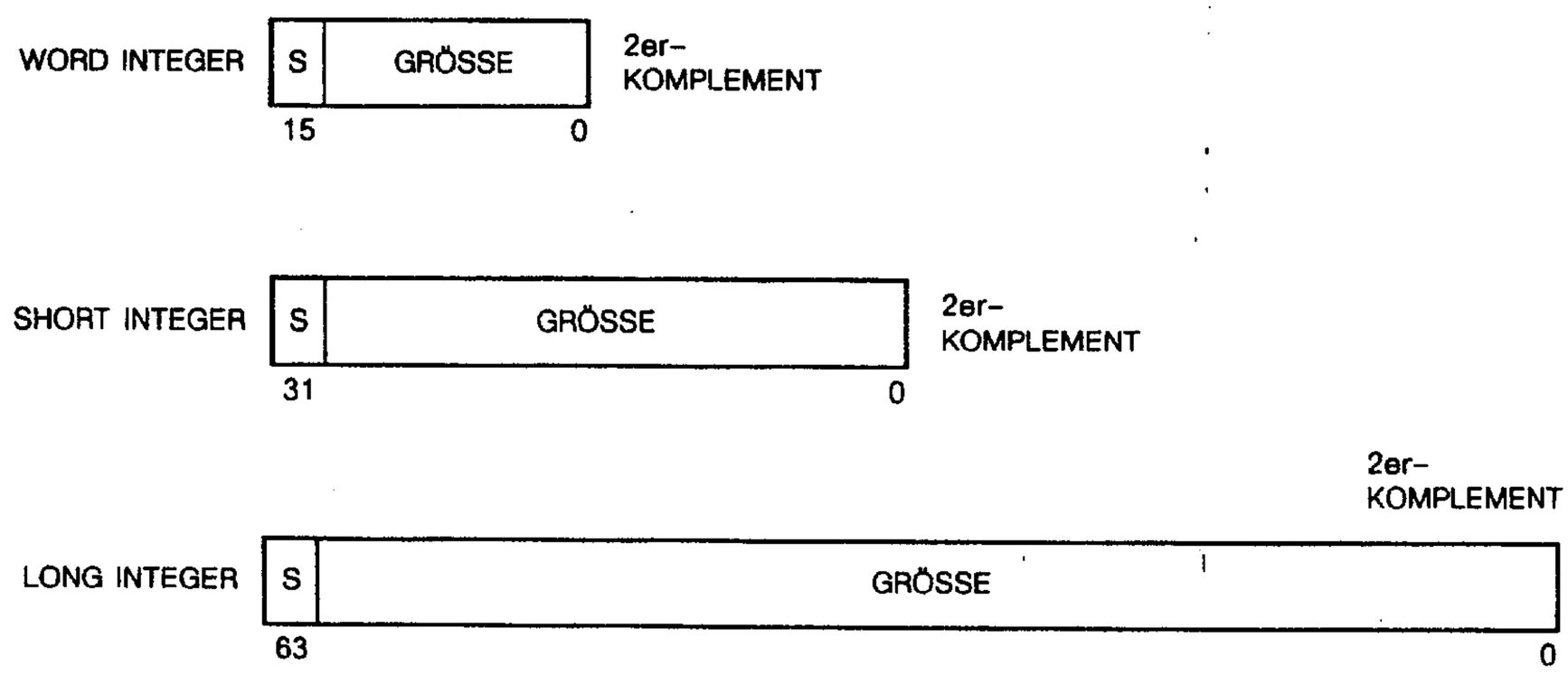
- Binär Integer,
- Packed Decimal Integer und
- Binär Real

8.2.1 Binär Integer

Die drei Integer-Formate sind bis auf ihre Länge identisch und existieren nur im Speicher. Wenn sie von der FPU benutzt werden, dann konvertiert sie die FPU automatisch in das interne 80-Bit erweiterte Real-Format. Dabei können alle binären Integerzahlen exakt im erweiterten Real-Format repräsentiert werden.

Die drei Integer-Formate und ihre dezimalen Zahlenbereiche sind in den folgenden Bildern zu sehen:

Binäre Integer-Formate:

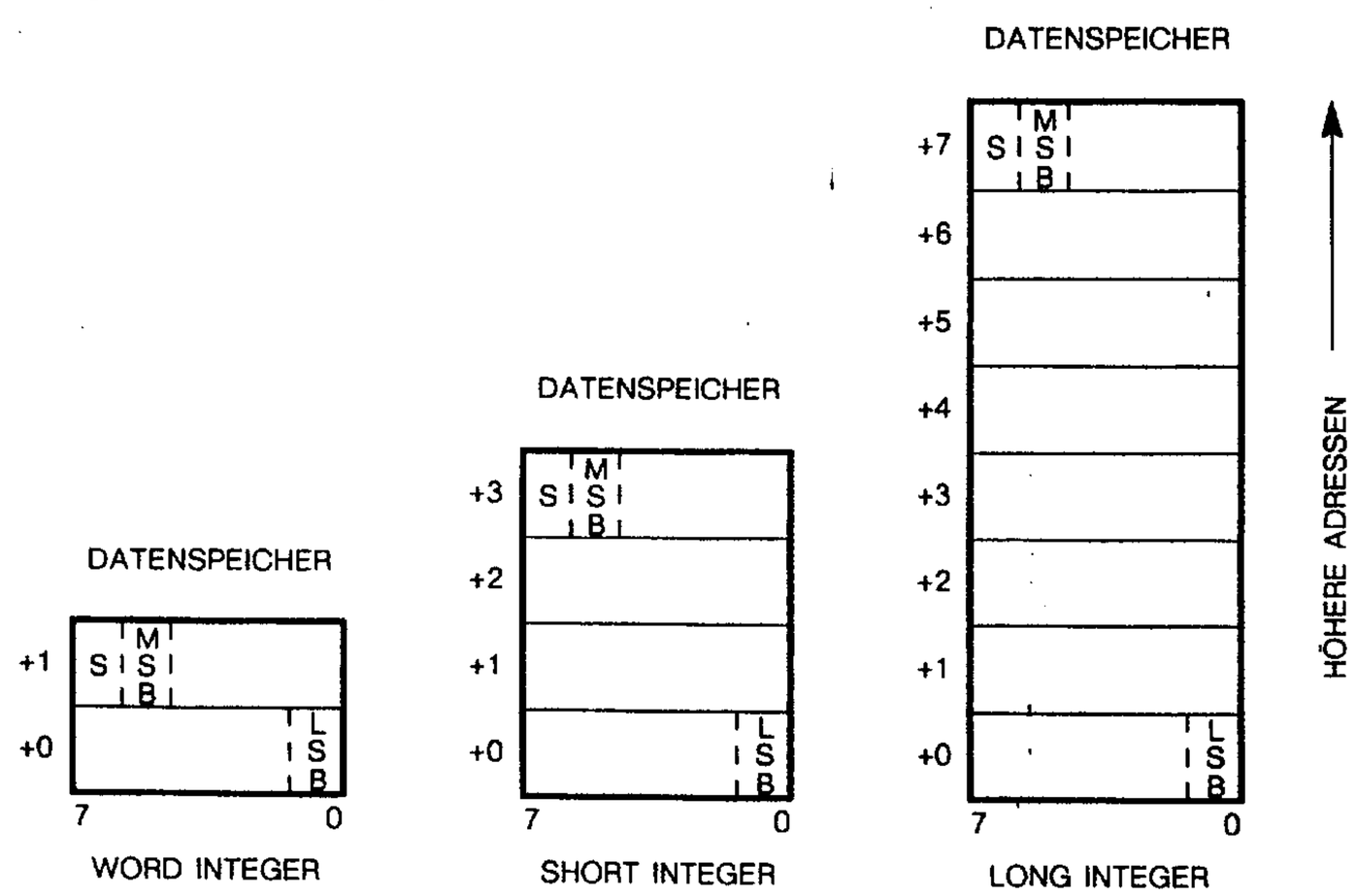


Mit Bit 15 (Sign) bei WORD-Integer, Bit 31 (Sign) bei SHORT-Integer und Bit 63 (Sign) bei LONG-Integer wird das Vorzeichen der Zahl bestimmt: **0 = positiv** und **1 = negativ.** Dabei werden negative Integerzahlen im Zweierkomplement dargestellt.

Dezimale Zahlenbereiche:

Datentyp	Bits	Dezimaler Zahlenbereich
WORD-Integer	16	$-32768 <= x <= +32767$
SHORT-Integer	32	$-2*10^9 <= x <= +2*10^9$
LONG-Integer	64	$-9*10^{18} <= x <= +9*10^{18}$

Die Speicher-Layouts der Integer-Operanden zeigen die folgenden Bilder. Wie zu sehen ist, liegt das Sign-Bit immer im höchstwertigen Adreßbyte und das LSB (Least Significant Bit) im niederwertigsten Adreßbyte:



S: Sign-Bit
MSB: Most Significant Bit
LSB: Least Significant Bit

8.2.1.1 Binär Integer Indefinite

In den drei Integer-Formaten gibt es jeweils ein Bitmuster, das entweder den Spezialwert **Integer Indefinite** (unbestimmt) oder die **größte negative** Integerzahl des betreffenden Integer-Formats repräsentiert, nämlich -2^{15} (WORD-Integer) oder -2^{31} (SHORT-Integer) oder -2^{63} (LONG-Integer). In der folgenden Darstellung des binären Integer-Zahlenbereichs ist dieses Bitmuster zu sehen:

Binärer Integer-Zahlenbereich:

	Klasse	Sign	Größe
Positiv	Größte	0	11 . . . 11
	.	.	.
	.	.	.
	Kleinste	0	00 . . . 01
	Null	0	00 . . . 00
Negativ	Kleinste	1	11 . . . 11
	.	.	.
	.	.	.
	Größte/Indefinite	**1**	**00 . . . 00**
		WORD:	<– 15 Bits –>
		SHORT:	<– 31 Bits –>
		LONG:	<– 63 Bits –>

Der 80486 liefert -2^{15}, -2^{31} oder -2^{63} (größte negative Zahl/Indefinite) in zwei Fällen:

1. Wenn der Wert (eventuell gerundet) in einem Numerik-Quellregister die größte negative Zahl repräsentiert oder
2. als Anwort auf eine **maskierte** ungültige Operation.

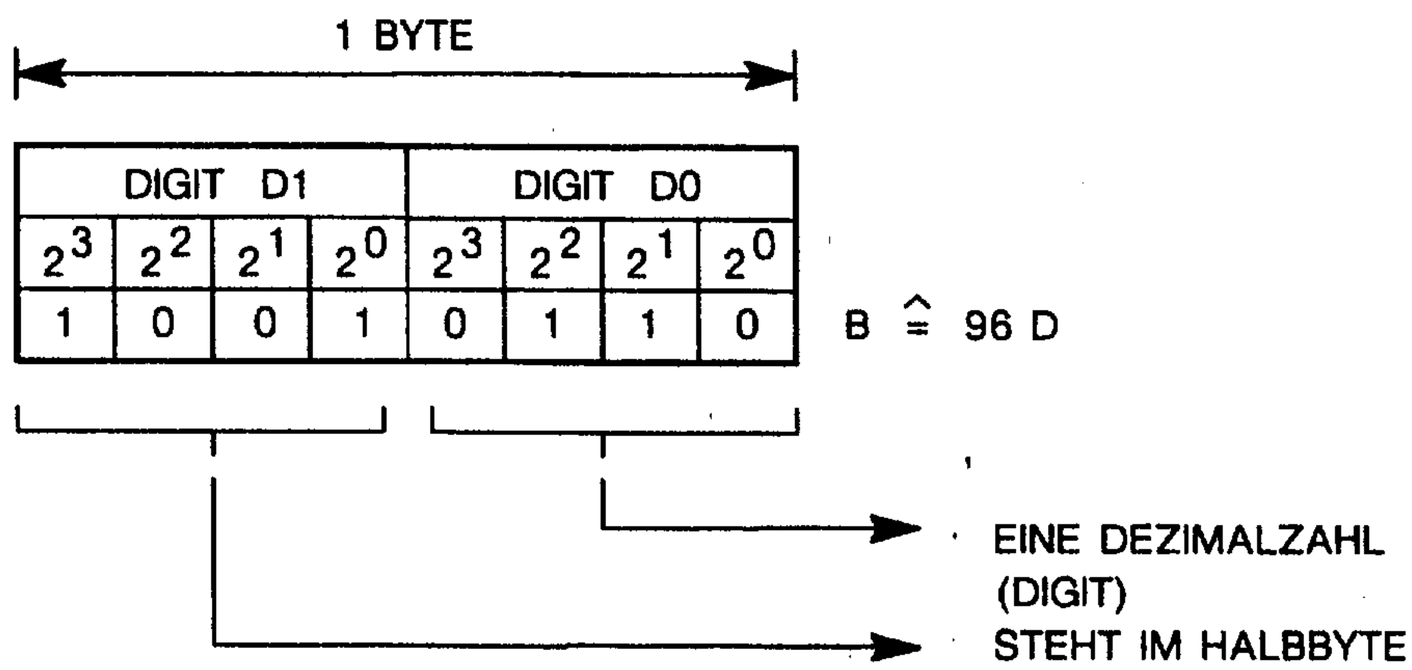
In Situationen, in denen unklar ist, ob das Operationsergebnis die größte negative Zahl darstellt oder binär Indefinite, kann zur Klärung das IE-Flag (Invalid Operation-Exception) im Statuswort überprüft werden.

Wird -2^{15} oder -2^{31} oder -2^{63} **geladen** oder in einer Integer-**Arithmetik** benutzt, dann interpretiert die FPU das betreffende Bitmuster als größte negative Zahl. Mit anderen Worten, **Indefinite** kann **nicht** als „normale" binäre Integer-Zahl geladen werden.

8.2.2 Packed Decimal Integer

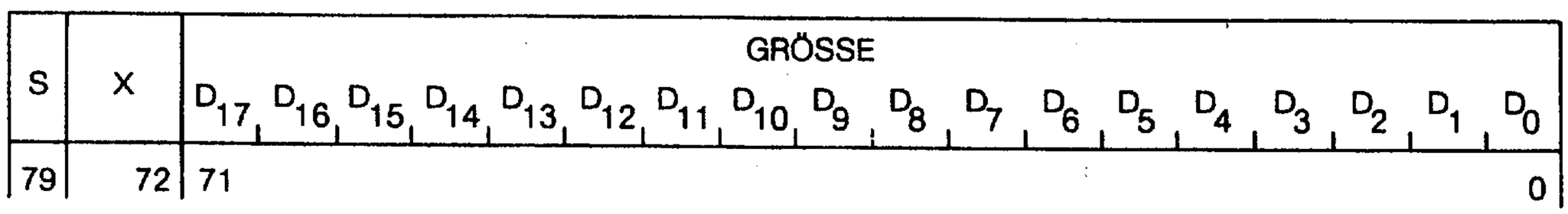
Dezimale Integer-Zahlen werden in **gepackter dezimaler** Form gespeichert, d.h. in jedes Byte werden **zwei** dezimale Stellen (Digits) im Zahlenbereich von 0..9 „gepackt".

Beispiel:



Das dezimale Integer-Format und ihr Zahlenbereich ist in den folgenden Bildern zu sehen:

PACKED DECIMAL INTEGER-FORMAT:

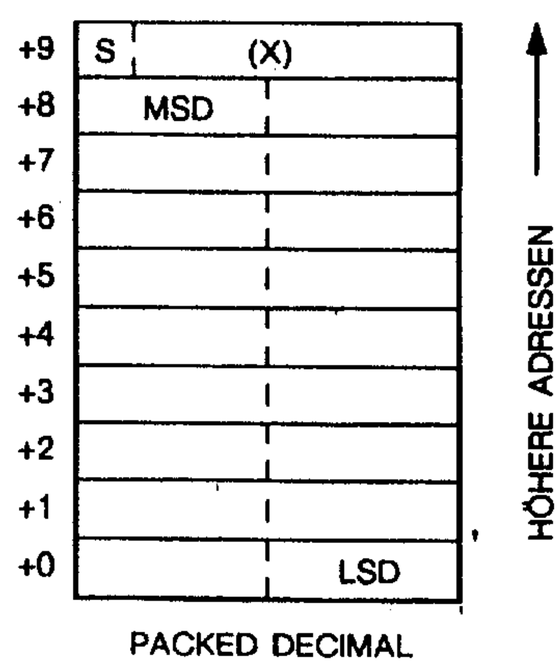


Die Genauigkeit dieses Datentyps liegt bei **18** Digits ($d_{17}..d_0$). Die höherwertigen Bits 79..72 haben mit Ausnahme von Bit 79 **keine** Bedeutung. Dieses Bit bestimmt das Vorzeichen der Zahl: **0** = **positiv** und **1** = **negativ**. Negative Zahlen werden hier **nicht** im Zweierkomplement dargestellt. Sie unterscheiden sich von positiven Zahlen nur durch das Vorzeichen.

Dezimaler Zahlenbereich:

Datentyp	Bits	Dezimaler Zahlenbereich
Packed Decimal Integer	80	–99 . . . 99 <= x <= +99 . . . 99

Das Speicher-Layout eines Packed Decimal Integer-Operanden zeigt das folgende Bild. Wie zu sehen ist, liegt das Sign-Bit im höchstwertigen Adreßbyte und das LSD (Least Significant Digit) im niederwertigsten Adreßbyte:



S: Sign-Bit
MSD: Most Significant Digit
LSD: Least Significant Digit
(X): Ohne Bedeutung

Genauso wie binäre Integerzahlen, existieren auch dezimale Integerzahlen nur im Speicher. Wenn sie von der FPU benutzt werden, dann konvertiert sie die FPU automatisch in das interne 80-Bit erweiterte Real-Format. Dabei können alle dezimalen Integerzahlen exakt im erweiterten Real-Format dargestellt werden.

8.2.2.1 Packed Decimal Integer Indefinite

In der folgenden Darstellung des dezimalen Integer-Zahlenbereichs ist ein Bitmuster zu erkennen, das den Spezialwert Packed Decimal Integer Indefinite repräsentiert:

KLASSE		SIGN		GRÖSSE					
				DIGIT	DIGIT	DIGIT	DIGIT		DIGIT
POSITIV	GRÖSSTE	0	0000000	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	. . .	1 0 0 1
	⋮	⋮	⋮				⋮		
	KLEINSTE	0	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 1
	NULL	0	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 0
NEGATIV	NULL	1	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 0
	KLEINSTE	1	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 1
	⋮	⋮	⋮				⋮		
	GRÖSSTE	1	0000000	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	. . .	1 0 0 1
	INDEFINITE	1	1111111	1 1 1 1	1 1 1 1	U U U U	U U U U	. . .	U U U U
		← 1 BYTE →		← 9 BYTES →					

UUUU bedeutet, daß die betreffenden Bits **undefiniert** sind.

Der 80486 liefert die Spezialzahl Indefinite als Antwort auf eine **maskierte** ungültige Operation. Sie kann mit dem Befehl FBSTP (Speichere Packed Decimal und Pop Stack) im Speicher hinterlegt werden.

Der Versuch, diese Zahl mit dem Befehl FBLD (Lade Packed Decimal in Numerik-Stack-Top) zu laden, führt zu einem undefinierten Ergebnis. Mit anderen Worten, **Indefinite** kann **nicht** als „normale" dezimale Integer-Zahl geladen werden.

