1 Basis-Programmier-Modell

Dieses Kapitel führt den Leser in Eigenschaften des 80486 ein, die ***direkt*** die Entwicklung und Implementierung von 80486-Anwender-Programmen beeinflussen.

Das Basis-Programmier-Modell bezieht sich auf die folgenden Aspekte der 80486-Architektur:

- Speicherorganisation und Segmentierung
- Datentypen
- Register
- Befehlsformat
- Auswahl der Operanden
- Interrupts und Ausnahmesituationen

1.1 Speicherorganisation und Segmentierung

Der physikalische Speicher eines 80486-Systems ist als eine Sequenz von 8-Bit-Bytes organisiert. Dabei ist jedem Byte eine eindeutige Adresse zugewiesen, die im Bereich 0 ... $2^{32} - 1$ (4 Gigabytes) liegt.

Jedes 80486-Programm ist vom physikalischen Adreßraum unabhängig. Dies bedeutet, daß Programme geschrieben werden können, ***ohne*** daß bekannt ist, wieviel physikalischer Speicher verfügbar ist und wo die Befehle und Daten positioniert sind.

Das Modell der Speicherorganisation, wie sie der Anwender-Programmierer sieht, wird vom System-Software-Entwickler vorgegeben. Dabei gibt die Architektur des 80486 dem Entwickler die Möglichkeit, für jede Task ein individuelles Modell zu wählen.

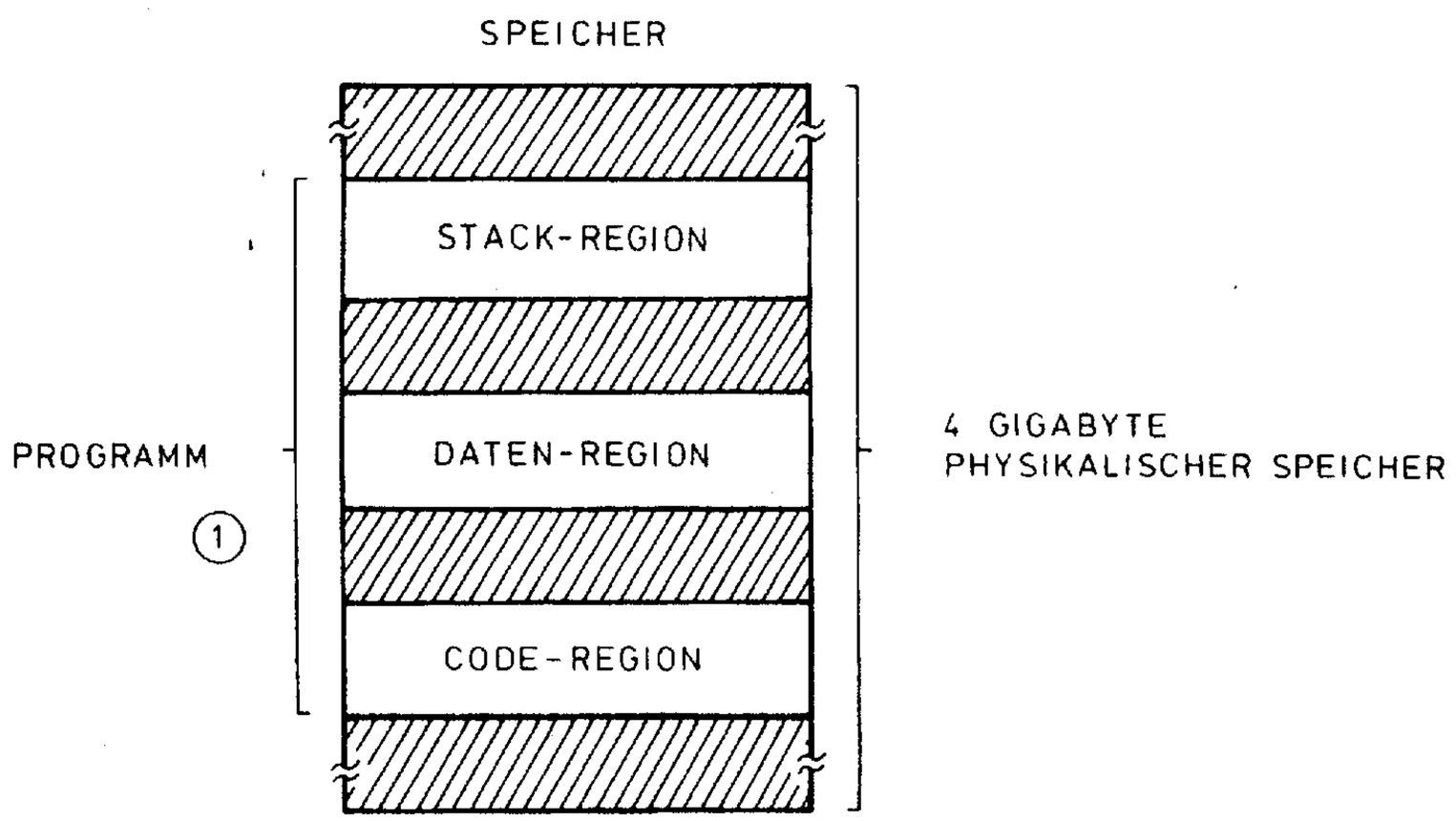
Dieses Speicherorganisations-Modell kann zwischen folgenden Extrem-Bereichen liegen:

- einem sogenannten ***flachen Adreßraum,*** der aus einem einzelnen Feld von bis zu 4 Gigabytes besteht oder
- einem ***segmentierten Adreßraum,*** der aus einer Ansammlung von bis zu 16384 unterschiedlichen Teilräumen besteht. Dabei kann jeder dieser Teilräume eine Länge von 1 Byte ... 4 Gigabytes haben.

1.1.1 Das Modell des „flachen“ Adreßraums

Aus der Sicht des Anwender-Programmierers ist ein Programm in mindestens 3 Regionen eingeteilt:

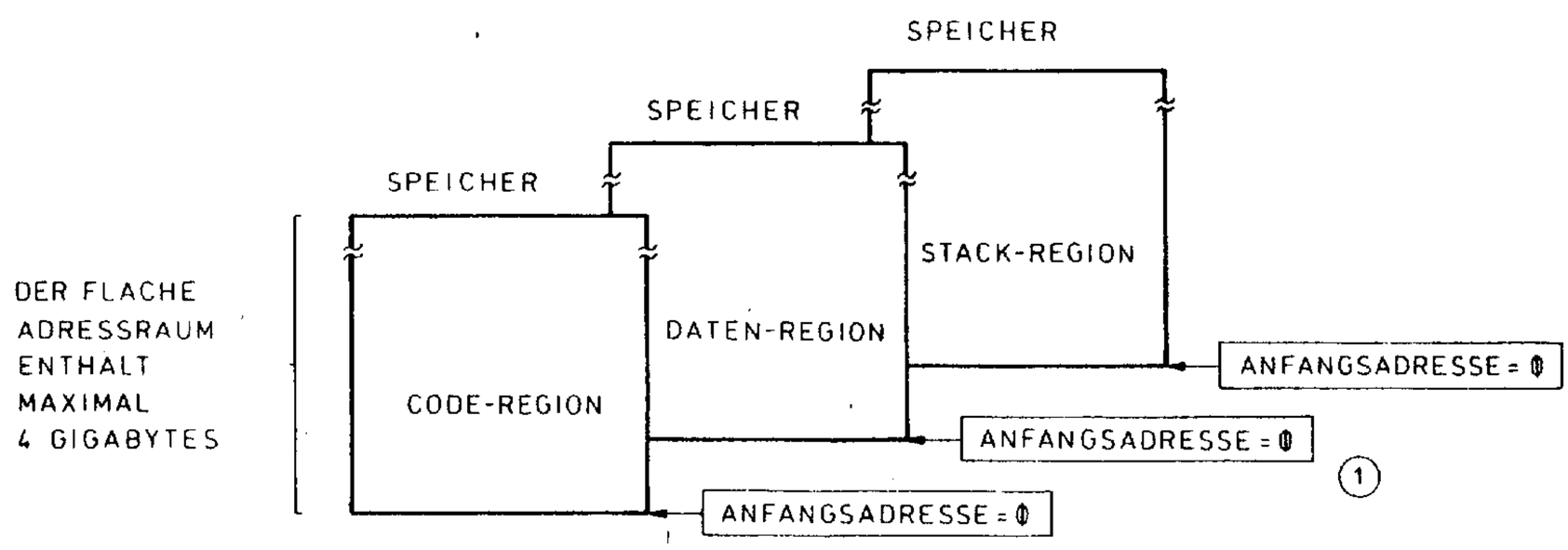
- Code-Region
- Daten-Region
- Stack-Region ①



Diese konzeptionell unterschiedlichen Teile eines Programms sollten in streng getrennten Speicherbereichen liegen, um Mischungen dieser Regionen auszuschließen.

Ist der Speicher als ***flacher Adreßraum*** organisiert, steht einer individuellen Code-, Daten- oder Stack-Region jeweils der gesamte physikalische Adreßraum von 4 Gigabytes zur Verfügung. In diesem Fall sieht der Anwender-Programmierer die spezifische Programm-Region als ein ***einzelnes*** Feld mit einer variablen Länge im Bereich 0 ... 2^{32} Bytes (4 Gigabytes).