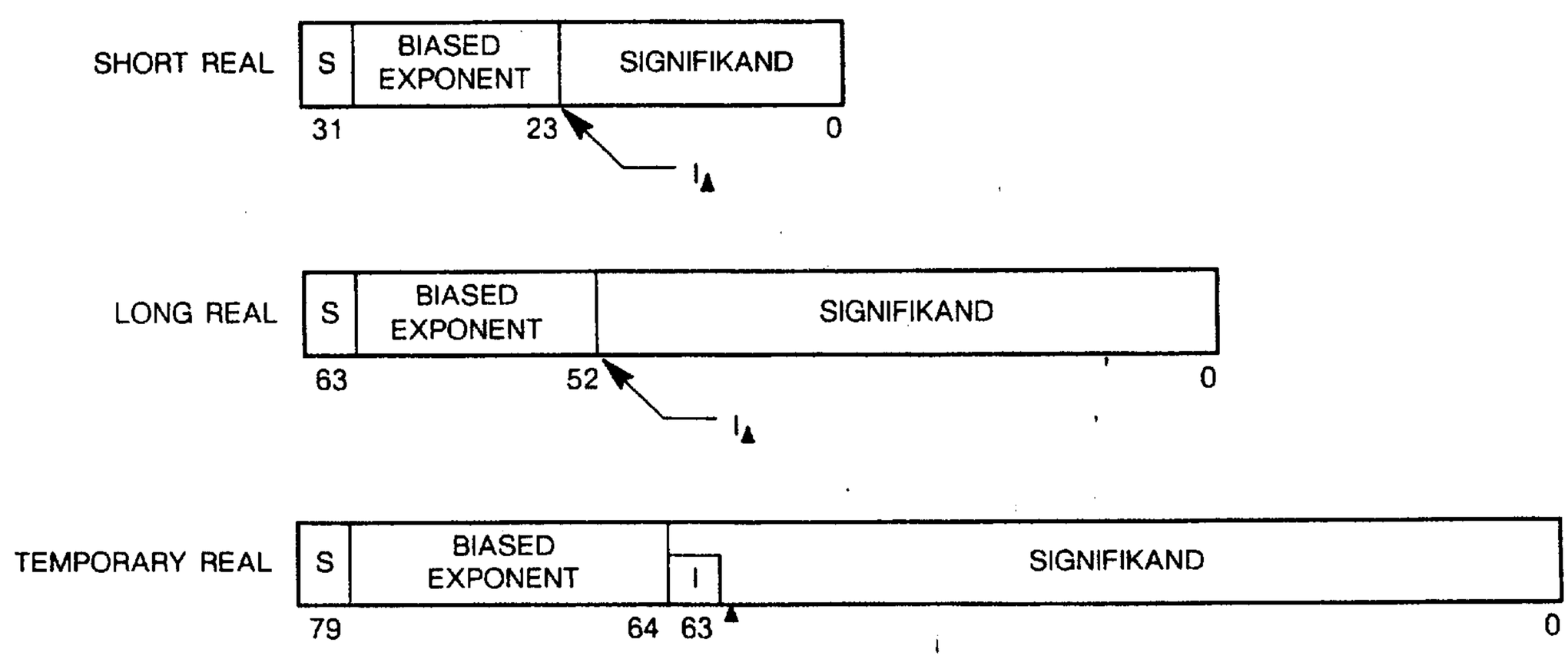
8.2.3 Real-Zahlen

Der 80486 speichert die Real-Zahlen (Gleitpunktzahlen) in drei verschiedenen Formaten, nämlich im SHORT-Real-Format (einfache Genauigkeit), im LONG-Real-Format (doppelte Genauigkeit) und im Temporary-Real-Format (erweiterte Genauigkeit). Jedes dieser Formate besteht aus drei Feldern und zwar aus

- dem **Signifikanden**-Feld,
- dem **Biased Exponent**-Feld und
- dem **Sign**-Feld

In den folgenden Bildern sind die drei Realzahl-Formate illustriert:

Realzahl-Formate:



Die Real-Zahlen des 80486 lassen sich allgemein folgendermaßen darstellen:

$$(-1)^S \cdot 2^E \cdot (f_0 \blacktriangle f_1 f_2 f_3 \ldots f_{P-1})$$

Hier ist

- S (Sign) = 0 (positiv) oder 1 (negativ)
- E (wahrer Exponent) = irgendeine Integerzahl zwischen E_{min} und E_{max}
- f_i (Signifikand) = 0 oder 1
- P (Präzision) = Anzahl der Genauigkeitsbits

Alle Real-Zahl-Formate können nun eindeutig durch ihre sogenannten **Format-Parameter** spezifiziert werden. Diese Parameter sind in der folgenden Tabelle aufgelistet:

Liste der Format-Parameter:

Parameter	Format		
	Short	Long	Temporary
Format-Breite in Bits	32	64	80
P (Anzahl der Genauigkeitsbits)	24	53	64
Exponenten-Breite in Bits	8	11	15
E_{max}	+127	+1023	+16383
E_{min}	–126	–1022	–16382
Bias	+127	+1023	+16383

Die **Genauigkeit** einer Real-Zahl ist durch die signifikanten Bits $f_0 \blacktriangle f_1 f_2 f_3 \ldots f_{P-1}$ bestimmt. Diese Bits werden im **Signifikanden**-Feld aufbewahrt. Dabei ist der Term „Signifikand" analog zum Ausdruck „Mantisse".

Die **Größe** einer Real-Zahl ist durch das **Exponenten**-Feld bestimmt. Dabei legt dieses Feld die Position des Gleitpunkts innerhalb der signifikanten Bits fest. Der Term „Exponent" ist analog zum Ausdruck „Charakteristik". Um den Vergleich von Real-Zahlen zu vereinfachen (z.B. beim Sortieren) speichert der 80486 den Exponeten in seiner sogenannten **biased** Form e. Dies bedeutet, daß der Prozessor zum **wahren** Exponenten E eine Konstante (Bias) addiert. Der zu speichernde (biased) Exponent hat daher die Form **e = E + Bias.** Abhängig vom Format der Real-Zahl hat die Bias unterschiediche Werte, nämlich +127 bei SHORT-Real, +1023 bei LONG-Real und +16383 bei Temporary- Real. Diese Werte wurden so gewählt, daß der zu speichernde biased Exponent immer **positiv** ist. Dadurch wird erreicht, daß Real-Zahlen mit gleichem Format und Vorzeichen wie **vorzeichenlose** Integer-Zahlen verglichen werden können.

Beispiele:

1, Der wahre Exponent E im SHORT-Realformat sei E_{min}, nämlich –126. Der biased Exponent ist dann e = E+Bias, also e = –126 + 127 = 1 (01H).
2, Der wahre Exponent E im SHORT-Realformat sei E_{max}, nämlich +127. Der biased Exponent ist dann e = E+Bias, also e = +127 + 127 = 254 (FEH)

Das **Vorzeichen** der Real-Zahl bestimmt das **Sign**-Feld: 0 = positiv, 1 = negativ.

Im folgenden Beispiel soll nun gezeigt werden, wie die in üblicher Weise notierte Dezimalzahl 175.125 in das 80486-SHORT-Realformat konvertiert wird. Das angegebene Verfahren benutzen so ähnlich auch Sprachübersetzer wie ASM386/486 oder iC386/486, wenn sie auf eine programm-definierte Realzahl treffen.

1. Schritt: Konvertierung von üblich dezimal nach üblich binär

ÜBLICH DEZIMAL ÜBLICH BINÄR

175.125 D -->

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	1	0	0	1	0	0	0	1	0

B

GLEITPUNKT

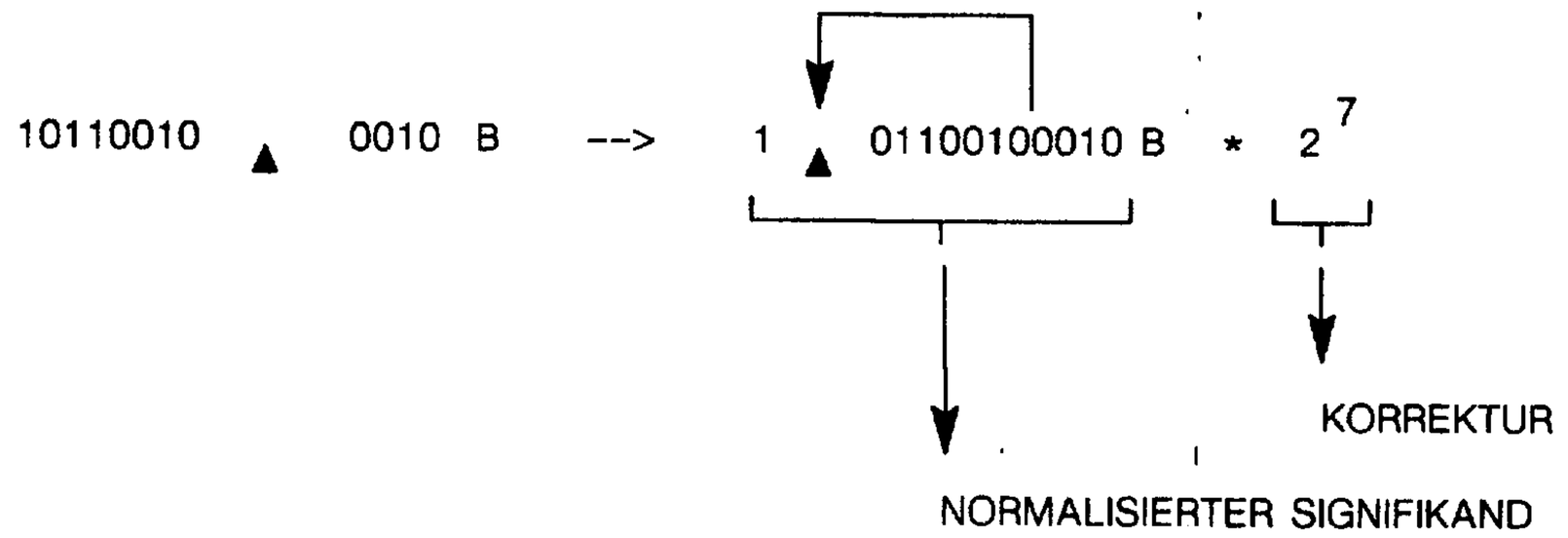
2. Schritt: Konvertierung von üblich binär nach wissenschaftlich binär

Bei diesem Schritt wird die Binärzahl in die sogenannte **normalisierte** Form gebracht. Dabei wird der Gleitpunkt solange nach links oder rechts verschoben, bis links neben dem Gleitpunkt eine einzelne **1** (Integer-Bit) erscheint. Das Ergebnis ist der künftige Signifikand, der aus dem **Integer**-Bit und den sogenannten **Fraktion**-Bits **f** besteht. Ein normalizierter Signifikand hat die folgende allgemeine Form:

1 ▲ fff..ff

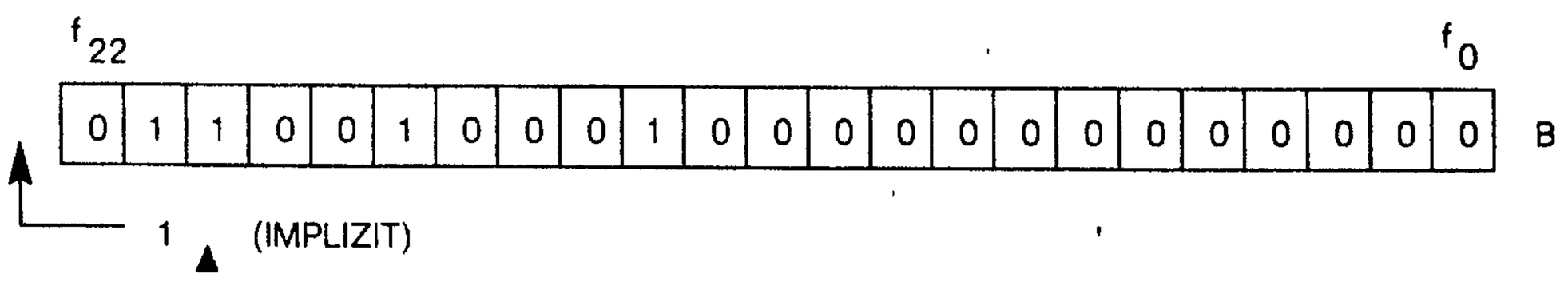
Dabei kennzeichnet ▲ den Gleitpunkt. Die Anzahl der Fraktion-Bits variiert abhängig vom Realzahl-Formt. Es sind 23 Bits bei SHORT-Reals, 52 Bits bei LONG-Reals und

63 Bits bei Temporary-Reals. Im SHORT- und LONG-Realformat ist das Integer-Bit nur implizit vorhanden und **nicht** tatsächlich gespeichert. Auf diese Weise erhöht sich die Anzahl der Bits im Signifikanden. Nur im Temporary-Realformat ist auch das Integer-Bit physikalisch präsent.



Hier wird beim Normalisierungs-Prozeß der Gleitpunkt um 7 Bitpositionen nach links verschoben. Dies bedeutet, die Binärzahl wird durch 2^7 **dividiert**. Damit aber der ursprüngliche Wert der Binärzahl nicht verloren geht, muß der normalisierte Signifikand mit 2^7 **multipliziert** werden.

Der aus dem Normalisierungs-Prozeß resultierende Signifikand wird folgendermaßen im Speicher aufbewahrt:



Im SHORT-Realformat ist die Anzahl der Genauigkeitsbits **P=24**. Weil aber hier das Integer-Bit **nicht** gespeichert wird, stehen dem Signifikanden noch 23 Bits zur Verfügung ($f_{22}..f_0$).

3. Schritt: Erzeugung des zu speichernden (biased) Exponenten e
   Der **wahre** Exponent E=7 im Korrekturfaktor 2^7 wird jetzt mit e = E+Bias in die **biased** Form gebracht:

   - $\underbrace{0110010001}_{\text{Signifikand}}...B * 2^{7}$ ← wahrer Exponent E
   - e = E + Bias; e = 7 + 127 = **134** D
     (7 ← wahrer Exponent E; 127 ← Bias des SHORT-Real-Formats)

Damit hat der biased Exponent folgendes Binärformat:

DEZIMAL		BINÄR								
		e7							e0	
		2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
e = 134 D	--->	1	0	0	0	0	1	1	0	B

4. Schritt: Bestimmung des Vorzeichens
   Weil die Original-Dezimalzahl 175.125 **positiv** ist, bekommt S (Sign) den Wert **0.**

Die folgenden Bilder zeigen zusammenfassend das vollständige Konvertierungsergebnis und das resultierende Speicher-Layout:

KONVERTIERUNGSERGEBNIS:

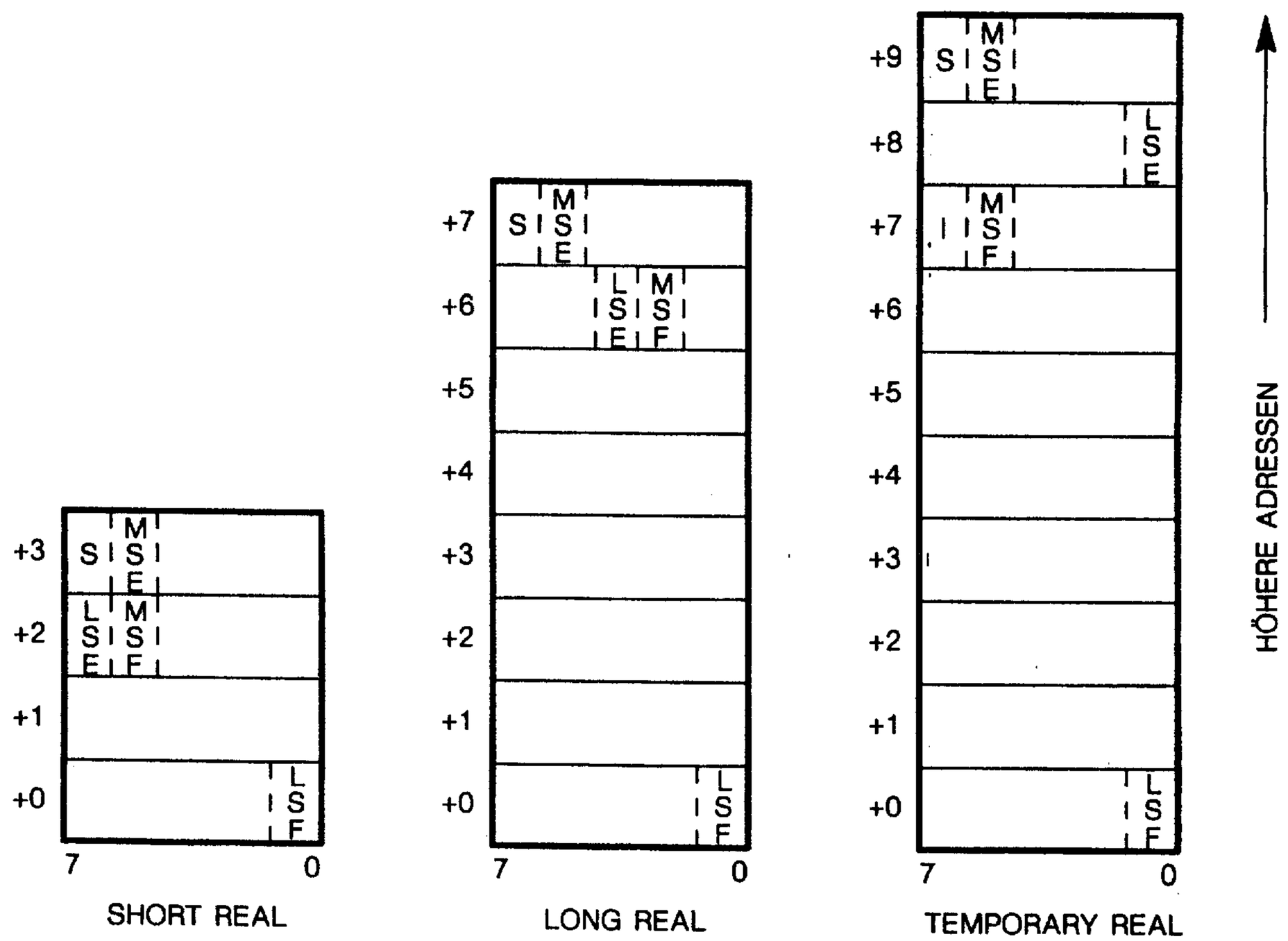
BEMERKUNG	WERT		
ÜBLICH DEZIMAL	178.125		
SHORT-REAL-FORMAT (NORMALISIERT)	SIGN	BIASED EXPONENT	SIGNIFIKAND
	0	10000110	01100100010000000000000 ← 1 △ (IMPLIZIT)

SPEICHER-LAYOUT DES SHORT-REAL-FORMATS:

S	BIASED EXPONENT								SIGNIFIKAND																								
	E^7	.	.	.	.	.	.	E^0	F^{22}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	F^0	
0	1	0	0	0	0	1	1	0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	B
4			3					3				2				2				0				0				0					H

← HÖHERE ADRESSEN

SHORT-Realzahlen und LONG-Realzahlen existieren nur im Speicher. Wenn eine dieser Zahlen in ein FPU-Register geladen wird, dann konvertiert sie die FPU automatisch in das interne 80-Bit erweiterte Real-Format. Umgekehrt können die 80-Bit-Registerdaten in allen drei Formaten gespeichert werden. Dabei liegt das Sign-Bit immer im höchstwertigen Adreßbyte und das LSF (Least Significant Fraction Bit) im niederwertigsten Adreßbyte. In den folgenden Bildern sind die Speicher-Layouts der Real-Operanden illustriert:



S: Sign
MSE: Most Significant Exponent Bit
LSE: Least Significant Exponent Bit
MSF: Most Significant Fraction Bit
LSF: Least Significant Fraction Bit
I: Integer-Bit des Signifikanden (**nur** im Temporary Real-Format)

8.3 Spezielle Numerikzahlen

Neben den typischen Real- und Integerwerten, können die numerischen Datenformate auch einige Spezialwerte aufnehmen. Solche Werte sind bei gewissen Berechnungssituationen und Operationen Träger wichtiger Informationen. Die verschiedenen Typen der Spezialwerte sind

- Denormal-Realzahlen,
- Nullen,
- Positiv und negativ Unendlich,
- NaNs (Not-a-Number),
- Indefinite (unbestimmt)
- nicht-unterstützte Bitmuster

8.3.1 Denormal-Realzahlen

Der 80486 speichert Nicht-Null-Zahlen in der **normalisierten** Floating-Point-Form, d.h. das Integer-Bit des Signifikanden ist immer 1 (im SHORT- und LONG-Realformat **nicht** gespeichert, im TEMPORARY-Realformat gespeichert). Wenn aber eine Realzahl einen Wert hat, der sehr nahe bei Null liegt, dann ist eine solche „winzige" Zahl in der normalisierten Form **nicht** exakt darstellbar. Eine Realzahl **R** gilt als „winzig", wenn sie folgende Bedingungen erfüllt:

- $-2^{Emin} < R < 0$ oder
- $0 < R < +2^{Emin}$

Zur Erinnerung: Emin hat, abhängig vom Realformat entweder, den Wert –126 (SHORT-Realformat), oder –1022 (LONG-Realformat) oder –16382 (Temporary-Realformat).

Denormal-Operanden werden durch ihre Exponenten identifiziert. In der biased Form ist dies immer der Bitstring **00...00**. Den **gleichen** Exponenten haben auch die beiden Nullen +0 und –0. Denormal-Operanden unterscheiden sich von den Nullen nur durch ihre Signifikanden:

- Denormals haben **Nicht-Null**-Signifikanden,
- Nullen haben **Null**-Signifikanden

Denormal-Operanden erfahren vom 80486 eine besondere Behandlung:

- Soweit es möglich ist, versucht der Prozessor keinerlei Denormals zu erzeugen. Wenn aber eine Denormal-Realzahl im SHORT- oder LONG-Realformat geladen wird und das **DM**-Bit im Kontrollwort ist gesetzt (maskierte Denormalized Operand Exception), dann **normalisiert** die FPU automatisch diese Zahl und konvertiert sie in das interne 80-Bit erweiterte Real-Format.
- Wenn nach einer Floating-Point-Operation ein **Underflow** auftritt, weil der wahre Exponent E des Ergebnisses zu klein ist (E < Emin) um im gewünschten Zielformat dargestellt werden zu können, dann leitet der Prozessor den **Denormalisierungsprozeß** ein und liefert ein gleichwertiges Denormal (siehe 8.1.2). In diesem Fall wird das DE-Bit im Statuswort gesetzt (DE=1). Der Programmierer ist dadurch in der Lage, Denormals zu entdecken, bevor sie in nachfolgenden Berechnungen benutzt werden.

8.3.2 NaN (Not-a-Number)

NaNs sind ebenfalls Mitglieder der Klasse von Spezialwerten und existieren nur im Real-Format. Sie können jedes Vorzeichen haben (Sign = 0 oder 1) und sie werden durch ihre Exponenten identifiziert. In der biased Form ist dies immer der Bitstring **11..11** binär. Die Signifikanden der NaNs können jeden beliebigen Bitstring enthalten, außer **.10..00** binär. Diese Realzahl ist ein **spezielles** NaN und wird als **Indefinite** (unbestimmt) bezeichnet. Die FPU generiert Indefinities als Antwort auf **ungültige** Operationen. Alle NaNs repräsentieren Werte, die vom Programmierer z.B. als Ersatz- oder Diagnoseergebnisse codiert werden können.

Die FPU unterscheidet zwei Klassen von NaNs: Die sogenannten **Signaling** NaNs (SNaNs) und die **Quiet** NaNs (QNaNs).

Die folgenden Tabellen zeigen die Bitmuster dieser NaNs in den drei Formaten SHORT-Real, LONG-Real und TEMPORARY-Real:

NaNs im SHORT- und LONG-Realformat

		Klasse	Sign	Biased Exponent	Signifikand * ▲ ff . . . ff
Positiv	NaNs	Quiet	0	11 . . . 11	**1**1 . . . 11
			.	.	.
			.	.	.
			0	11 . . . 11	**1**0 . . . 00
		Signaling	0	11 . . . 11	**0**1 . . . 11
			.	.	.
			.	.	.
			0	11 . . . 11	**0**0 . . . 01
Negativ	NaNs	Signaling	1	11 . . . 11	**0**0 . . . 01
			.	.	.
			.	.	.
			1	11 . . . 11	**0**1 . . . 11
		Quiet **Indefinite**	1	11 . . . 11	**1**0 . . . 00
			.	.	.
			.	.	.
			1	11 . . . 11	**1**1 . . . 11
			SHORT:	<— 8 Bits —>	<— 23 Bits —>
			LONG:	<— 11 Bits —>	<— 52 Bits —>

* Das Integer-Bit des Signifikanden ist nur implizit vorhanden, also **nicht** gespeichert

NaNs im Temporary-Realformat

		Klasse	Sign	Biased Exponent	Signifikand * 1 ▲ ff . . . ff
Positiv	NaNs	Quiet	0	11 . . . 11	1 **1**1 . . . 11
			.	.	.
			.	.	.
			0	11 . . . 11	1 **1**0 . . . 00
		Signaling	0	11 . . . 11	1 **0**1 . . . 11
			.	.	.
			.	.	.
			0	11 . . . 11	1 **0**0 . . . 01
Negativ	NaNs	Signaling	1	11 . . . 11	1 **0**0 . . . 01
			.	.	.
			.	.	.
			1	11 . . . 11	1 **0**1 . . . 11
		Quiet **Indefinite**	1	11 . . . 11	1 **1**0 . . . 00
			.	.	.
			.	.	.
			1	11 . . . 11	1 **1**1 . . . 11
			TEMPORARY:	<— 15 Bits —>	<— 64 Bits —>

* Das Integer-Bit des Signifikanden ist im Temporary-Realformat gespeichert

8.3.2.1 Signaling NaNs

Ein Signaling NaN ist ein NaN, bei dem das **MSB** des Signifikanden **gelöscht** ist (MSB=0). Die restlichen Bits des Signifikanden können beliebige Werte haben. Die FPU liefert als Ergebnis niemals ein SNaN. Sie kann aber SNaNs als Operanden erkennen. Daher sind Ladeoperationen und arithmetische Befehle wie FXCH, FCHS und FABS mit SNaNs erlaubt. Alle anderen arithmetischen Operationen mit SNaNs führen zur Exception ‚Ungültige Operation'. Wenn diese Exception **unmaskiert** ist, dann kann der Programmierer SNaNs benützen, um den Exception-Handler zu aktivieren. So ist es z.B. denkbar, daß ein System-Programmierer die Real-Elemente eines Vektors mit SNaNs belegt, um sie als nicht-initialisiert zu kennzeichnen. Dabei können die Signifikanden der individuellen SNaNs die relativen Positionen (Indizes) der Vektorelemente spezifizieren. Wenn nun ein Anwender-Programm versucht, auf ein solches Element zuzugreifen, dann trifft es auf ein SNaN. Ist diese ungültige Operation **unmaskiert**, dann wird der Exception-Handler aufgerufen. Dieser kann nun feststellen, auf welches Element absichtlich oder unabsichtlich zugegriffen worden ist. Hierfür steht ihm

- der **Operanden-Pointer** im ‚Environment' zur Verfügung, der auf das SNaN zeigt und
- der **Signifikand** dieses NaNs, der den Index des Vektorelements enthält.

8.3.2.2 Quiet NaNs

Ein Quiet NaN ist ein NaN, bei dem das **MSB** des Signifikanden **gesetzt** ist (MSB=1). Zu den QNaNs gehört auch der Wert **Indefinite**, der bei bestimmten Ausnahme-Bedingungen als Ersatzergebnis geliefert wird. Die FPU kann die anderen QNaNs aus den SNaNs ableiten. Dies geschieht einfach dadurch, daß sie das MSB im Signifikanden eines SNaNs (das MSB ist hier 0) auf **1** setzt. Weil die restlichen Bits des Signifikanden unverändert bleiben, pflanzen sich die Diagnose-Informationen, die durch diese Bits repräsentiert werden, von den SNaNs in die QNaNs fort. Ein QNaN wird von der FPU in zwei Fällen generiert:

- als maskierte Antwort einer ungültigen Operation und
- als Ergebnis einer Operation, bei der mindestens ein Operand ein QNaN ist.

Welche Regeln die FPU dabei befolgt, zeigt die folgende Tabelle:

Operation	Aktion
Real-Operation mit einem SNaN und einem QNaN	Als Ergebnis wird der QNaN-Operand geliefert
Real-Operation mit zwei SNaNs	Das SNaN mit dem größeren Signifikanden wird in ein QNaN konvertiert und dieses als Ergebnis geliefert
Real-Operation mit zwei QNaNs	Als Ergebnis wird das QNaN mit dem größeren Signifikanden geliefert
Real-Operation mit einem SNaN und einer anderen Zahl	Das SNaN wird in ein QNaN konvertiert und dieses als Ergebnis geliefert
Real-Operation mit einem QNaN und einer anderen Zahl	Als Ergebnis wird der QNaN-Operand geliefert
Ungültige Operation, bei der kein NaN beteiligt ist	Als Ergebnis wird das QNaN Indefinite geliefert

QNaNs können z.B. benützt werden, um die Fehlersuche beim Programmtest zu beschleunigen. Dabei kann der Exception-Handler helfen, der jedesmal, wenn er aufgerufen wird, eine bestimmte Diagnose-Information im Speicher hinterlegt. Wenn diese Information gespeichert ist, dann kann der Handler für den fehlerhaften Befehl als Ersatzergebnis ein QNaN bereitstellen. Dieses QNaN wirkt dann wie ein Etikett, das den mit ihm verbundenen Diagnoseraum kennzeichnet.

Bei Fortsetzung des Programms kann dann der Handler für jeden weiteren fehlerhaften Befehl ein anderes QNaN erzeugen. Wenn das Programm beendet ist, kann über die resultierenden NaNs auf alle Diagnose-Daten zugegriffen werden. Auf diese Weise ist es möglich, in einem einzigen Testlauf viele Fehler zu diagnostizieren und zu beseitigen.

Wie bereits oben erwähnt, kommt es immer dann zur Exception **Ungültige Operation**, wenn die FPU auf **SNaN**s trifft. Diese Exception wird aber auch dann generiert, wenn **QNaN**s in den Befehlen **FCOM** (Vergleiche Realzahlen), **FIST** (Speichere Integerzahl) und **FBSTP** (Speichere Packed Decimal und Pop Stack) benutzt werden.

8.3.3 Infinity (Unendlich)

Alle Realformate unterstützen die vorzeichenbehaftete Darstellung von **Unendlich**, also +Unendlich und –Unendlich.

- Der biased Exponent dieser Spezialzahlen ist **11..11** binär und der
- Signifikand enthält den Bitstring **1▲00..00** binär.

Unterschieden werden diese beiden Spezialzahlen durch ihr Vorzeichen, Sign = 0 (+Unendlich), Sign = 1 (–Unendlich). Ihre Codes können auf zweierlei Weise erzeugt werden.

Entweder vom Programmierer selbst oder von der FPU als Antwort auf eine **Overflow**- oder **Zero Divide**-Exception. Weil die FPU nur das **affine** Modell der Zahlendarstellung unterstützt, liegen endliche Zahlen im Bereich von

- –Unendlich < (irgendeine endliche Zahl) < +Unendlich

8.3.4 Nicht-unterstützte Datenformate

Das **Temporary**-Realformat läßt viele Bitmuster zu, die zum großen Teil vom 8087/80287 unterstützt werden. Die meisten dieser Bitmuster unterstützen aber der 80387 und die FPU des 80486 nicht mehr. Dazu gehören die **Pseudo-NaNs**, die **Pseudoinfinities** (Pseudo-Unendlich) und die **Unnormals**. Sollte die FPU auf diese Operanden treffen, dann führt dies zur Exception **Ungültige Operation**. Es gibt außerdem Bitmuster, die als **Pseudodenormals** bekannt sind, diese Bitmuster werden von der FPU **nicht** generiert. Wenn aber die FPU auf ein solches Pseudodenormal trifft, behandelt sie es wie ein **Normal**. D. h. sie tut so, wie wenn der Exponent **00..01** wäre und beläßt den Signifikanden wie er ist. Es wird aber trotzdem die Exception **Denormalized Operand** generiert (DE=1).

Die folgende Tabelle zeigt im Überblick alle Datenformate, die von der FPU **nicht** unterstützt werden:

Nicht-unterstützte TEMPORARY-Real-Formate

KLASSE			SIGN	BIASED EXPONENT	SIGNIFIKAND* I△FF...FF
POSITIV	PSEUDO NaNs	QUIET	0 . 0	11 ... 11 . 11 ... 11	0 11...11 . 0 10...00
		SIGNALING	0 . 0	11 ... 11 . 11 ... 11	0 01...11 . 0 00...01
		PSEUDOINFINITY	0	11 ... 11	0 00...00
	REALS	UNNORMALS	0 . 0	11 ... 10 . 00 ... 01	0 11...11 . 0 00...00
		PSEUDODENORMALS	0 . 0	00 ... 00 . 00 ... 00	1 11...11 . 1 00...00
NEGATIV	REALS	PSEUDODENORMALS	1 . 1	00 ... 00 . 00 ... 00	1 11...11 . 1 00...00
		UNNORMALS	1 . 1	11 ... 10 . 00 ... 01	0 11...11 . 0 00...00
		PSEUDOINFINITY	1	11 ... 11	0 00...00
	PSEUDO NaNs	SIGNALING	1 . 1	11 ... 11 . 11 ... 11	0 01...11 . 0 00...01
		QUIET	1 . 1	11 ... 11 . 11 ... 11	0 11...11 . 0 10...00
				TEMPORARY: ← 15 BITS →	← 64 BITS →

* Das Integer-Bit ist gespeichert

8.4 Zahlenbereiche der unterstützten Datenformate

Das von der FPU unterstützte **Floating-Point**-Zahlensystem enthält nur **Normals**, **Denormals** und **NaN**s.

Die folgenden Tabellen zeigen ihre Bitmuster, ihre hexadezimalen Repräsentationen und ihre dezimalen Werte in allen drei Formaten.

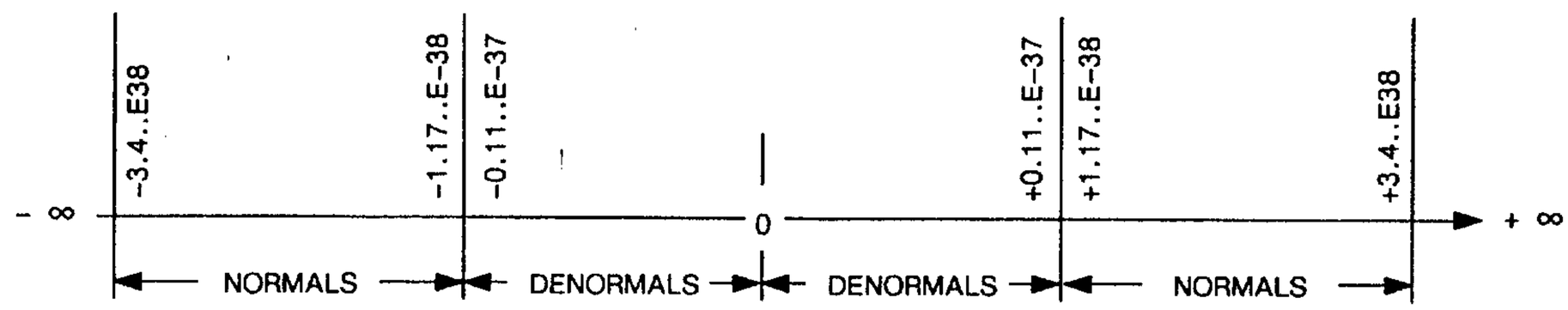
8.4.1 Zahlenbereich der SHORT-Reals:

TYP			SIGN	BIASED EXPONENT	SIGNIFIKAND* Δ FF...FF	HEXADEZIMALE REPRÄSENTATION	DEZIMALER WERT
POSITIV	NANs		0	11 ... 11	11...11	7F FF FF FF H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	11 ... 11	00...01	7F 80 00 01 H	–
	∞		0	11 ... 11	00...00	7F 80 00 00 H	–
	REALS	NORMALs	0	11 ... 10	11...11	7F 7F FF FF H	+3.40..E38
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	00 ... 01	00...00	00 80 00 00 H	+1.17..E-38
		DENORMALs	0	00 ... 00	11...11	00 7F FF FF H	+0.11..E-37
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	00 ... 00	00...01	00 00 00 01 H	+0.00..E-37
		ZERO	0	00 ... 00	00...00	00 00 00 00 H	+0.00..E0
NEGATIV	REALS	ZERO	1	00 ... 00	00...00	80 00 00 00 H	-0.00..E0
		DENORMALs	1	00 ... 00	00...01	80 00 00 01 H	-0.00..E-37
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	00 ... 00	11...11	80 7F FF FF H	-0.11..E-37
		NORMALs	1	00 ... 01	00...00	80 80 00 00 H	-1.17..E-38
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	11 ... 10	11...11	FF 7F FF FF H	-3.40..E38
	∞		0	11 ... 11	00...00	FF 80 00 00 H	–
	NANs		1	11 ... 11	00...01	FF 80 00 01 H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
		INDEFINITE	1	11 ... 11	10...00	FF 00 00 00 H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	11 ... 11	11...11	FF FF FF FF H	

SHORT: ← 8 BITS → ← 23 BITS → | HIGH ← LOW |

* Das Integer-Bit ist nur implizit vorhanden, also **nicht** gespeichert

Die Lage der SHORT-Normals und SHORT-Denormals auf dem Zahlenstrahl illustriert das folgende Bild:



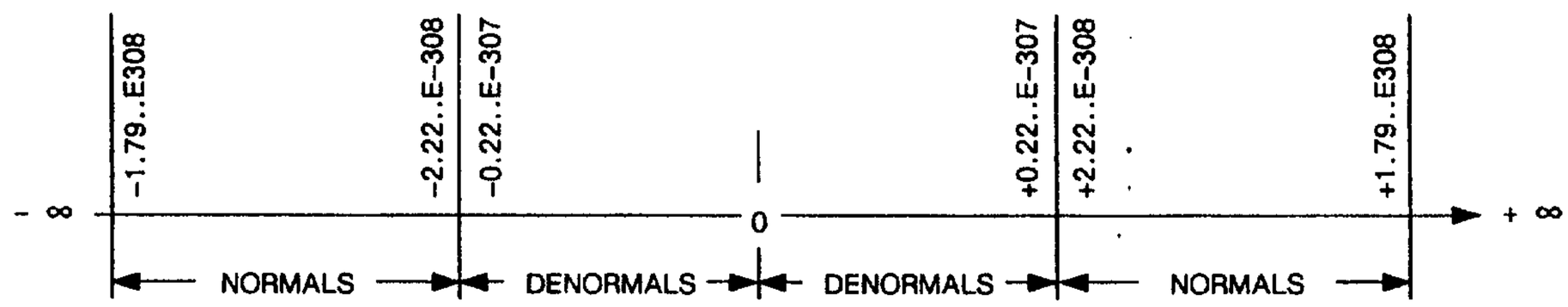
8.4.2 Zahlenbereich der LONG-Reals:

TYP			SIGN	BIASED EXPONENT	SIGNIFIKAND* Δ FF...FF	HEXADEZIMALE REPRÄSENTATION	DEZIMALER WERT
POSITIV		NANs	0	11 ... 11	11...11	7F FF FF FF FF FF FF FF H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	11 ... 11	00...01	7F 00 00 00 00 00 00 01 H	–
		∞	0	11 ... 11	00...00	7F F0 00 00 00 00 00 00 H	–
	REALS	NORMALs	0	11 ... 10	11...11	7F EF FF FF FF FF FF FF H	+1.79..E308
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	00 ... 01	00...00	00 10 00 00 00 00 00 00 H	+2.22..E-308
		DENORMALs	0	00 ... 00	11...11	00 0F FF FF FF FF FF FF H	+0.22..E-307
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	00 ... 00	00...01	00 00 00 00 00 00 00 01 H	+0.00..E-307
		ZERO	0	00 ... 00	00...00	00 00 00 00 00 00 00 00 H	+0.00..E0
NEGATIV	REALS	ZERO	1	00 ... 00	00...00	80 00 00 00 00 00 00 00 H	-0.00..E0
		DENORMALs	1	00 ... 00	00...01	80 00 00 00 00 00 00 01 H	-0.00..E-307
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	00 ... 00	11...11	80 0F FF FF FF FF FF FF H	-0.22..E-307
		NORMALs	1	00 ... 01	00...00	80 10 00 00 00 00 00 00 H	-2.22..E-308
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	11 ... 10	11...11	FF EF FF FF FF FF FF FF H	-1.79..E308
	∞		0	11 ... 11	00...00	FF F0 00 00 00 00 00 00 H	–
	NANs		1	11 ... 11	00...01	FF F0 00 00 00 00 00 00 H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
		INDEFINITE	1	11 ... 11	10...00	FF F8 00 00 00 00 00 00 H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	11 ... 11	11...11	FF FF FF FF FF FF FF FF H	

LONG: ← 11 BITS → ← 52 BITS → HIGH ← LOW

* Das Integer-Bit ist nur implizit vorhanden, also **nicht** gespeichert

Die Lage der LONG-Normals und LONG-Denormals auf dem Zahlenstrahl illustriert das folgende Bild:



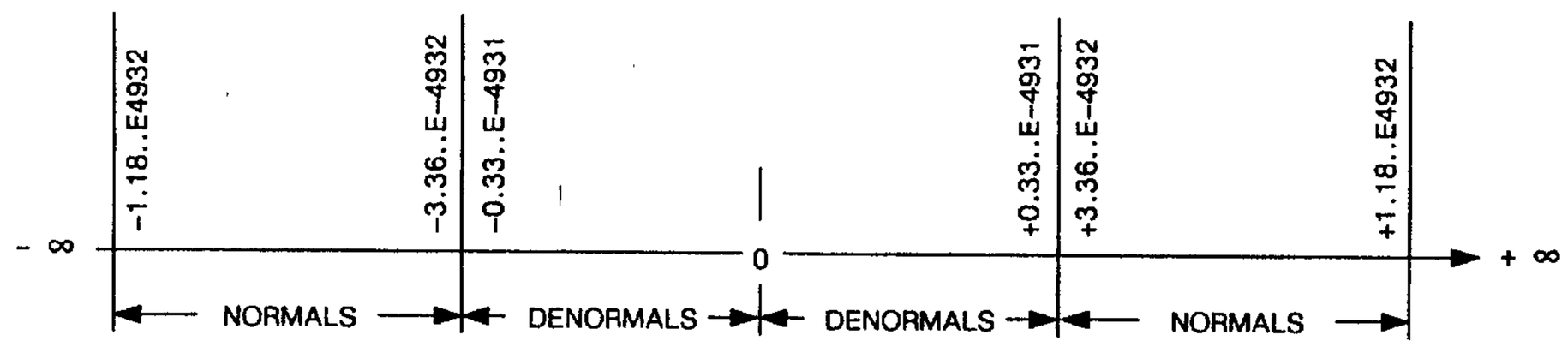
8.4.3 Zahlenbereich der TEMPORARY-Reals:

TYP			SIGN	BIASED EXPONENT	SIGNIFIKAND* I△ FF...FF	HEXADEZIMALE REPRÄSENTATION	DEZIMALER WERT
POSITIV		NANs	0	11 ... 11	1 11...11	7F FF FF FF FF FF FF FF FF FF H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	11 ... 11	1 00...01	7F FF 80 00 00 00 00 00 00 01 H	–
		∞	0	11 ... 11	1 00...00	7F FF 80 00 00 00 00 00 00 00 H	–
	REALS	NORMALs	0	11 ... 10	1 11...11	7F FE FF FF FF FF FF FF FF FF H	+1.18..E4932
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	00 ... 01	1 00...00	00 01 80 00 00 00 00 00 00 00 H	+3.36..E-4932
		DENORMALs	0	00 ... 00	0 11...11	00 00 7F FF FF FF FF FF FF FF H	+0.33..E-4931
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			0	00 ... 00	0 00...01	00 00 00 00 00 00 00 00 00 01 H	+0.00..E-4931
		ZERO	0	00 ... 00	0 00...00	00 00 00 00 00 00 00 00 00 00 H	+0.00..E0
NEGATIV	REALS	ZERO	1	00 ... 00	0 00...00	80 00 00 00 00 00 00 00 00 00 H	-0.00..E0
		DENORMALs	1	00 ... 00	0 00...01	80 00 00 00 00 00 00 00 00 01 H	-0.00..E-4931
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	00 ... 00	0 11...11	80 00 7F FF FF FF FF FF FF FF H	-0.33..E-4931
		NORMALs	1	00 ... 01	1 00...00	80 01 80 00 00 00 00 00 00 00 H	-3.36..E-4932
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	11 ... 10	1 11...11	FF FE FF FF FF FF FF FF FF FF H	-1.18..E4932
	∞		0	11 ... 11	1 00...00	FF FF 80 00 00 00 00 00 00 00 H	–
	NANs		1	11 ... 11	1 00...01	FF FF 80 00 00 00 00 00 00 01 H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
		INDEFINITE	1	11 ... 11	1 10...00	FF FF C0 00 00 00 00 00 00 00 H	–
			.	.	.	.	–
			.	.	.	.	–
			.	.	.	.	–
			1	11 ... 11	1 11...11	FF FF FF FF FF FF FF FF FF FF H	

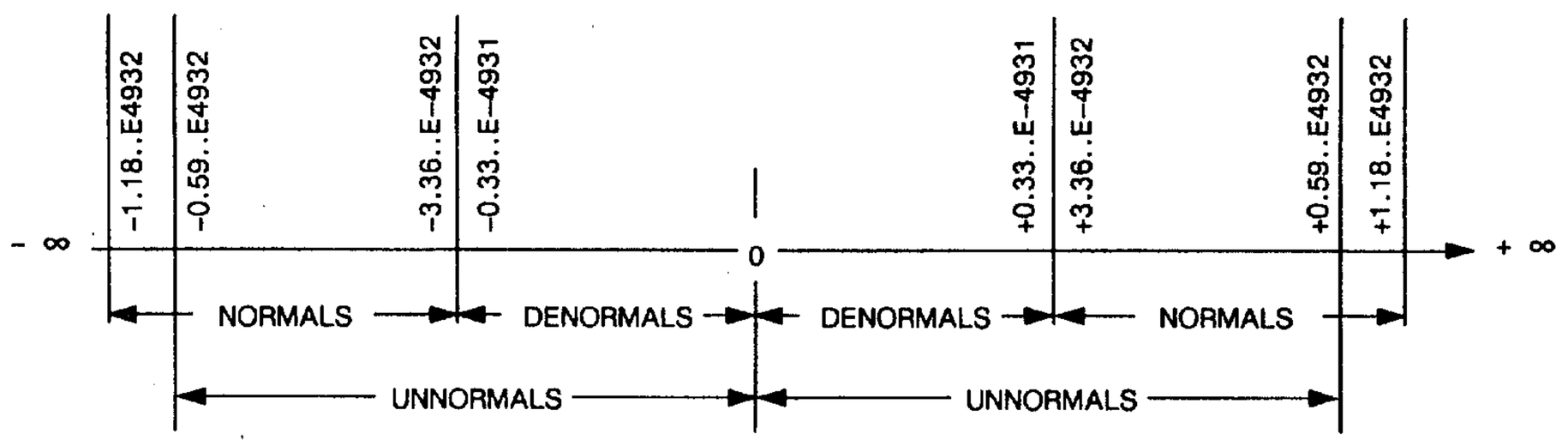
TEMPORARY: ← 15 BITS → ← 64 BITS → HIGH ← LOW

* Das Integer-Bit ist gespeichert

Die Lage der TEMPORARY-Normals und TEMPORARY-Denormals auf dem Zahlenstrahl illustriert das folgende Bild:



Im Gegensatz zu früheren Numerik-Prozessoren unterstützen der 80387 und die 80486-FPU die Unnormals **nicht** mehr. Warum sie überflüssig sind, illustriert das folgende Bild:



Wie zu erkennen ist, decken die Denormals, die Normals und die beiden Nullen zusammen einen Zahlenbereich ab, der zwischen

–1.18..E4932 und +1.18..E4932

liegt. Dagegen können Unnormals zusammen mit den Nullen nur die Hälfte dieses Zahlenbereichs zwischen

–0.59...E4932 und +0.59...E4932

darstellen. Der Unterschied zwischen Normals und Unnormals ist dadurch gekennzeichnet, daß

- bei den Normals das Integer-Bit der Signifikanden den Wert **1** hat, während
- bei den Unnormals das Integer-Bit der Signifikanden den Wert **0** hat.

Aus diesem Grund stehen den Normals alle 64 Bits des Signifikanden zur Verfügung. Die Unnormals können dagegen nur 63 signifikante Bits beanspruchen. Die Beträge der größten und kleinsten Normals sind daher **zweimal** so groß wie die der Unnormals:

(+–1.18..E4932 = 2 x +–0.59..E4932)

8.5 Spezielle Berechnungs-Situationen

Während der Ausführung von Numerik-Programmen kommt es manchmal zu speziellen Situationen, die entweder vom Programm selbst oder vom Exception-Handler interpretiert und behandelt werden müssen.

8.5.1 Ungültige arithmetische Operationen

Die Klasse der ungültigen arithmetischen Operationen ist im IEEE-Standard 754 definiert. Die Bedingungen, die dabei zur Exception **Ungültige Operation** führen, sind in der folgenden Tabelle aufgelistet. Sie zeigt zusätzlich die FPU-Antworten, wenn die Exception **maskiert** ist. Sollte sie **nicht** maskiert sein, generiert der Prozessor Interrupt 16 (Floating-Point-Error-Interrupt). Die Operanden bleiben in diesem Fall **unberührt**.

Maskierte Antworten auf ungültige arithmetische Operationen:

Bedingung	**maskierte Antwort**
In einer arithmetischen Operation wird ein nicht-unterstützes Format benutzt	Liefere das QNaN Indefinite
In einer arithmetischen Operation wird ein SNaN benutzt	Liefere ein QNaN (Siehe Regeln zur Generierung von QNaNs)
Vergleichs- und Testoperationen: Einer oder beide Operanden sind NaNs	Setze Condition-Code **nicht vergleichbar** (C3,C2,C0 = 111)
Addition von Infinities (+Unendlich, –Unendlich) mit entgegengesetzten Vorzeichen oder Subtraktion von Infinities mit gleichen Vorzeichen	Liefere das QNaN Indefinite
Multiplikation : $\infty \times 0$ oder $0 \times \infty$	Liefere das QNaN Indefinite
Division: $\frac{\infty}{\infty}$ oder $\frac{0}{0}$	Liefere das QNaN Indefinite
Remainder-Befehle FPREM und FPREM1: Der Modulus (Divisor) ist **Null** oder der Dividend ist **Unendlich**	Liefere das QNaN Indefinite und setze das Condition-Bit C2 = 1
Trigonometrische Befehle: FCOS, FPTAN, FSIN, FSINCOS: Das Argument ist **Unendlich**	Liefere das QNaN Indefinite und setze das Condition-Bit C2 = 1
FSQRT eines negativen Operanden (ausgenommen FSQRT(–0) = –0) oder FYL2X eines negativen Operanden (ausgenommen FYL2X(–0) = – ∞) oder FYL2XP1 eines Operanden, der kleiner als –1 ist	Liefere das QNaN Indefinite
FIST(P)-Befehle: Das Quellregister ist leer, ein NaN, ∞ oder der darstellbare Bereich des Ziels wird überschritten	Speichere Integer-Indefinite
FBSTP-Befehl: Das Quellregister ist leer, ein NaN, ∞ oder überschreitet 18 dezimale Digits	Speichere Packed Decimal-Indefinite
FXCH-Befehl: Einer oder beide Numerik-Register sind als leer gekennzeichnet	Ersetze die leeren Register durch das QNaN Indefinite und führe den Befehl aus

8.5.2 Die Exception Overflow

Diese Exception kann bei der Ausführung der meisten arithmetischen Befehle und bei einigen Konvertierungsbefehlen vorkommen, nämlich bei FST(P), F(I)ADD(P), F(I)SUB(R)(P), F(I)MUL(P), FDIV(R)(P), FSCALE, FYL2X und FYL2XP1.

8.5.2.1 Maskierter Overflow

Die Ergebnisse, die bei einem maskierten Overflow geliefert werden, hängen von der gewählten Rundungsart ab und sind in der folgenden Tabelle zusammengestellt:

Rundungsart	Vorzeichen des wahren Ergebnisses	Ergebnis
Zum nähesten Wert	+	+ ∞
	–	– ∞
Richtung – ∞	+	Größte endliche positive Zahl
	–	– ∞
Richtung + ∞	+	+ ∞
	–	Größte endliche negative Zahl
Richtung 0 (Chop)	+	Größte endliche positive Zahl
	–	Größte endliche negative Zahl

8.5.5.2 Nicht-maskierter Overflow

Die nicht-maskierte Antwort hängt vom Aufbewahrungsort des Operationsergebnisses ab. Dieser Ort ist entweder der **Numerik-Stack** oder der **Speicher**.

In beiden Fällen wird die Exception **Precision** angezeigt (PE=1).

- Ist das Ziel der Numerik-Stack, dann wird das Ergebnis durch 2^{24576} dividiert. Dadurch wird das Ergebnis so nahe wie möglich an die Mitte des Exponentenbereichs herangebracht. Außerdem wird der Signifikand in Übereinstimmung mit dem Precision-Control-Bit PC im Kontrollwort gerundet. Dies geschieht nur bei jenen Befehlen, die vom PC-Bit beeinflußt werden. Bei allen anderen Befehlen wird die erweiterte Genauigkeit von 64 Bits benutzt. Erfolgt die Rundung des Signifikanden in Richtung + ∞, wird zusätzlich im Statuswort das Bit **C1** (Roundup Bit) gesetzt (C=1).
- Ist das Ziel der Speicher (dies kommt nur bei STORE-Befehlen vor), dann wird dort kein Ergebnis hinterlegt. Das Ergebnis bleibt statt dessen unberührt im Numerik-Stack zurück.

8.5.3 Die Exception Underflow

Diese Exception kann bei der Ausführung der folgenden Befehle vorkommen: FST(P), FADD(P), FSUB(R)(P), FMUL(P), F(I)DIV(R)(P), FSCALE, FPREM(1), FPTAN, FSIN, FCOS, FSINCOS, FPATAN, F2XM1, FYL2X und FYL2XP1.

Dabei gibt es zwei miteinander verwandte Ereignisse, die zur Exception Underflow führen.

1. Die Erzeugung eines **sehr kleinen** Ergebnisses. Ist dies der Fall, dann kann dies zu einer weiteren Exception führen. So könnte es z.B. zur Exception Overflow durch eine nachfolgende Division kommen.
2. Die Erzeugung eines **ungenauen** Ergebnisses. Dies kann vorkommen, wenn sowohl der Exponenten-Bereich, als auch die Genauigkeit eingeschränkt sind.

8.5.3.1 Maskierter Underflow

Wenn die Exception Underflow maskiert ist, wird sie sowohl bei sehr kleinen als auch bei ungenauen Ergebnissen generiert. Die FPU liefert dann ein **Denormal** oder eine **Null**. Zusätzlich wird das PE-Bit (Precision Exception) im Statuswort gesetzt (PE=1).

8.5.3.2 Nicht-maskierter Underflow

Die nicht-maskierte Antwort hängt vom Aufbewahrungsort des Operationsergebnisses ab. Dieser Ort kann entweder der Numerik-Stack oder der Speicher sein.

In beiden Fällen wird die **Exception** Precision angezeigt (PE=1).

- Ist das Ziel der Numerik-Stack, dann wird das Ergebnis mit 2^{24576} multipliziert. Dadurch wird das Ergebnis so nahe wie möglich an die Mitte des Exponentenbereichs herangebracht. Außerdem wird der Signifikand in Übereinstimmung mit dem Precision-Control-Bit PC im Kontrollwort gerundet. Dies geschieht nur bei jenen Befehlen, die vom PC-Bit beeinflußt werden. Bei allen anderen Befehlen wird die erweiterte Genauigkeit von 64 Bits benutzt. Erfolgt die Rundung des Signifikanden in Richtung $+\infty$, wird zusätzlich im Statuswort das Bit **C1** (Roundup Bit) gesetzt (C=1).
- Ist das Ziel der Speicher (dies kommt nur bei STORE-Befehlen vor), dann wird dort kein Ergebnis hinterlegt. Das Ergebnis bleibt statt dessen unberührt im Numerik-Stack zurück.

8.5.4 Die Exception Precision

Diese Exception kommt häufig vor und bedeutet immer einen Verlust von Genauigkeit. Sie tritt auf, wenn das Ergebnis einer Operation **nicht exakt** in binärer Form dargestellt werden kann. Der Quotient 1/3 ist hierfür ein Beispiel.

Wenn das PE-Bit (Precision Exception) im Statuswort gesetzt ist (PE=1), dann bekommt bei einigen Befehlen das **Condition**-Bit **C1** im Statuswort die Bedeutung von **Roundup**. Die betreffenden Befehle sind in der folgenden Tabelle aufgelistet.

Befehl	**C0**	**C3**	**C1**	**C2**
FIST, FBSTP, FRNDINT, FST, FSTP, FADD, FMUL, FDIV, FDIVR, FSUB, FSUBR, FSCALE, FSQRT, FPATAN, F2XM1, FYL2X, FYL2XP1	Undefiniert		Roundup	Undefiniert
FPTAN, FSIN, FCOS, FSINCOS	Undefiniert		Roundup oder Undefiniert wenn C2=1	0 = Argument liegt im Bereich 1 = Argument liegt nicht im Bereich

Die meisten dieser Befehle beeinflussen **nicht** die restlichen Condition-Bits C3, C2, C0. Ihr Status ist undefiniert. Davon ausgenommen sind die trigonometrischen Befehle FPTAN, FSIN, FCOS und FSINCOS, die auch das Condition-Bit C2 beeinflussen. (Siehe 8.5.6.3).

Betrachten wir zunächst nur das Roundup-Bit C1. Es zeigt an, ob einer der aufgelisteten Befehle das ungenaue Ergebnis

- in Richtung +Unendlich **aufgerundet** (C = 1) oder
- in Richtung Null **abgeschnitten** (C = 0) hat (Chop).

Hinweis:
Sollte es bei einem **Underflow** zusätzlich zu einem ungenauen Ergebnis kommen, dann wird neben der Underflow-Exception (UE=1) auch die Precision-Exception angezeigt (PE=1).

8.5.5 Laden von Denormals

Wenn die FPU mit FLD (Lade Realzahl) einen Denormal-Operanden im **SHORT**-Realformat oder **LONG**-Realformat lädt, dann konvertiert sie das Denormal in ein äquivalentes Normal im TEMPORARY-Realformat. Wie dies geschieht, illustriert die folgende Tabelle am Beispiel eines Denormals im LONG-Realformat:

Operation	Vorzeichen (1 Bit)	Exponent (11 Bits)	Signifikand (52 Bits)
Denormal	0	–1022	0.001010..00
Normalisieren	0	–1023	0.01010...00
Normalisieren	0	–1024	0.1010....00
Normalisieren	0	–1025	1.010.....00
Normalisiertes Ergebnis	0	–1025	**1**.010.....00

Im LONG-Realformat hat der kleinste darstellbare Exponent den Wert Emin = –1022. In den Exponentenfeldern aller LONG-Denormals ist dieser Wert als Bitstring 00..00 (Emin+1022=0) gespeichert. Beim **Normalisierungsprozeß** wird der Exponent des Denormals, also –1022 solange **dekrementiert** und gleichzeitig der Signifikand um eine Binärstelle nach **links** verschoben, bis im Signifikanden links vom Gleitpunkt eine **1** erscheint. Das resultierende Normal hat auf diese Weise den gleichen Wert wie das ursprüngliche Denormal. Im Beispiel bekommt der Exponent den Wert –1025, der im 15-Bit-Exponentenfeld des internen TEMPORARY-Realformats dargestellt werden kann. Das Integer-Bit des neuen Signifikanden mit dem Wert **1,** wird wie bei Normals im TEMPORARY-Realformat üblich, im 64-Bit-Signifikandenfeld mit abgespeichert.

8.5.6 Interpretation der Condition-Codes

Es gibt eine Klasse von Befehlen, mit deren Hilfe alle von der FPU unterstützten Real- und Inter-Datentypen **verglichen**, **geprüft** und **getestet** werden können. Jeder dieser Befehle analysiert das **Top**-Stackregister (oft auch im Beziehung zu anderen Operanden) und meldet das Ergebnis als Condition-Code (Bits C3, C2 und C0) im Statuswort.

In der folgenden Tabelle sind die Mitglieder dieser Befehlsklasse zusammengefaßt:

Vergleichs und Testbefehle:

Mnemonik	Operation
F(U)COM	Vergleiche Realzahlen
F(U)COMP	Vergleiche Realzahlen und Pop
F(U)COMPP	Vergleiche Realzahlenund Pop zweimal
FICOM	Vergleiche Integerzahlen
FICOMP	Vergleiche Integerzahlen und Pop
FTST	Teste Top-Stackregister
FXAM	Prüfe Top-Stackregister

8.5.6.1 Condition-Codes nach Examine

Der Examine-Befehl FXAM (Examine) überprüft den Inhalt des **Top**-Stackregisters und gibt Auskunft über die Klasse des gespeicherten Operanden. Zu diesem Zweck belegt FXAM die Condition-Bits C0..C3 mit bestimmten Codes, die in der folgenden Tabelle aufgelistet sind:

Tabelle der Condition-Codes:

C3	C2	C1	C0	Operand im TOP-Stackregister
0	0	0	0	+Nicht unterstützt
0	0	0	0	+NaN
0	0	1	0	–Nicht unterstützt
0	0	1	1	–NaN
0	1	0	0	+Normal
0	1	0	1	+Unendlich
0	1	1	0	–Normal
0	1	1	1	–Unendlich
1	0	0	0	+0
1	0	0	1	+Leer
1	0	1	0	–Null
1	0	1	1	–Leer
1	1	0	0	+Denormal
1	1	1	0	–Denormal

Wenn es während der Ausführung von Numerik-Programmen zu Exceptions kommt, dann liegt es im Interesse des Programmierers, daß der Exception-Handler den fehlerverursachenden Operanden imTop-Stackregister feststellt und meldet. Auf welche Weise dies geschehen kann, zeigt das folgende Assembler-Programm.

Es benutzt den Vektor fxam_tabelle, der aus 14 Doubleword-Elementen (Anzahl der Operanden-Typen) besteht. Jedes dieser Elemente ist mit dem OFFSET einer korrespondierenden NEAR-Prozedur initialisiert. Das Programm führt zunächst fxam aus und speichert das FPU-Statuswort in der Wordvariablen statuswort. Danach wird der augenblickliche Condition-Code (C3, C2, C1, C0) in einen **Index** umgewandelt und im EBX-Register der CPU gespeichert. Dieser Index wird vom Befehl

- call fxam_tabelle[ebx]

benutzt. Er selektiert zunächst über EBX ein bestimmtes Element im Vektor fxam_tabelle und ruft danach die korrespondierende Prozedur auf. Diese kann daraufhin den Typ des Operanden melden.

```
DOS 3.20 (045-N) 386(TM) MACRO ASSEMBLER EXAM
                                    1          NAME exam
                                    2
                                    3  ASSUME DS:DATA
                                    4
                                    5  CODE32 SEGMENT USE32 EO
                                    6
```

```
00000000 D9E5                    7 exam:  fxam                          ;Pruefe Top-Stackregister
00000002 9BDD3D38000000     R    8        fstsw statuswort              ;Speichere Statuswort
00000009 9B                      9        fwait                         ;Warten, bis Statuswort gespeichert
0000000A BB00000000             10        mov  ebx,0;                   Loesche EBX-Register
0000000F 8A1D39000000       R   11        mov  bl,byte ptr statuswort+1 ;Lade Condition-Code in das
                                12                                      ;Low-Byte von BX
00000015 88D8                   13        mov  al,bl                    ;Kopiere Condition-Code
00000017 80E307                 14        and  bl,00000111B             ;Loesche alle Bits ausser C2,C1,C0
0000001A 2440                   15        and  al,01000000B             ;Loesche alle Bits ausser C3
0000001C C0E802                 16        shr  qal,2                    ;Schiebe C3 zwei Bitpositionen
                                17                                      ;nach rechts
                                19                                        ;nach links
00000022 08C3                   20        or   bl,al                    ;Bilde BX=000 C3C2C1C0 0 binaer
00000024 FF9300000000       R   21        call fxam_tabelle[ebx]        ;Aufruf der Prozedur, die durch
                                22                                        ;den Condition-Code in EBX
                                23                 ;                      ;adressiert ist
                                24                 ;
                                25
                                26 ;Moegliche Prozeduren, die abhaengig vom Condition-Code aufgerufen
                                27 ;werden koennen
                                28
0000002A                        29 plus_Nicht_unterstuetzt               PROC NEAR ; - - - - - - - - - - - - - - - - -
                                30                                       ;
0000002A C3                     31                                       ret
0000002B                        32 plus_Nicht_unterstuetzt               ENDP
                                33
0000002B                        34 plus_NaN                              PROC NEAR ; - - - - - - - - - - - - - - - - -
                                35                                       ;
0000002B C3                     36                                       ret
0000002C                        37 plus_NaN                              ENDP
                                38
0000002C                        39 minus_Nicht_unterstuetzt              PROC NEAR ; - - - - - - - - - - - - - - - - -
                                40                                       ;
0000002C C3                     41                                       ret
0000002D                        42 minus_Nicht_unterstuetzt              ENDP
                                43
0000002D                        44 minus_NaN                             PROC NEAR ; - - - - - - - - - - - - - - - - -
                                45                                       ;
0000002D C3                     46                                       ret
0000002E                        47          minus_NaN                    ENDP
                                48
0000002E                        49 plus_Normal                           PROC NEAR ; - - - - - - - - - - - - - - - - -
                                50                                       ;
0000002E C3                     51                                       ret
0000002F                        52 plus_Normal                           ENDP
                                53
0000002F                        54 plus_Unendlich                        PROC NEAR ; - - - - - - - - - - - - - - - - -
                                55                                       ;
0000002F C3                     56 ret
00000030                        57 plus_Unendlich ENDP
                                58
00000030                        59 minus_Normal                          PROC NEAR ; - - - - - - - - - - - - - - - - -
                                60                                       ;
00000030 C3                     61                                       ret
00000031                        62 minus_Normal                          ENDP
                                63
00000031                        64 minus_Unendlich                       PROC NEAR ; - - - - - - - - - - - - - - - - -
                                65                                       ;
00000031 C3                     66                                       ret
00000032                        67 minus_Unendlich                       ENDP
                                68
00000032                        69 plus_Null                             PROC NEAR ; - - - - - - - - - - - - - - - - -
                                70                                       ;
```

```
00000032 C3                    71                                  ret
00000033                       72  plus_Null                       ENDP
                               73
00000033                       74  plus_Leer                       PROC NEAR ; ------------------
                               75                                  ;
00000033 C3                    76                                  ret
00000034                       77  plus_Leer                       ENDP
                               78
00000034                       79  minus_Null                      PROC NEAR ; ------------------
                               80                                  ;
00000034 C3                    81                                  ret
00000035                       82  minus_Null                      ENDP
                               83
00000035                       84  minus_Leer                      PROC NEAR ; ------------------
                               85                                  ;
00000035 C3                    86                                  ret
00000036                       87  minus_Leer                      ENDP
                               88
00000036                       89  plus_Denormal                   PROC NEAR ; ------------------
                               90                                  ;
00000036 C3                    91                                  ret
00000037                       92  plus_Denormal ENDP
                               93
00000037                       94  minus_Denormal                  PROC NEAR ; ------------------
                               95                                  ;
00000037 C3                    96                                  ret
00000038                       97  minus_Denormal                  ENDP
                               98
---------                      99  CODE32 ENDS
                              100
--------                      101  DATA                            SEGMENT USE32 RW
                              102
00000000 2A000000          R  103  fxam_tabelle  dd plus_Nicht_unterstuetzt,plus_NaN,minus_Nicht_unter
                                                   stuetzt,
00000004 2B000000          R  104  &             minus_NaN,plus_Normal,plus_Unendlich,minus_Normal,
00000008 2C000000          R  105  &             minus_Unendlich,plus_Null,plus_Leer,minus_Null,minus_Leer,
0000000C 2D000000          R  106  &             plus_Denormal,minus_Denormal
00000010 2E000000
00000014 2F000000
00000018 30000000
0000001C 31000000
00000020 32000000
00000024 33000000
00000028 34000000
0000002C 35000000
00000030 36000000
00000034 37000000
                              107
00000038 ????                 108  statuswort     dw ?                                ;Statuswort
                              109
--------                      110          DATA  ENDS
                              111          END

ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS.
```

8.5.6.2 Condition-Codes nach Compare und Test

1. Compare:

Die Compare-Befehle FCOM, FUCOM, FCOMP, FUCOMP, FCOMPP, FUCOMPP, FICOM, FICOMP **vergleichen** einen Operanden im **Top**-Stack-Register ST(0) mit einem Operanden im Speicher oder in einem anderen Numerik-Stack-Register. Das Vergleichsergebnis geben anschließend die Condition-Bits C3, C2 und C0 bekannt und zwar in Übereinstimmung mit der folgenden Tabelle:

Vergleichsergebnis	**C3 (ZF)**	**C2 (PF)**	**C0 (CF)**	**Bedingte * Verzweigung**
ST > Operand	0	0	0	JA
ST < Operand	0	0	1	JB
ST = Operand	1	0	0	JE
Nicht vergleichbar	1	1	1	JP

Wenn einer der Operanden ein NaN ist, oder ein nicht-unterstütztes Format hat oder wenn es zu einer Stack-Fault-Exception kommt, dann wird der Condition-Code „nicht vergleichbar" (C3, C2, C0 = 111) gesetzt. Zusätzlich wird die Exception **Invalid Operation** angezeigt (IE=1).

2. Test

Der Test-Befehl FTST vergleicht den Inhalt des **Top**-Stack-Registers ST(0) mit **0**. Dabei wird das Vorzeichen von Null ignoriert, d.h. +0.0 = –0.0. Das Testergebnis geben anschließend die Condition-Bits C3, C2 und C0 bekannt und zwar in Übereinstimmung mit der folgenden Tabelle:

Testergebnis	**C3 (ZF)**	**C2 (PF)**	**C0 (CF)**	**Bedingte * Verzweigung**
ST > 0.0	0	0	0	JA
ST < 0.0	0	0	1	JB
ST = 0.0	1	0	0	JE
Nicht testbar	1	1	1	JP

Wenn der Operand in ST(0) ein NaN ist, oder ein nicht-unterstütztes Format hat oder wenn es zu einer Stack-Fault-Exception kommt, dann wird der Condition-Code „nicht testbar" (C3, C2, C0 = 111) gesetzt. Zusätzlich wird die Exception **Invalid Operation** angezeigt (IE=1).

* **JA** – Jump if Above (Springe, wenn ZF und CF = 0)
JB – Jump if Below (Springe, wenn CF = 1)
JE – Jump if Equal (Springe, wenn ZF = 1)
JP – Jump if Parity (Springe, wenn PF = 1)

Der folgende Assembler-Code zeigt, wie nach der Ausführung eines Vergleichs- oder Testbefehls die Condition-Bits ausgewertet werden können.

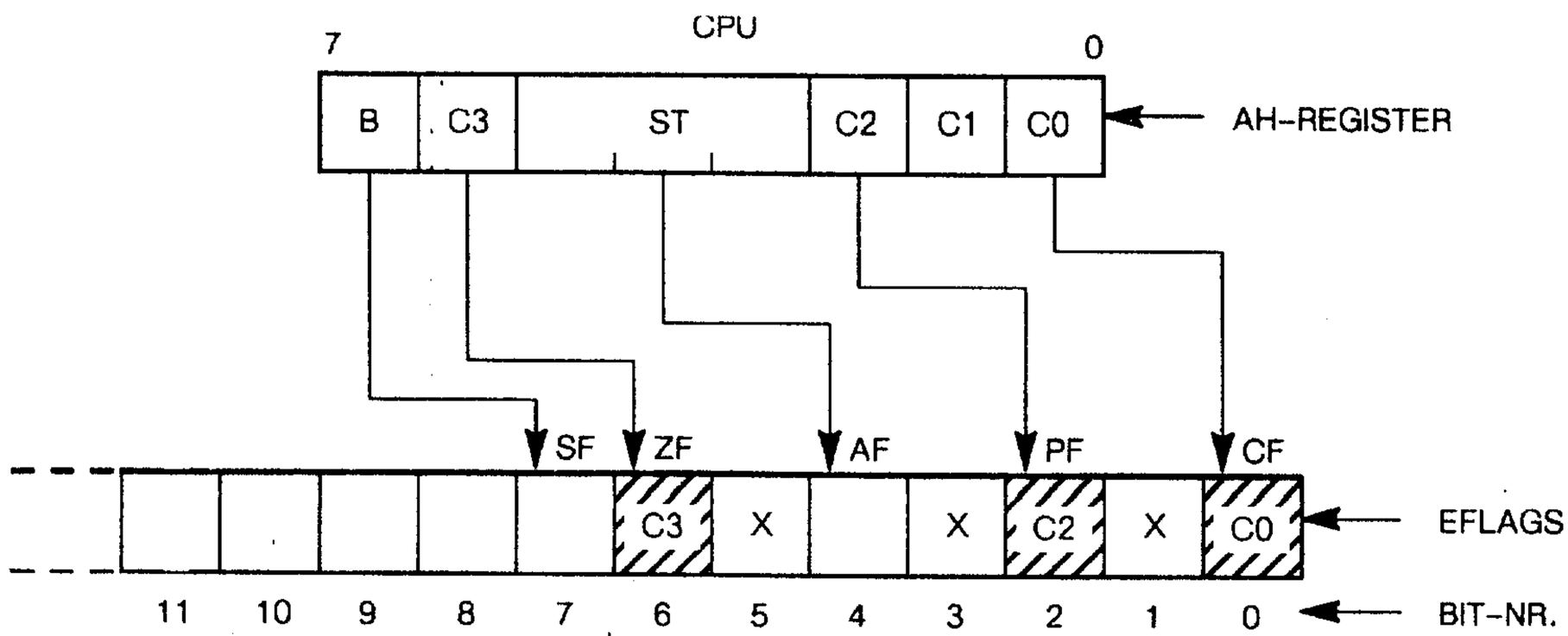
In der Prozedur **auswertung** speichert zunächst der Befehl

- FSTSW AX

das FPU-Statuswort im AX-Register. Danach überträgt

- SAHF

das AH-Register in die EFLAGs SF, ZF, X, AF, X, PF, X, CF. Dabei gelangt **C0** in das **Carry**-Flag (CF), **C2** in das **Parity**-Flag (PF) und **C3** in das in das **Zero**-Flag (ZF). Im folgenden Bild ist dies illustriert:



Danach erfolgt die Auswertung der Condition-Bits über die bedingten Sprungbefehle JA, JB, JE und JP, die abhängig vom Status der korrespondierenden EFLAGs zu den passenden Labels verzweigen. Dort kommt es dann zu individuellen anwenderspezifischen Aktionen.

```
LOC       OBJ          LINE    SOURCE

                        1              NAME      comtst
                        2
--------                3     CODE32   SEGMENT   EO USE32
                        4              ;
                        5              ;                                ;fcom oder ftst-Befehl
00000000 E800000000     6              call auswertung
                        7              ;
00000005                8     auswertung PROC NEAR
00000005 9BDFE0         9              fstsw     ax                     ;Speichere nach fcom oder ftst das
                       10                                               ;FPU-Statuswort im AX-Register
00000008 9E            11              sahf                             ;Uebertrage AH-Register in die EFLAGs
                       12                                               ;SF ZF xx AF xx PF xx CF
                       13
00000009 7706          14              ja        top_stack_groesser     ;Springe, wenn ST>Operand oder ST>0.0
0000000B 7206          15              jb        top_stack_kleiner      ;Springe, wenn ST<Operand oder ST<0.0
0000000D 7406          16              je        top_stack_gleich       ;Springe, wenn ST=Operand oder ST=0.0
0000000F 7A06          17              jp        top_stack_nicht_vergleich_oder_testbar
                       18                                               ;Springe, wenn ST nicht vergleich- oder
                       19                                               ;testbar
                       20
00000011               21     top_stack_groesser:
```

```
                    22              ;anwenderspezifische Aktion
00000011 EB06       23              jmp         @
                    24
00000013            25      top_stack_kleiner:
                    26              ;anwenderspezifische Aktion
00000013 EB04       27              jmp         @
                    28
00000015            29      top_stack_gleich:
                    30              ;anwenderspezifische Aktion
00000015 EB02       31              jmp         @
                    32
00000017            33      top_stack_nicht_vergleich_oder_testbar:
                    34              ;anwenderspezifische Aktion
00000017 EB00       35              jmp         @
00000019 C3         36      @:      ret
                    37
0000001A            38      auswertung ENDP
                    39
--------            40      CODE32  ENDS
                    41              END

ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS.
```

8.5.6.3 Condition-Codes nach FPREM und FPREM1 (Partial Remainder)

Mit den Remainder-Befehlen FPREM und FPREM1 wird eine Modulo-Division zwischen dem **Top**-Stack-Register ST(0) = **Dividend** und ST(1) = **Modulus** durchgeführt. Als Ergebnis liefern die Befehle den sogenannten **Remainder**, der den Dividenden in ST(0) überschreibt:

- ST(0) <— ST(0) MOD ST(1)

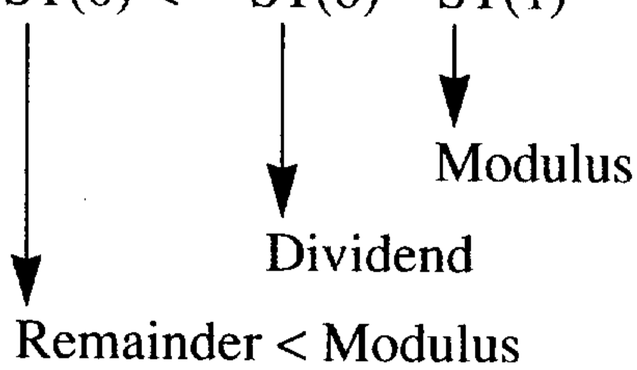
Das Vorzeichen des Remainders entspricht dem Vorzeichen des Dividenden. Die Befehle FPREM/FPREM1 liefern immer ein exaktes Ergebnis. Da also kein Runden notwendig ist, wird die Rundungsfunktion ignoriert.

Die Modulo-Division wird durchgeführt, indem FPREM oder FPREM1 den Modulus in ST(1) vom Dividenden in ST(0) subtrahieren:

- ST(0) <— ST(0) – ST(1)

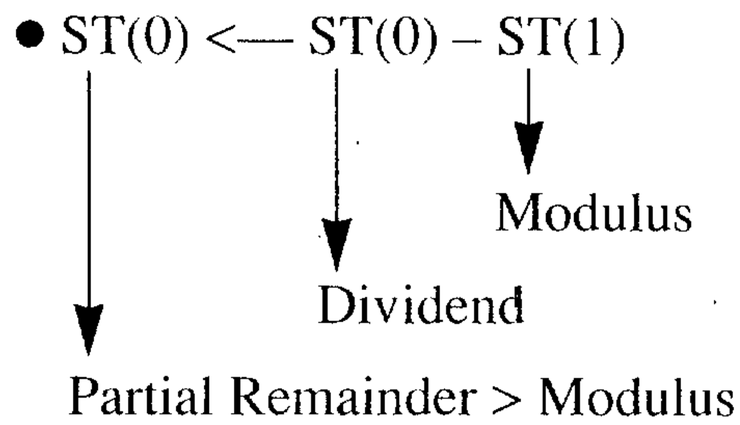
Wenn FPREM oder FPREM1 einen Remainder liefern, der **kleiner** als der Modulus ist, dann ist die Funktion beendet und das Condition-Bit **C2** wird gelöscht (C2=0):

- ST(0) <— ST(0) – ST(1)



Die Funktion ist ebenfalls beendet, wenn der Remainder **0** ist. In diesem Fall sind der Dividend und der Modulus **identisch**.

Wenn aber FPREM oder FPREM1 einen Remainder liefern, der **größer** als der Modulus ist, dann ist die Funktion **nicht** beendet und das Condition-Bit **C2** wird gesetzt (C2=1):



Das Ergebnis ist der sogenannte **Partial Remainder**. Hier muß dann in einer Programmschleife der Befehl FPREM oder FPREM1 solange wiederholt werden, bis die Bedingungen

- Remainder < Modulus oder
- Remainder = Modulus

erfüllt sind. Dabei wird der Partial Remainder in ST(0) als Dividend benutzt:

- ST(0) <— Repeat [ST(0) – ST(1)]

Verfahren: Schreibe nach der Ausführung von FPREM oder FPREM1 das FPU-Statuswort mit FSTSW (Speicher Statuswort) in den Speicher oder in das AX-Register. Prüfe anschließend mit einem CPU-Maskenbefehl (TEST oder AND) den Status von C2. Ist **C2 = 1**, dann **wiederhole** FPREM oder FPREM1. Ist **C2 = 0**, dann ist die Funktion **beendet**.

Wenn der Befehl FPREM die Modulo-Division durchführt, ermittelt er neben dem Remainder **REM** auch einen Quotienten **Q**. Ähnliches gilt für den Befehl FPREM1. Er ermittelt neben dem Remainder **REM1** einen Quotienten **Q1**. Sowohl Q als auch Q1 werden aus dem wahren Quotienten ST(0)/ST(1) = i.ff..ff abgeleitet und zwar mit folgendem Unterschied:

- **FPREM** schneidet den gebrochenen Anteil .ff..ff des wahren Quotienten ST(0)/ST(1) ab und liefert die **Integer**zahl **Q = i**.
- **FPREM1** liefert dagegen eine **Integer**zahl **Q1**, die am **nähesten** beim wahren Quotienten ST(0)/ST(1) liegt. Wenn der gebrochene Anteil .ff..ff = 0.5 ist, gibt es zwei Integerzahlen, die vom wahren Quotienten gleichweit entfernt sind. Bei dieser Situation bekommt Q1 den **geradzahligen** Integerwert.

Mit den Informationen Dividend in ST(0), Modulus in ST(1) und Quotient Q oder Q1 lassen sich die von FPREM und FPREM1 gelieferten Remainder durch folgende Formeln berechnen:

Befehl	Remainder
FPREM	REM = ST(0) – ST(1)Q
FPREM1	REM1 = ST(0) – ST(1)Q1

Wie sind nun diese Formeln zu interpretieren? Am Beispiel der wichtigsten Anwendung von FPREM/FPREM1 soll diese Frage beantwortet werden.

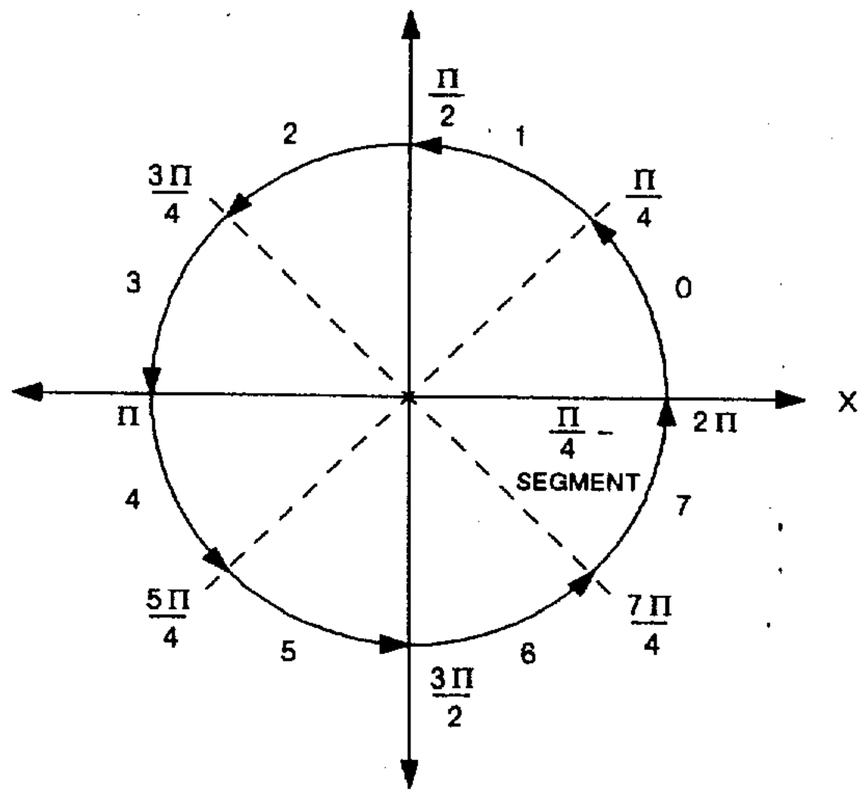
Bei trigonometrischen Funktionen ist es notwendig, die Argumente in ST(0) in einen Bereich zu bringen, die für diese Funktionen erlaubt sind. So lassen die Befehle FSIN (Sinus), FCOS (Cosinus), FSINCOS (kombinierter Sinus-Cosinus) und FPTAN (partieller Tangens) „nur" Argumente im Bereich von $-2^{63} < ST(0) < +2^{63}$ zu. Wenn das Argument einer trigonometrischen Funktion **in** diesem Bereich liegt, dann **reduzieren** diese Befehle automatisch das Argument um ein passendes Mehrfaches von 2π. (π hat eine Genauigkeit von 66 Bits) und liefern das Ergebnis. Das Condition-Bit **C2** wird in diesem Fall gelöscht (C2=0).

Liegt dagegen das Argument **außerhalb** des akzeptierten Bereichs, dann wird das Condition-Bit **C2** gesetzt (C2=1) und das Argument bleibt unberührt.

In diesem Fall kann FPREM oder FPREM1 benutzt werden, um das Argument in ST(0) in den erlaubten Bereich zu bringen. FPREM/FPREM1 erlauben zwischen dem Dividenden und dem Modulus eine Differenz von bis zu 2^{64} = 9.223372036854776E+18. Mit anderen Worten, beide Befehle können das Original-Argument um diesen Betrag durch **einmalige** Ausführung reduzieren.

- Wenn sich bei der Reduktion REM/REM1 <= Modulus ergibt, dann wird C2 gelöscht (C2=0). Damit wird angezeigt, daß die Differenz zwischen dem Dividenden und dem Modulus **kleiner** oder **gleich** 2^{64} ist.
- Wenn sich bei der Reduktion REM/REM1 > Modulus ergibt, dann wird C2 gesetzt (C2=1). Damit wird angezeigt, daß die Differenz zwischen dem Dividenden und dem Modulus **größer** als 2^{64} ist. In diesem Fall kann die komplette Reduktion des Arguments nur durch eine Wiederholung der FPREM/FPREM1-Befehle erreicht werden.

Benützen die Befehle FPREM/FPREM1 als Modulus $\frac{\pi}{4}$, dann zeigen die Condition-Bits **C3**, **C1** und **C0** die Lage des reduzierten Winkels im Einheitskreis an. Damit der Programmierer den gelieferten Condition-Code interpretieren kann, empfiehlt es sich, den Einheitskreis gedanklich in **acht** $\frac{\pi}{4}$-Segmente mit den Nummern **0** bis **7** zu unterteilen, so wie es im folgenden Bild illustriert ist:



Hierzu nun ein Beispiel. Der Originalwinkel sei $\Theta > 2^{63}$ Rad. Weil der Winkel außerhalb von $-2^{63} < \Theta < +2^{63}$ liegt, muß er mit FPREM oder FPREM1 reduziert werden.

Beide Befehle berechnen zunächst den wahren Quotienten

$$\frac{ST(0)}{ST(1)} = \frac{\text{Originalwinkel}}{\text{Modulus } \frac{\pi}{4}} = \frac{>2^{63}\text{ Rad}}{0.785..\text{Rad}} = \text{i.ff..ff}$$

Wenn wir der Einfachheit halber annehmen, daß .ff..ff <= 0.5 ist, dann ermitteln FPREM und FPREM1 die gleiche Integerzahl, nämlich Q = Q1 = i. Unter dieser Voraussetzung liefern dann beide Befehle identische Remainder:

- FPREM: REM = ST(0) – ST(1)Q => 2^{63} Rad – 0.785..Rad·i
- FPREM1: REM1 = ST(0) – ST(1)Q1 => 2^{63} Rad – 0.785..Rad·i

Wenn wir weiter annehmen, daß sich z.B. für

- **REM** bzw. **REM1 = 0.39..Rad**

ergibt, dann ist die Reduktion gelungen. Die Bedingung REM bzw. REM1 <= Modulus (0.39..Rad <= 0.785..Rad) ist erfüllt und das Condition-Bit C2 ist gelöscht (C2=0).

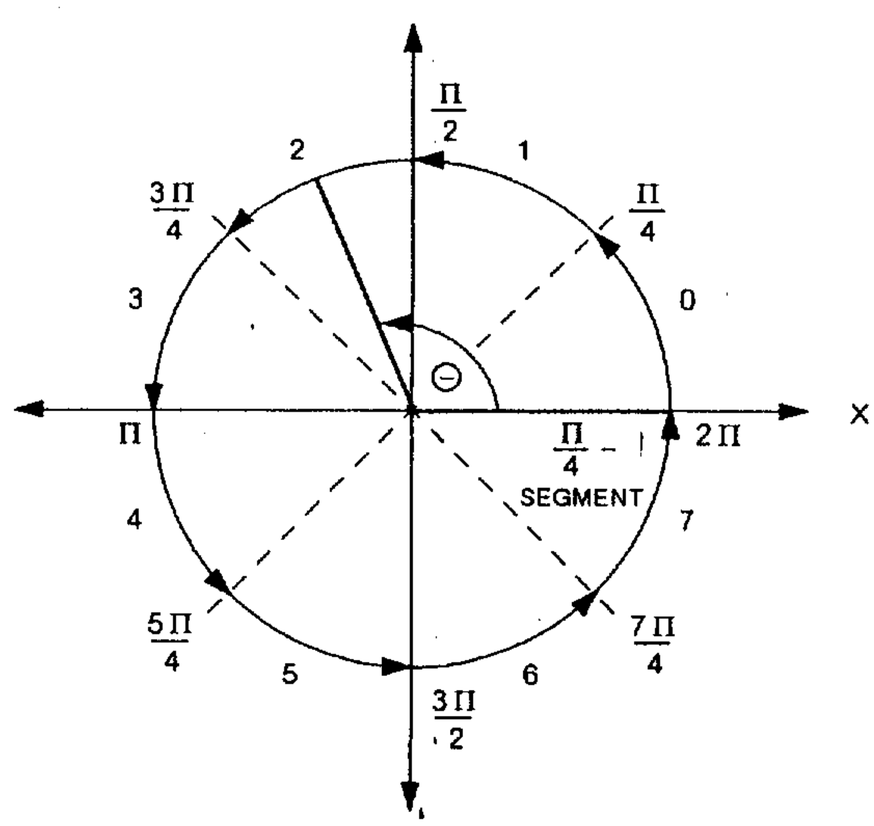
Man kann nun den Condition-Code und damit die Lage des reduzierten Winkels im Einheitskreis auf einfache Weise durch die Modulo-Divisionen

- Q MOD 8 oder Q1 MOD 8

ermitteln. Weil Q=Q1=i ist, ergibt sich i MOD 8 = x Rest **y**. Dabei ist der Restwert y eine Integerzahl im Bereich von 0..7 und ist identisch mit dem von der FPU gelieferten Condition-Code. Wenn z.B **y = 2** ist, dann bedeutet dies, daß der Originalwinkel $\Theta > 2^{63}$ von FPREM/FPREM1 auf einen Winkel reduziert wurde, der sich im **2ten** $\frac{\pi}{4}$-Segment des Einheitskreises befindet. Der reduzierte Winkel ergibt sich dann zu

- $2\frac{\pi}{4}$ Rad + 0.39..Rad = 90 Grad + 22,57 Grad = **112,57 Grad**

Er liegt im Definitionsbereich der trigonometrischen Funktionen. Seine Lage illustriert das folgende Bild:



Der Originalwinkel $\Theta > 2^{63}$ Rad entspricht dem reduzierten Winkel Θ = 112,57 Grad

Die FPU hinterlegt in den Condition-Bits C3,C2,C1,C0 folgenden Code:

Q1	–	Q0	Q2
C3	C2	C1	C0
1	0	0	0

In der Tabelle ist zu sehen, daß die Condition-Bits C3, C1 und C0 jetzt die Bedeutung von Q1,Q0 und Q2 haben. Sie sind identisch mit den letzten drei signifikanten Bits der Integerquotienten Q bzw. Q1.

Wenn man die signifikanten Bits in der gewohnten Weise anordnet, dann kann man leicht erkennen, daß sie den Dezimalwert 2 repräsentieren:

Q2	Q1	Q0
0	1	0

binär = 2 dezimal

Eine vollständige Interpretation der Condition-Codes nach der einmaligen Ausführung der Befehle FPREM/FPREM1 zeigt die folgende Tabelle:

Condition-Code				**Interpretation nach FPREM/FPREM1**	
C2	**C3**	**C1**	**C0**		
1	X	X	X		Unvollständige Reduktion: Für eine komplette Reduktion ist die Wiederholung von FPREM/FPREM1 erforderlich
0	Q1	Q0	Q2	Q/Q1 MOD 8	Vollständige Reduktion: C3, C1, C0 enthalten die letzten drei signifikanten Bits der Integerquotienten Q/Q1
	0	0	0	0	
	0	1	0	1	
	1	0	0	2	
	1	1	0	3	
	0	0	1	4	
	0	1	1	5	
	1	0	1	6	
	1	1	1	7	

FPREM1 unterscheidet sich von FPREM in folgenden Punkten:

- Da FPREM1 den Integerquotienten anders ermittelt als FPREM, können sich in manchen Fällen die Condition-Codes C3, C1, C0 um **ein** Bit unterscheiden.
- Abhängig vom Vorzeichen des Dividenden in ST(0) produzieren FPREM und FPREM1 unterschiedliche Remainder.

Bei **FPREM** liegt der Remainder REM im Bereich von
$0 < REM < |ST(1)|$ oder $-|ST(1)| < REM < 0$.
Bei **FPREM1** liegt der Remainder REM1 im Bereich von
$-|ST(1)|/2 < REM1 < +|ST(1)|/2$.

8.5.7 Behandlung infiniter Operanden

Infinite Operanden kann der Programmierer selbst codieren oder sie werden von der FPU **maskierte** als Antwort auf eine **Overflow**- oder **Zero Divide**-Exception produziert. Eine Arithmetik mit infiniten Operanden ist immer exakt und führt zu keinen Exceptions. Davon ausgenommen sind ungültige Operationen, die in der folgenden Tabelle aufgelistet sind:

Operation	Operanden			Ergebnis
FLD, FBLD	$+-\infty$			(1) ∞
FST, FSTP, FRNDINT	$+-\infty$			(1) ∞
FCHS	$+\infty$			$-\infty$
	$-\infty$			$+\infty$
FABS	$+-\infty$			$+\infty$
Addition	$+\infty$	plus	$+\infty$	$+\infty$
	$-\infty$	plus	$-\infty$	$-\infty$
	$+\infty$	plus	$-\infty$	Ungültige Operation
	$-\infty$	plus	$+\infty$	Ungültige Operation
	$+-\infty$	plus	$+-X$	(1) ∞
	$+-X$	plus	$+-\infty$	(1) ∞
Subtraktion	$+\infty$	minus	$-\infty$	$+\infty$
	$-\infty$	minus	$+\infty$	$-\infty$
	$+\infty$	minus	$+\infty$	Ungültige Operation
	$-\infty$	minus	$-\infty$	Ungültige Operation
	$+-\infty$	minus	$+-X$	(1) ∞
	$+-X$	minus	$+-\infty$	(2) ∞
Multiplikation	$+-\infty$	mal	$+-\infty$	(3) ∞
	$+-\infty$	mal	$+-Y$	(3) ∞
	$+-Y$	mal	$+-\infty$	(3) ∞
	$+-0$	mal	$+-\infty$	Ungültige Operation
	$+-\infty$	mal	$+-0$	Ungültige Operation
Division	$+-\infty$	div	$+-\infty$	Ungültige Operation
	$+-\infty$	div	$+-X$	(3) ∞
	$+-X$	div	$+-\infty$	(3) 0
	$+-\infty$	div	$+-0$	(3) ∞
FPREM, FPREM1	$+-\infty$	rem	$+-\infty$	Ungültige Operation
	$+-\infty$	rem	$+-X$	Ungültige Operation
	$+-X$	rem	$+-\infty$	(4) X, Q/Q1 = 0
FSQRT	$-\infty$			Ungültige Operation
	$+\infty$			$+\infty$
Compare	$+\infty$	comp	$+\infty$	$+\infty = +\infty$
	$-\infty$	comp	$-\infty$	$-\infty = -\infty$
	$+\infty$	comp	$-\infty$	$+\infty > -\infty$
	$-\infty$	comp	$+\infty$	$-\infty < +\infty$
	$+\infty$	comp	$+-X$	$+\infty > X$
	$-\infty$	comp	$+-X$	$-\infty < X$
	$+-X$	comp	$+\infty$	$X < +\infty$
	$+-X$	comp	$-\infty$	$X > +\infty$

Operation	Operanden	Ergebnis
FTST	+ ∞ – ∞	+ ∞ > 0 – ∞ < 0
FSCALE	+ – ∞ skaliert mit – ∞ + – ∞ skaliert mit + ∞ + – ∞ skaliert mit + –X + –0 skaliert mit – ∞ + –0 skaliert mit + – ∞ + –Y skaliert mit + ∞ + –Y skaliert mit – ∞	Ungültige Operation (1) ∞ (1) ∞ + –0 (5) Ungültige Operation (6) ∞ (6) 0
FXTRACT	+ – ∞	ST(0) = (1) ∞; ST(1) = + ∞
FXAM	+ ∞ – ∞	C0=C2=1; C1=C3=0 C0=C1=C2=1; C3=0
FPATAN	arctan(+ – ∞/+ –X) arctan(+ –Y/+ ∞) arctan(+ –Y/– ∞) arctan(+ – ∞/+ ∞) arctan(+ – ∞/– ∞) arctan(+ – ∞/+ –0) arctan(+0/+ ∞) arctan(+0/– ∞) arctan(–0/+ ∞) arctan(–0/– ∞)	(1) π/2 (6) 0 (6) π (1) π/4 (1) 3π/4 (1) π/2 +0 +π –0 –π
F2XM1	+ ∞ – ∞	+ ∞ –1
FYL2X	+ – ∞ x log(1) + – ∞ x log(X>1) + – ∞ x log(0<X<1) + –Y x log(+ ∞) + –0 x log(+ ∞) + –Y x log(– ∞)	Ungültige Operation (1) ∞ (2) ∞ (6) ∞ Ungültige Operation Ungültige Operation
FYL2XP1	+ – ∞ x log(1) + – ∞ x log(X>0) + – ∞ x log(–1<X<0) + –Y x log(+ ∞) + –0 x log(+ ∞) + –Y x log(– ∞)	Ungültige Operation (1) ∞ (2) ∞ (6) ∞ Ungültige Operation Ungültige Operation

X Null oder positiver Operand
Y Positiver Operand (nicht Null)
(1) Vorzeichen des originalen infiniten Operanden
(2) Komplement des Vorzeichens des originalen infiniten Operanden
(3) Exclusiv-Oder-Verknüpfung der Operanden-Vorzeichen
(4) Vorzeichen des originalen Operanden
(5) Vorzeichen des originalen Null-Operanden
(6) Vorzeichen des originalen Y-Operanden

8.5.8 Behandlung von Null-Operanden

Null-Operanden kann der Programmierer selbst codieren oder sie werden von der FPU als **maskierte** Antwort auf eine **Underflow**-Exception produziert. Die folgende Tabelle zeigt, welche Ergebnisse die Befehle liefern, wenn sie mit Null-Operanden ausgeführt werden. Sie zeigt außerdem, wie eine Null aus Nicht-Null-Operanden erzeugt wird:

Operation	Operanden	Ergebnis
FLD, FBLD	+ – 0	(1) 0
FILD	+0	+0
FST, FSTP, FRNDINT	+ – 0	(1) 0
	+X	+0 (2)
	– X	– 0 (2)
FBSTP	+ – 0	(1) 0
FIST; FISTP	+ – 0	(1) 0
	+X	+0 (3)
	– X	– 0 (4)
FCHS	+0	– 0
	– 0	+0
FABS	+ – 0	+0
Addition	+0 plus +0	+0
	– 0 plus – 0	– 0
	+0 plus – 0	+ – 0 (5)
	– 0 plus +0	+ – 0 (5)
	– X plus +X	+ – 0 (5)
	+X plus – X	+ – 0 (5)
	+ – 0 plus + – X	(6) X
	+ – X plus + – 0	(6) X
Subtraktion	+0 minus – 0	+0
	– 0 minus +0	– 0
	+0 minus +0	+ – 0 (5)
	– 0 minus – 0	+ – 0 (5)
	+X minus +X	+ – 0 (5)
	– X minus – X	+ – 0 (5)
	+ – 0 minus + – X	(7) X
	+ – X minus + – 0	(6) X
Multiplikation	+ – 0 mal + – 0	(8) 0
	+ – 0 mal + – X	(8) 0
	+ – X mal + – 0	(8) 0
	+X mal +Y	+0 (2)
	– X mal – Y	+0 (2)
	+X mal – Y	– 0 (2)
	– X mal +Y	– 0 (2)

Operation	Operanden	Ergebnis
Division	+–0 div +–0	Ungültige Operation
	+–X div +–0	(8) ∞ (Division durch 0)
	+–X div +– ∞	(8) 0
	+0 div +X	+0
	–0 div –X	+0
	+0 div –X	–0
	–0 div +X	–0
	–X div –Y	+0 (2)
	+X div +Y	+0 (2)
	–X div +Y	–0 (2)
	+X div –Y	–0 (2)
FPREM, FPREM1	+–0 rem +–0	Ungültige Operation
	+–X rem +–0	Ungültige Operation
	+–0 rem +–X	+0
	–0 rem +–X	–0
	+X rem +–Y	+0 (wenn X/Y exakt)
	–X rem +–Y	–0 (wenn X/Y exakt)
FSQRT	+–0	(1) 0
Compare	+–0 comp +X	+–0 < +X
	+–0 comp +–0	+–0 = +–0
	+–0 comp –X	+–0 > –X
FTST	+–0	+–0 = 0
FXAM	+0	C3=1, C2=C1=C0=0
	–0	C3=C1=1, C2=C0=0
FXTRACT	+0	Division durch Null ST(0) = +0, ST(1) = –∞
	–0	Division durch Null ST(0) = –0, ST(1) = –∞
FPTAN	+–0	(1) 0
FSIN (oder SIN als Teilergebnis von FSINCOS)	+–0	(1) 0
FCOS (oder COS als Teilergebnis von FSINCOS)	+–0	+1
FPATAN	arctan(+–0/+X)	(1) 0
	arctan(+–0/–X)	(1) π
	arctan(+–X/+–0)	(6) π/2
	arctan(+–0/+0)	(1) 0
	arctan(+–0/–0)	(1) π
	arctan(+∞/+–0)	+ π/2
	arctan(–∞/+–0)	– π/2
	arctan(+–0/+∞)	(1) 0
	arctan(+–0/–∞)	(1) π

Operation	Operanden	Ergebnis
F2XM1	+0 – 0	+0 – 0
FYL2X	+ – Y x log(+ – 0) + – 0 x log(+ – 0)	Division durch Null Ungültige Operation
FYL2XP1	+Y x log(+ – 0+1) – Y x log(+ – 0+1)	(1) 0 – (1)0
FSCALE	+ – 0 skaliert mit – ∞ + – 0 skaliert mit + ∞ + – 0 skaliert mit X	(1) 0 Ungültige Operation (1) 0

X und **Y**Positive Operanden

(**1**) Vorzeichen des Originalen Null-Operanden

(**2**) Bei extremen Underflow wird das Ergebnis nach Null denormalisiert

(**3**) Wenn 0 < X < 1 ist und **nicht** die Rundungsart „Aufrunden in Richtung +Unendlich" gewählt ist

(**4**) Wenn –1 < X < 0 ist und **nicht** die Rundungsart „Abrunden in Richtung –Unendlich" gewählt ist.

(**5**) Das Vorzeichen wird durch die Rundungsart bestimmt:

+ bei

- Runden zum nähesten Wert, Aufrunden in Richtung +Unendlich oder Abschneiden (Chop)

– bei

- Abrunden in Richtung in Richtung –Unendlich

(**6**) Vorzeichen des originalen X-Operanden

(**7**) Komplement des Vorzeichens des originalen X-Operanden

(**8**) Exclusiv-Oder-Verknüpfung der Operanden-Vorzeichen

8.5.9 Rangordnung der Exceptions

Die FPU behandelt die Exceptions nach einer vordefinierten Rangordnug. Dabei werden höherpriorisierte Exceptions angezeigt und gemäß ihrer Erfordernisse passende Ergebnisse geliefert. Dagegen werden niedriger-priorisierte Exceptions **nicht** angezeigt, selbst wenn sie vorkommen. So führt z.B. die Division SNaN/0 zur Exception „Ungültige Operation" (wegen SNaN) und **nicht** zur Exception „Division durch Null".

Die Rangordnung der Exceptions ist folgende:

1. Ungültige Operation, eingeteilt in
   a. Numerik-Stack-Underflow (ein Befehl hat versucht, ein **leeres** Numerik-Stack-Register zu lesen),
   b. Numerik-Stack-Overflow (ein Befehl hat versucht, in ein **nicht-leeres** Numerik-Stack-Register zu schreiben),
   c. nicht-unterstütztes Operandenformat,
   d. SNaN-Operand.

2. QNaN-Operand
   Ein solcher Operand führt normalerweise zu **keiner** Exception. Wird aber ein QNaN

in einer Operation benützt, dann hat sie Vorrang vor den niedriger-priorisierten Exceptions. Was bedeutet das? Wenn z.B. ein QNaN durch Null dividiert wird, ist das Ergebnis ein QNaN und führt **nicht** zur niedrigerpriorisierten Exception ‚Division durch Null'.

3. Division durch Null und andere ungültige Operationen, die oben nicht erwähnt sind.
4. Denormal-Operand
   Ist diese Exception maskiert, dann wird der betreffende Befehl ungehindert ausgeführt. Das Denormal kann aber eine niedriger-priorisierte Exception verursachen.
5. Numerik-Overflow/Underflow
   Zu diesen Exceptions kommt es, wenn nach einer Operation der Exponent des Ergebnisses **zu groß** (Overflow) oder **zu klein** (Underflow) ist, um im gewünschten Real-Format dargestellt werden zu können. Ist das Ergebnis außerdem noch ungenau, dann wird zusätzlich die Exception ‚Precision' angezeigt.
6. Ungenaues Ergebnis (Precision).

8.5.10 Initialisierung der FPU

Nach einem Hardware-**RESET** sollte die FPU in einen definierten Anfangszustand gebracht werden. Diese Aufgabe erledigt der Befehl F(N)INIT, der die FPU in Übereinstimmung mit der folgenden Tabelle initialisiert:

FPU-Status nach der Initialisierung:

Feld	Wert	Interpretation
Kontrollwort:	037FH	
(Infinity Control)	0	Affines Modell
Rounding Control RC	00	Runden zum nähesten Wert
Precision Control	11	64 Bits Signifikanden-Genauigkeit
Masken-Bits		
PM/UM/OM/ZM/DM/IM	111111	Alle Exceptions sind maskiert
Statuswort:	0000H	
(Busy)	0	–
Condition-Code		
C3,C2,C1,C0	0000	–
Stack-Top ST	000	Stack-Register R0 ist Stack-Top
Error-Summary ES	0	Keine Exceptions
Stack-Fault	0	Kein Numerik-Stack-Overflow/Underflow
Exception-Flags		
PE/UE/OE/ZE/DE/IE	000000	Keine Exceptions
Tagwort:	FFFFH	
Tags	11	Leer
Stack-Register R0..R7		Unverändert
Exception Pointer:		
Befehlscode		Gelöscht
Befehlsadresse		Gelöscht
Operanden-Adresse		Gelöscht

8.5.11 Synchronisation von Exceptions

Die Befehle des 80486 werden von zwei separaten Einheiten ausgeführt:

- Von der **I**nteger **U**nit (**IU**) und
- von der **F**loating-**P**oint-**U**nit (**FPU**)

Wegen dieser Aufteilung kann die FPU ihre Numerik-Befehle **parallel** zu den Integer-Befehlen der IU ausführen. Dabei ist **keine** spezielle Programmiertechnik notwendig. Die Numerik-Befehle der FPU werden einfach in den Befehlsstrom der IU platziert. Weil aber FPU-Operationen im allgemeinen mehr Zeit beanspruchen als IU-Operationen, kann die IU während der Ausführungszeit eines Numerik-Befehls oft noch mehrere eigene Befehle ausführen. Die Synchronisation der FPU- und IU-Befehle erfolgt dabei automatisch durch den Prozessor. Der spezielle Synchronisations-Code (F)WAIT = 9BH **vor** den Floating-Point-Befehlen ist im Gegensatz zu früheren Numerik-Prozessoren jetzt **nicht** mehr notwendig und wird von ASM386/486 auch nicht generiert. Allerdings wird (F)WAIT immer noch benutzt und zwar zur Synchronisation von **unmaskierten** Exceptions. Unter dem Aspekt der Synchronisation läßt sich der FPU-Befehlssatz in **drei** Klassen einteilen:

- Nicht-Steuerbefehle
- Steuerbefehle der WAIT-Form und
- Steuerbefehle der Nicht-WAIT-Form

Im folgenden Assembler-Programmfragment ist jeweils ein Mitglied dieser Befehlsklassen enthalten:

```
LOC       OBJ        LINE    SOURCE
                       1              NAME   synch
                       2
--------               3    CODE32 SEGMENT ER USE32
                       4
00000000 D9FA          5          .   fsqrt          ;Nicht-Steuerbefehl
00000002 9BDFE0        6          .   fstsw ax       ;Steuerbefehl der WAIT-Form
00000005 DFE0          7          .   fnstsw ax      ;Steuerbefehl der Nicht-WAIT-Form
                       8
--------               9    code32 ENDS
                      10              END
ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS.
```

1, Zur Klasse der **Nicht-Steuer**befehle gehört fsqrt (square root). ASM386/486 generiert hier **keinen** WAIT-Code.

2, Zur Klasse der Steuerbefehle in der **Nicht-WAIT**-Form gehört fnstsw ax (Speichere Statuswort, n bedeutet no-wait). Auch hier generiert ASM386/486 **keinen** WAIT-Code.

3, Zur Klasse der Steuerbefehle in der **WAIT**-Form gehört fstsw ax. Hier generiert ASM386/486 den WAIT-Code 9BH.

Immer wenn eine **unmaskierte** Numerik-Exception auftritt, dann stoppt die FPU die weitere Ausführung des betreffenden Befehls und gibt ein internes Fehlersignal an den Prozessor. Dieses Signal quittiert der Prozessor beim **nächsten** WAIT-Befehl (Steuerbefehl

der WAIT-Form; fstsw ax...) oder beim **nächsten** Nicht-Steuerbefehl (fsqrt...) und verzweigt zum Software-Exception-Handler.

> **Hinweis:**
> In Steuerbefehlen der Nicht-WAIT-Form (fnstsw ax...) werden **keine** unmaskierten Exceptions quittiert.

Betrachten wir als Beispiel für eine **unkorrekte Exception-Synchronisation** den folgenden Programm-Code:

```
LOC       OBJ                 LINE   SOURCE
                                1            NAME    unkorrekt_synch
                                2
                                3    ASSUME DS:DATA
                                4
--------                        5    CODE32 SEGMENT ER USE32
                                6
00000000  DF0500000000    R     7            fild    zaehler     ;ST(0) <— Integervariable
00000006  66FF0500000000  R     8            inc     zaehler     ;zaehler=zaehler+1
0000000D  D9FA                  9            fsqrt               ;Wurzel aus ST(0)
                               10
--------                       11    code32 ENDS
                               12
--------                       13    DATA   SEGMENT RW USE32
00000000  ????                 14           zaehler dw   ?
--------                       15    DATA   ENDS
                               16           END

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS.
```

Hier wird zunächst die Integer-Variable **zaehler** in das Numerik-Top-Stackregister geladen, anschließend die Integer-Variable inkrementiert und danach die Wurzel der originalen Integer-Variablen berechnet. Wenn der FILD-Befehl **keine** Exception verursacht, dann wird das Programm korrekt ausgeführt. Die Situation ändert sich aber, wenn FILD versucht, die Integer-Variable in ein **nicht-leeres** Numerik-Stackregister zu laden. Es kommt dann zu einem **Numerik-Stack-Overflow** (SF=1) und das IE-Bit (Invalid Operation) wird gesetzt (IE=1). Ist diese Exception **nicht maskiert** (IM=0 im Kontrollwort), dann wird sie erst beim Nicht-Steuerbefehl **fsqrt** entdeckt und quittiert. Bevor aber an dieser Stelle der Exception-Handler aufgerufen wird, hat bereits der **inc**-Befehl die Integer-Variable zaehler inkrementiert. Die Folge ist, daß der Exception-Handler einen **falschen** zaehler-Wert bekommt.

Wie ist dieses Problem zu lösen? Im folgenden Programm-Code ist zu sehen, daß die beiden Befehle **inc zaehler** und **fsqrt** vertauscht sind.

```
LOC       OBJ                 LINE   SOURCE
                                1            NAME    korrekt_synch
                                2
                                3    ASSUME DS:DATA
                                4
--------                        5    CODE32 SEGMENT ER USE32
                                6
00000000  DF0500000000    R     7            fild    zaehler     ;ST(0) <— Integervariable
00000006  D9FA                  8            fsqrt               ;Wurzel aus ST(0)
00000008  66FF0500000000  R     9            inc     zaehler     ;zaehler=zaehler+1
                               10
```

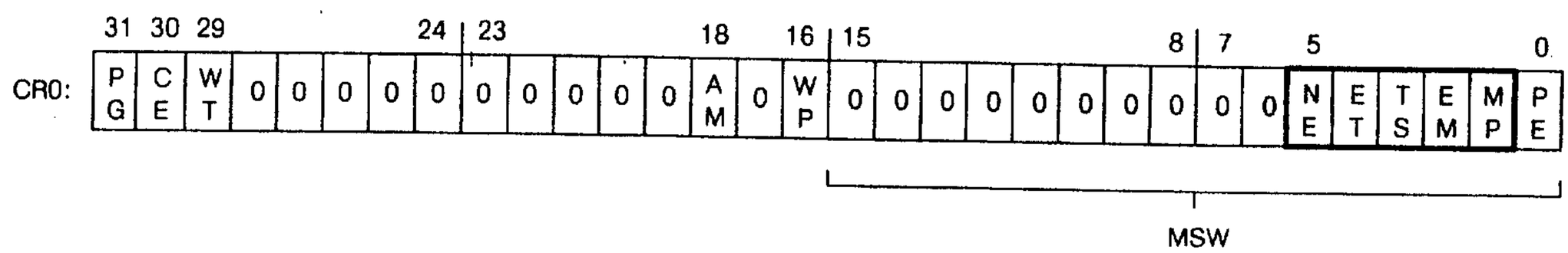
```
--------                    11    code32   ENDS
                            12
--------                    13    DATA     SEGMENT RW USE32
00000000  ????              14    zaehler dw        ?
                            15    DATA     ENDS
--------                    16             END

ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS.
```

Wenn jetzt der FILD-Befehl eine Invalid-Operation-Exception verursacht, dann wird der Exception-Handler bereits beim unmittelbar folgenden Nicht-Steuerbefehl fsqrt aufgerufen. Der IU-Befehl inc zaehler wird daher **nicht** mehr ausgeführt und der Exception-Handler bekommt jetzt den **richtigen** zaehler-Wert. Der obige Programm-Code ist also ein Beispiel für eine **korrekte Exception-Synchronisation.**

8.6 Numerik-Umgebung konfigurieren

Die Numerik-Umgebung konfiguriert die System-Software. Zu diesem Zweck lädt sie das **ET**-Bit (Prozessor **E**x**T**ension), das **EM**-Bit (**EM**ulate Coprozessor), das **EN**-Bit (**N**umerik **E**xception) und das **MP**-Bit (**M**onitor **C**oprozessor) im CR0-Register mit passenden Werten. Das MP-Bit steht dabei in enger Beziehung zum **TS**-Bit (**T**ask **S**witched). Die Positionen dieser Bits sind im folgenden CR0-Layout zu sehen:



8.6.1 Prozessor Extension-Bit ET

Dieses Bit wird nur in 80386-Systemen benutzt.

- Es wird dort gesetzt (ET=1), wenn als Coprozessor der 80387 erkannt wurde und
- es wird dort gelöscht (ET=0), wenn als Coprozessor der 80287 erkannt wurde.

Die 80486-Hardware benutzt dieses Bit **nicht**. Nach einem Hardware-RESET ist das ET-Bit gelöscht (ET=0).

8.6.2 Emulate Coprozessor-Bit EM

Dieses Bit wird ebenfalls nur in 80386-Systemen benutzt. Falls dort **kein** Numerik-Coprozesor existiert, ist es möglich die Numerik-Funktionen mit einem Software-Emulator zu emulieren.

- Wenn das EM-Bit während der Ausführung eines Numerik-Befehls gelöscht ist (EM=0), dann wird der Befehl vom Coprozessor ausgeführt.

- Wenn dagegen das EM-Bit während der Ausführung eines Numerik-Befehls gesetzt ist (EM=1), dann wird **Interrupt 7** ausgelöst (Coprozessor nicht vorhanden). Der aufgerufene Exception-Handler kann daraufhin die Numerik-Funktion emulieren.

Weil im 80486 der 80387-Coprozessor mit enthält ist, sollte das EM-Bit gelöscht sein (EM=0).

8.6.3 Numerik Exception-Bit EN

Diese Bit bestimmt in 80486-Systemen, ob **unmaskierte** Floating-Point-Exceptions vom internen Interrupt-Vektor 16 (NE=1) oder durch einen externen Interrupt (NE=0) gemeldet werden. Nach einem Hardware-RESET ist das NE-Bit = 0. Daher muß die Software dieses Bit setzen, wenn die Fehlermeldung über den internen Mechanismus erfolgen soll.

8.6.4 Monitor Coprozessor-Bit MP und Task Switched-Bit TS

- Ist während der Ausführung eines ESC- oder WAIT-Befehls das MP-Bit gesetzt (MP=1), dann prüft der Prozessor das TS-Bit. Ist es auch gesetzt (TS=1), dann hat irgendwann **vorher** ein **Taskwechsel** stattgefunden und dies bedeutet, daß die augenblickliche Task **nicht** Eigentümerin der FPU ist. Mit anderen Worten, der Context der FPU paßt nicht zur augenblicklichen Aufgabe der Prozessors. Aus diesem Grund wird **Interrupt 7** (Coprozessor nicht vorhanden) ausgelöst und der aufgerufene Exception-Handler bekommt Gelegenheit den notwendigen FPU-Context-Wechsel zu veranlassen. Danach kann er das TS-Bit mit dem privilegierten Befehl (Ebene 0) CLTS (Clear Task Switched Flag) löschen.
- Ist während der Ausführung eines ESC- oder WAIT-Befehls das MP-Bit gelöscht (MP=0), dann wird das TS-Bit **nicht** getestet.

Für den Normalbetrieb sollte daher das MP-Bit gesetzt sein (MP=1).

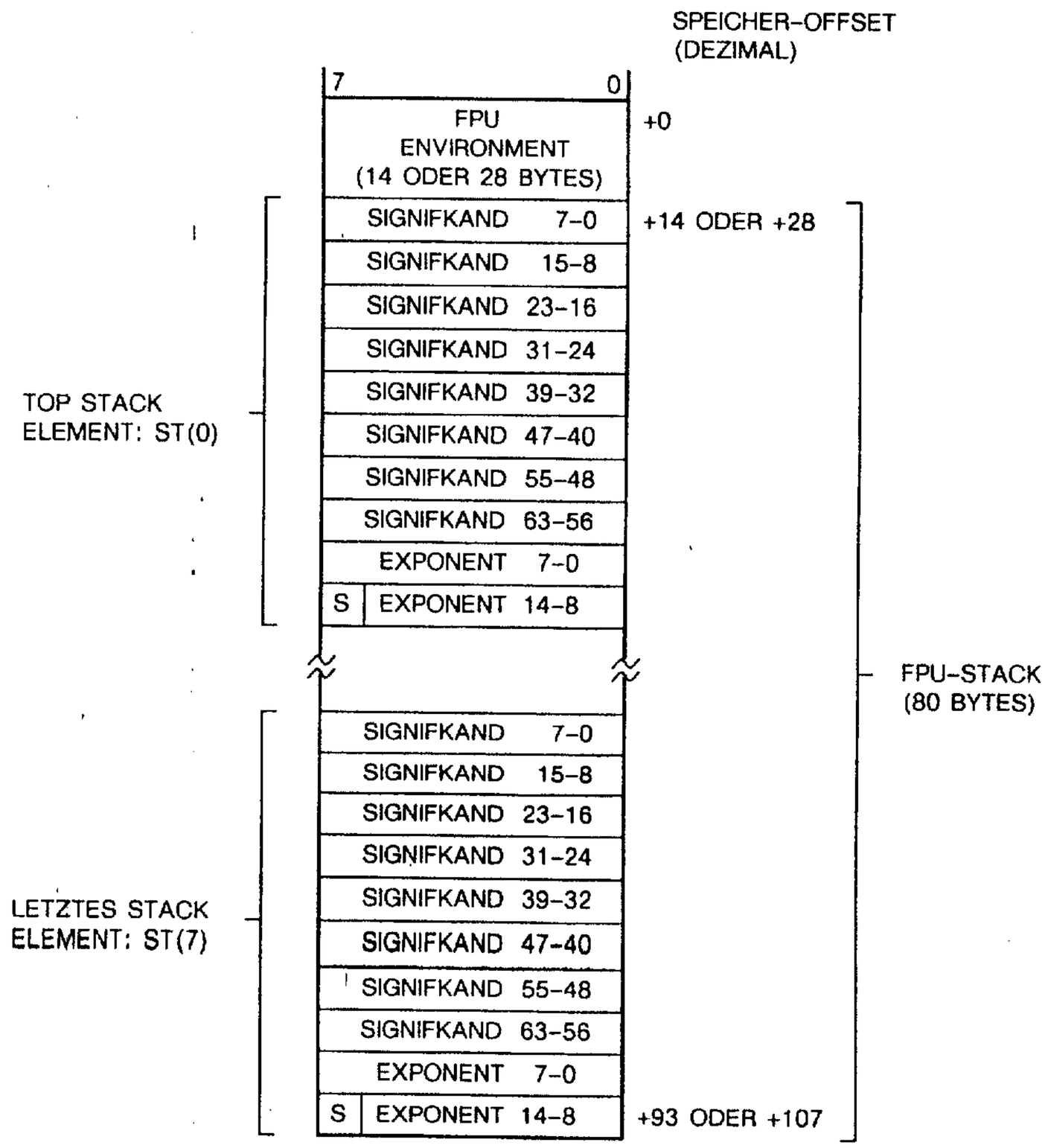
8.7 Numerik-Task-Status

Wenn eine Task die FPU benützt, hat das Betriebssystem am Task-Status zwei wesentliche Interessen:

1, Damit ein Task-Status-Segment auch die FPU-Statusinformationen aufnehmen kann, muß die 80486-Datenbasis um einige Bytes für den Numerik-Status erweitert werden. Soll der CPU/FPU-Status in einem **USE16**-Segment aufbewahrt werden, dann sind noch mindestens **94** zusätzliche Bytes erforderlich.

   Soll dagegen der CPU/FPU-Status in einem **USE32**-Segment aufbewahrt werden, müssen noch mindestens **108** zusätzliche Bytes verfügbar sein.

   Das 94- oder 108-Byte-Layout des FPU-Status ist im folgenden Bild zu sehen:



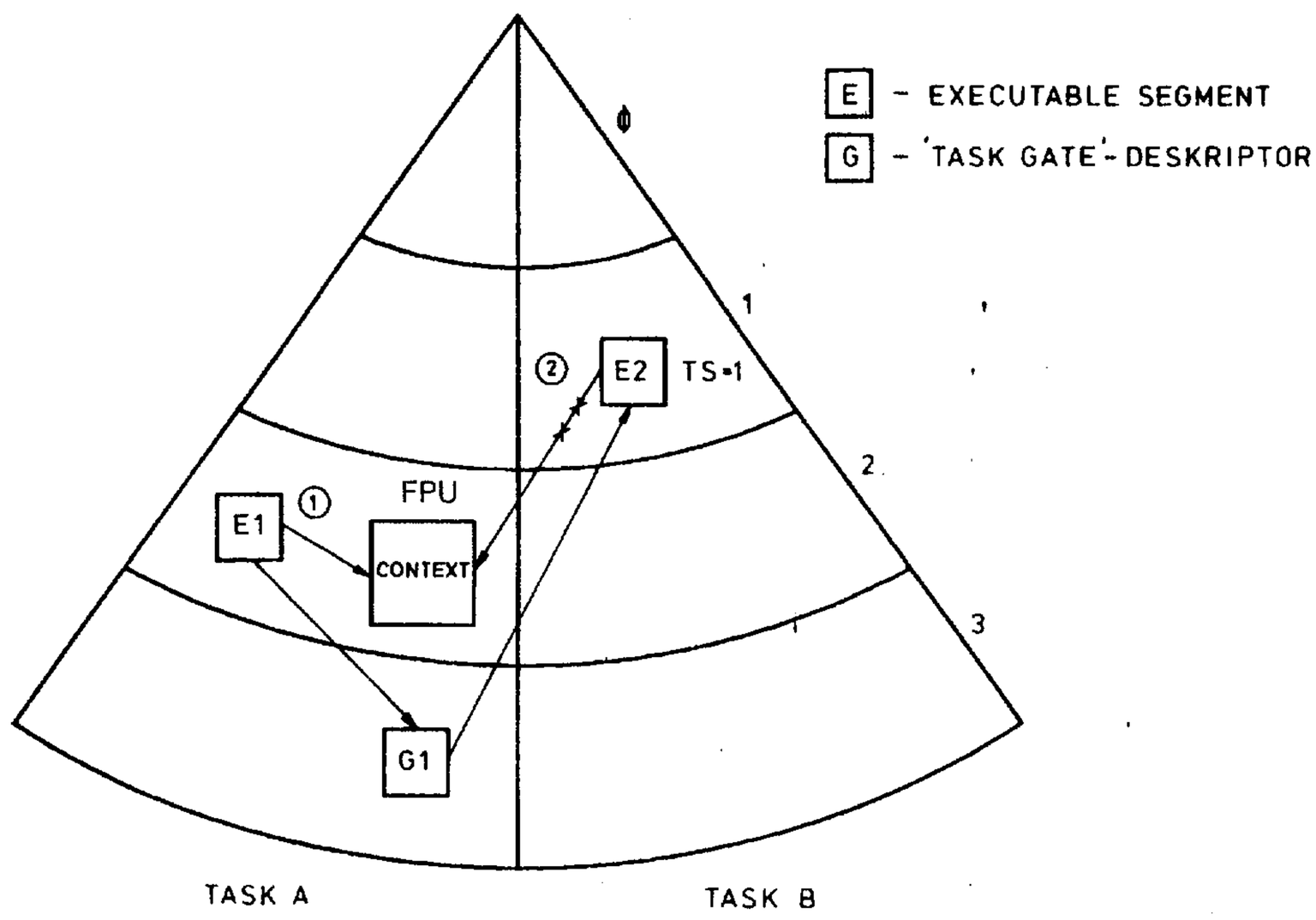
Wie zu erkennen ist, setzt sich der FPU-Status aus dem FPU-**Environment** und den acht **Numerik-Stack-Registern** ST(0)..ST(7) zusammen. Dabei kann, abhängig vom USE-Attribut, das Environment entweder in 14 oder in 28 Bytes aufbewahrt sein.

2, Das Betriebssystem muß den FPU-Status ändern, wenn eine andere Task versucht, die Floating-Point-Einheit zu benützen. Da typischerweise nur einige Tasks in einem System die FPU verwenden, wäre es verschwenderisch, die Status-Informationen bei jedem Task-Wechsel zu retten und zu aktualisieren. Daher ist es vorteilhafter, wenn das Betriebssystem feststellt, welche Task die FPU benutzt und die Status-Informationen nur dann tauscht, wenn eine andere Task die FPU benutzen will. Hier hilft das **TS**-Flag. Es wird vom Prozessor automatisch bei jedem Task-Wechsel gesetzt. Trifft der Prozessor in der neuen Task auf einen ESC- oder WAIT-Befehl, wird über Interrupt 7 der Exception-Handler im Betriebssystem aufgerufen. Dieser kann nun die Task, in der sich die FPU aufhält, verfolgen und wenn notwendig, ihren Status ändern.

8.7.1 FPU-Context-Wechsel

Der Context, d. h. die Funktionseinstellung der FPU wird bei einem Task-Wechsel **nicht** geändert, es sei denn, die neue Task versucht, den Numerik-Prozessor zu benützen und benötigt eigene Funktionseinstellungen.

Das folgende Bild verdeutlicht eine solche Situation:



Es ist ersichtlich, daß E1 in Task A über den Task-Gate-Deskriptor G1 einen Wechsel nach Task B realisiert, wodurch das TS-Flag im CR0-Register der 80486-CPU gesetzt wird (TS = 1).
Da in E1 auch numerische Befehle ausgeführt worden sind, enthält die FPU nach dem Task-Wechsel einen bestimmten Context ①.

Das Segment E2 in Task B benützt ebenfalls die FPU. Weil aber der augenblickliche Context zur vorherigen Task A gehört, darf ihn E2 **nicht** benützen ②. Dies bedeutet, daß die FPU für die CPU **nicht** verfügbar ist.

Führt der Prozessor einen WAIT- oder ESC-Befehl aus und ist dabei das TS-Flag gesetzt, wird die Ausnahme „Prozessor-Erweiterung nicht vorhanden" (Interrupt 7) ausgelöst. Dadurch erkennt die CPU nach einem Task-Wechsel die Existenz der FPU.

Der Exception-Handler muß nun entscheiden, ob ein FPU-Context-Wechsel notwendig ist oder nicht. Zu diesem Zweck muß der Handler wissen, welche Task im Augenblick der „Eigentümer" der FPU ist. D.h., er muß die letzte Task kennen, damit er ein entsprechendes Kommando an die FPU ausgeben kann.

Wenn sich der Eigentümer von der augenblicklichen Task identisch ist, wurde er lediglich unterbrochen. Der Interrupt 7-Handler wird dann nur das TS-Flag löschen und zum Eigentümer zurückkehren.

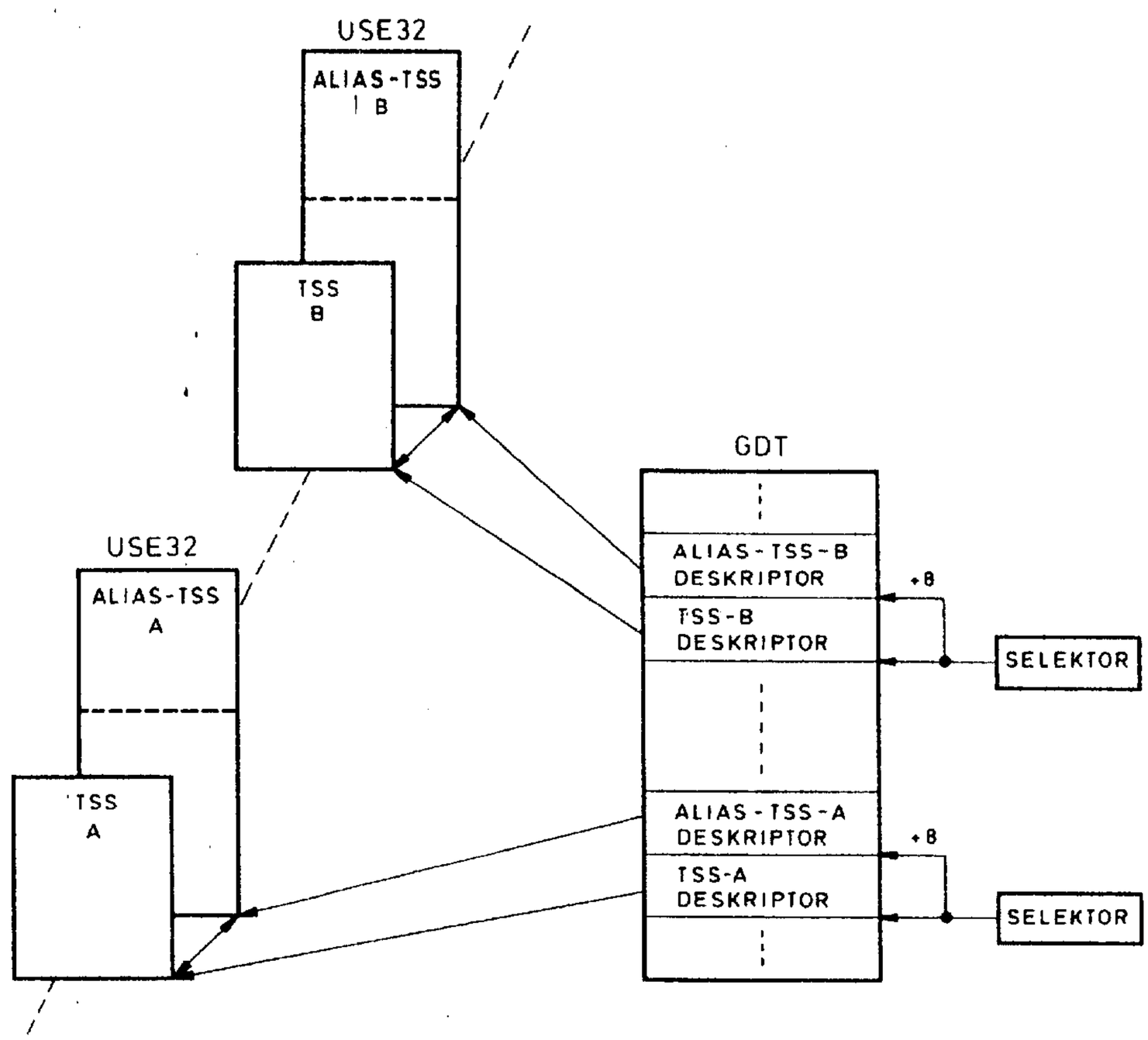
Wenn der Eigentümer mit der augenblicklichen Task unterscheidet, muß der Handler zuerst den existierenden Numerik-Context in einen sogenannten Aufbewahrungs-Bereich der **alten** Task retten (FSAVE-Befehl). Er kann dann anschließend den korrekten Numerik-Context vom Aufbewahrungs-Bereich der **augenblicklichen** Task in die FPU zurückschreiben (FRSTOR-Befehl).

Das folgende ASM386/486-Programm-Fragment enthält hierfür ein Codier-Beispiel:

8.7.1.1 **Programmierbeispiel:** Context-Wechsel

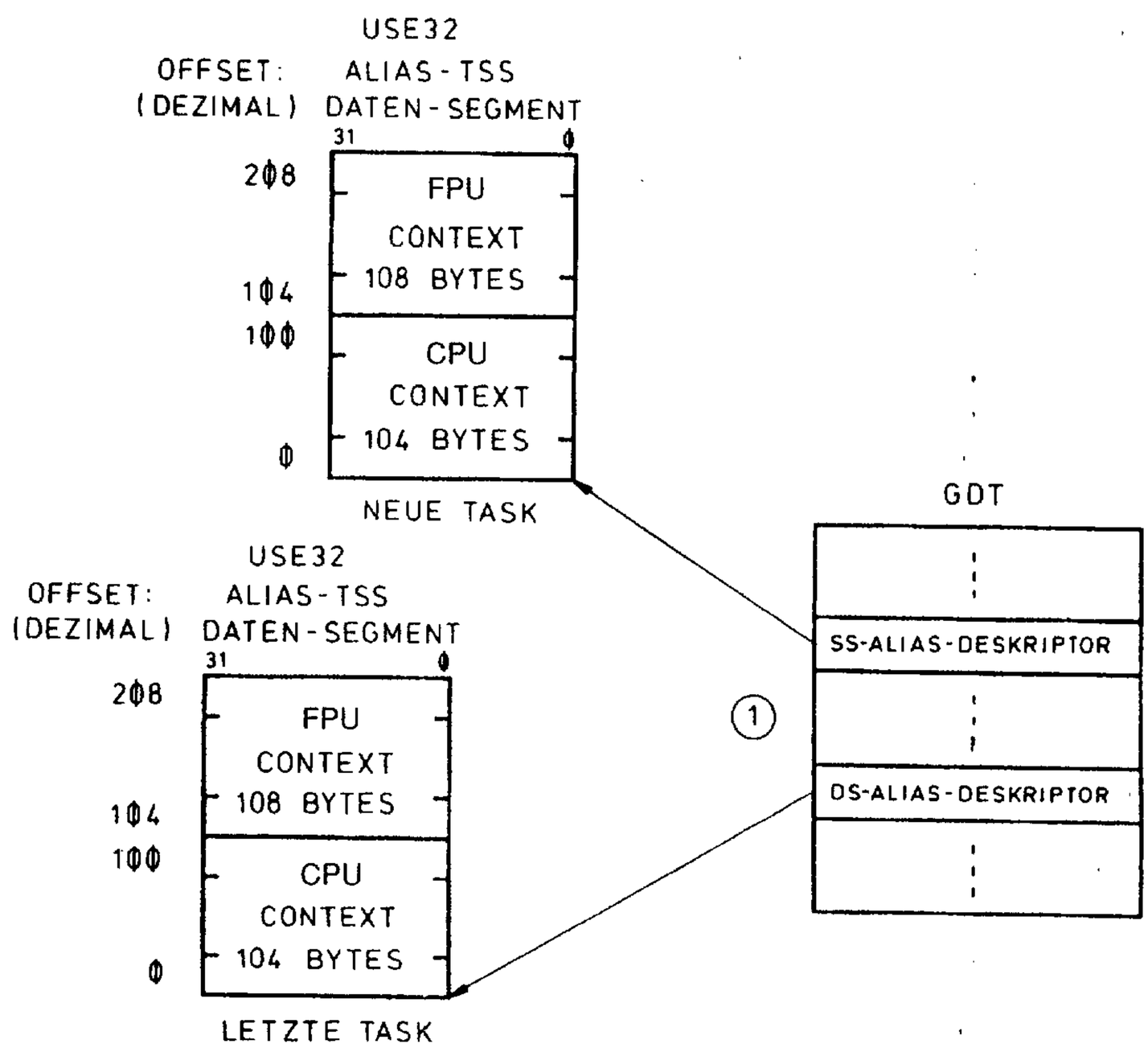
Das Programm-Fragment geht davon aus, daß in der GDT jedem TSS-Deskriptor ein Alias-Deskriptor für ein USE32-Datensegment folgt. Sie liegen daher bei Offset +8 relativ zu ihren Vorgängern. Diese Alias-TSS-Deskriptoren zeigen auf die gleichen physikalischen Regionen wie die TSS-Deskriptoren, wobei diese Regionen den Status der verschiedenen Tasks enthalten.

Das folgende Bild soll dies verdeutlichen:



Die ersten 104 Bytes der USE32-Alias-Daten-Segmente enthalten den CPU-Context, während für den FPU-Context weitere 108 Bytes überlagert (overlayed) sind.

Dies zeigt das folgende Bild:



Das Programm-Fragment geht zusätzlich davon aus, daß DS zum Alias-TSS-Datensegment der **letzten** Task und SS zum Alias-TSS-Datensegment der **neuen** Task zeigt ①.

Zu den Basisdaten des Betriebssystems gehört eine Word-Variable, die den Selektor des letzten Task-Status-Segments enthält. An dieser für den Ausnahme-Handler **externen** Variablen kann er erkennen, welche Task im Augenblick der Eigentümer der Floating-Point-Einheit ist.

```
LOC        OBJ              LINE SOURCE

                                1          NAME     context_wechsel
                                2
                                3 EXTRN letzte_FPU_task:WORD
                                4
00000068                        5 numerik_context_bereich  EQU  dword ptr 104   ;Offset des FPU-Context-
                                6                                              ;Bereichs im TSS
                                7
                                8 ASSUME DS:NOTHING,SS:NOTHING
                                9
--------                       10 interrupt_7_kern  SEGMENT   ER USE32          ;Nur ausfuehrbares Segment mit
                               11                                              ;DPL=0
                               12
00000000                       13 numerik_context_wechsel  PROC  FAR  WC(0)
                               14
00000000 50                    15          PUSH     EAX                        ;Rette Arbeits-Register
00000001 1E                    16          PUSH     DS
00000002 16                    17          PUSH     SS
```

```
00000003 66B8----      E 18        MOV     AX,SEG letzte_FPU_task
                         19                                          ;Lade AX mit dem Selektor des
                         20                                          ;Segments, in dem sich die Variable
                         21                                          ;letzte_FPU_task aufhaelt
00000007 8ED8            22        MOV     DS,AX                     ;Diesen Selektor in DS laden
00000009 0F00C8          23        STR     AX                        ;AX bekommt den Selektor des
                         24                                          ;augenblicklichen Task-Status-
                         25                                          ;Segments
0000000C 24FC            26        AND     AL,NOT 3                  ;RPL-Wert entfernen
0000000E 8ED0            27        MOV     SS,AX                     ;SS bekommt den Selektor der
                         28                                          ;augenblicklichen Task
00000010 0F06            29        CLTS                              ;Loesche TS-Flag
00000012 FA              30        CLI                               ;Interrupts sperren
00000013 663B0500000000  E 31      CMP     AX,DS:letzte_FPU_task
                         32                                          ;Pruefen, ob gleiche Task
0000001A 741B            33        JE      gleiche_task              ;Sprung, wenn gleiche Task
0000001C 66870500000000  E 34      XCHG    AX,DS:letzte_FPU_task
                         35                                          ;Die Variable
                         36                                          ;letzte_FPU_task
                         37                                          ;bekommt den Selektor der augen-
                         38                                          ;blicklichen Task
                         39                                          ;AX bekommt den Selektor der
                         40                                          ;letzten Task
00000023 66050800        41        ADD     AX,8                      ;AX zeigt zum Alias-TSS-Deskriptor
                         42                                          ;der letzten Task
00000027 8ED8            43        MOV     DS,AX                     ;Lade DS-Cache-Register mit dem
                         44                                          ;Alias-TSS-Deskriptor der letzten
                         45                                          ;Task
00000029 9BDD3568000000  46        FSAVE   DS:numerik_context_bereich ;Rette den FPU-Context der
                         47                                          ;letzten Task
00000030 36DD2568000000  48        FRSTOR  SS:numerik_context_bereich ;Lade den zur augenblicklichen
                         49                                          ;Task gehoerenden FPU-Context
00000037                 50  gleiche_task:
00000037 17              51          POP     SS
00000038 1F              52          POP     DS                      ;Zurueck in das unterbrochene
00000039 58              53          POP     EAX                     ;Programm
0000003A 66CF            54          IRET
                         55
0000003C                 56  numerik_context_wechsel ENDP
                         57
--------                 58  interrupt_7_kern  ENDS

*** WARNING #377 IN 58, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
                         59                  END

ASSEMBLY COMPLETE,   1 WARNING,   NO ERRORS.
```

Die im Programm-Fragment enthaltene Interrupt-Prozedur NUMERIK_CONTEXT_WECHSEL soll über einen „Trap Gate"-Deskriptor mit der Vektor-Nummer 7 in der IDT aufgerufen werden. Da die Interrupt-Prozedur vom Betriebssystem benützt wird, muß sie sich in der Privileg-Ebene 0 aufhalten.

Der Aufruf der Prozedur ist aber nur dann möglich, wenn die folgende Bedingung

- Interrupt-Prozedur DPL ≤Task CPL

erfüllt ist.
Genauso wie bei Hardware-Interrupts, wird auch bei ausnahmebedingten Interrupts der Vergleich Task CPL ≤ Gate DPL **nicht** durchgeführt.

Die angegebene Wechselbeziehung der Privileg-Kennummern ist folgendermaßen interpretierbar:

- Wenn der DPL-Wert der Interrupt-Prozedur 0 ist, kann die Task in jeder Privileg-Ebene durch Interrupt 7 unterbrochen werden
  (Interrupt-Prozedur DPL(0) ≤ Task CPL).

Wie das Programm-Listing zeigt, benützt die NUMERIK_CONTEXT_WECHSEL-Prozedur die Register EAX, DS und SS als Arbeits-Register und rettet sie zunächst in den Stack.

Damit nun die Prozedur prüfen kann, welche Task im Augenblick der Eigentümer der FPU ist, lädt sie zunächst mit den Befehlen

- ***MOV AX,SEG LETZTE_FPU_TASK***
- ***MOV DS,AX***

den Datensegment-Selektor der externen Word-Variablen LETZTE_FPU_TASK in das Arbeitsregister DS. Diese Variable enthält den Selektor des **letzten** Task-Status-Segments.

Bei dieser Operation ist der

- ***ASM386/486-SEG-Operator***

nützlich. Er liefert hier in bequemer Weise den Datensegment-Selektor der globalen **extern** definierten Variablen, die dem Modul nicht bekannt ist.

Der folgende geschützte Befehl

- ***STR AX***

liefert den Selektor des augenblicklichen Task-Status-Segments und speichert ihn im AX-Register.

Er hat die allgemeine Form:

- ***STR ew*** ; Store Task Register

Dabei kennzeichnet ew ein **Word**-Register oder die effektive Adresse eines **Word**-Operanden.

Der in AX enthaltene Selektor wird später in der Interrupt-Prozedur benützt, um über SS das Alias-TSS-Datensegment der augenblicklichen Task zu adressieren.

Da sich das augenblickliche Task-Status-Segment in jeder beliebigen Privileg-Ebene befinden kann, hat auch das RPL-Feld des zugehörigen Selektors irgendeinen Wert. Daher wird zunächst mit

- ***AND AL, NOT 3***

der RPL-Wert aus dem Selektor entfernt und anschließend mit

- ***MOV SS, AX***

in das SS-Register geladen.

Der folgende Befehl

- ***CLTS***

löscht das TS-Flag im CR0-Register und bewirkt, daß nach dem Verlassen der Interrupt-Prozedur die FPU ihre Arbeit aufnehmen kann.

Damit die Interrupt-Prozedur die Task-Überprüfung ungestört fortsetzen kann, sperrt sie mit

- ***CLI***

mögliche Interrupt-Anforderungen.

Welche Task im Augenblick der Eigentümer des Numerik-Prozessors ist, stellt der Befehl

- ***CMP AX,DS:LETZTE_FPU_TASK***

fest. Dabei zeigt DS:LETZTE_FPU_TASK zur externen Variablen, in der der Selektor des **letzten** Task-Status-Segments aufbewahrt ist, während sich in AX der Selektor des **augenblicklichen** Task-Status-Segments befindet.

Sollten die Selektoren **gleich** sein, gehört die FPU zur augenblicklichen Task und die Interrupt-Prozedur wird mit

- ***JE GLEICHE_TASK***

verlassen.

Im anderen Fall muß der letzte FPU-Context gerettet und der neue in die Floating-Point-Einheit geladen werden. Zu diesem Zweck werden mit

- ***XCHG AX,DS:LETZTE_FPU_TASK***

die Selektoren vertauscht. In der Variablen LETZTE_FPU_TASK ist nun der Selektor des **augenblicklichen** Task-Status-Segments aufbewahrt, während AX den Selektor des **letzten** Task-Status-Segments enthält.

Nach Ausführung der Befehle

- ***ADD AX,8***
- ***MOV DS,AX***

zeigt nun DS zum Alias-TSS-Datensegment der letzten Task, so daß mit

- ***FSAVE DS:NUMERIK_CONTEXT_BEREICH***

der FPU-Context der letzten Task gerettet werden kann.

Da SS zum Alias-TSS-Datensegment der augenblicklichen Task zeigt, kann mit

- ***FRSTOR SS:NUMERIK_CONTEXT_BEREICH***

der zur augenblicklichen Task gehörende FPU-Context in die Numerik-Einheit geladen werden.

In beiden Fällen zeigen wegen

- ***NUMERIK_CONTEXT_BEREICH EQU DWORD PTR 104***

die Offset-Komponenten der Befehle auf die FPU-Context-Bereiche (Offset = 104) der beiden Alias-TSS-Datensegmente.

Diese Context-Bereiche („Environment" und Floating-Point-Stacks) haben eine Länge von 94 Bytes oder 108 Bytes.

Dies bedeutet, daß die Befehle FSAVE bzw. FRSTOR in Übereinstimmung mit dem USE-Attribut des augenblicklichen **Code**-Segments entweder einen 94-Byte-Operanden oder einen 108-Byte-Operanden transferieren müssen.

Befinden sich FSAVE bzw. FRSTOR in einem USE16-Code-Segment, benützen sie einen 94-Byte-Operanden.

Befinden sie sich in einem USE32-Code-Segment, benützen sie einen 108-Byte-Operanden.

Im angegebenen Programmbeispiel erfolgen mit DWORD PTR 104 die Zugriffe auf die FPU-Context-Bereiche **anonym** mit wechselnden Werten in den DS- und SS-Registern. Aus diesem Grund erscheint im ASSUME-Statement das Schlüsselwort NOTHING (DS:NOTHING, SS:NOTHING).

Da der RPL-Wert im SS-Selektor entfernt worden ist und ein Daten-Zugriff über SS nur dann erlaubt ist, wenn DPL = CPL ist (im Beispiel Alias-Segment DPL = Interrupt-Prozedur CPL), müssen sich die Alias-TSS-Datensegmente in der Privileg-Ebene 0 aufhalten.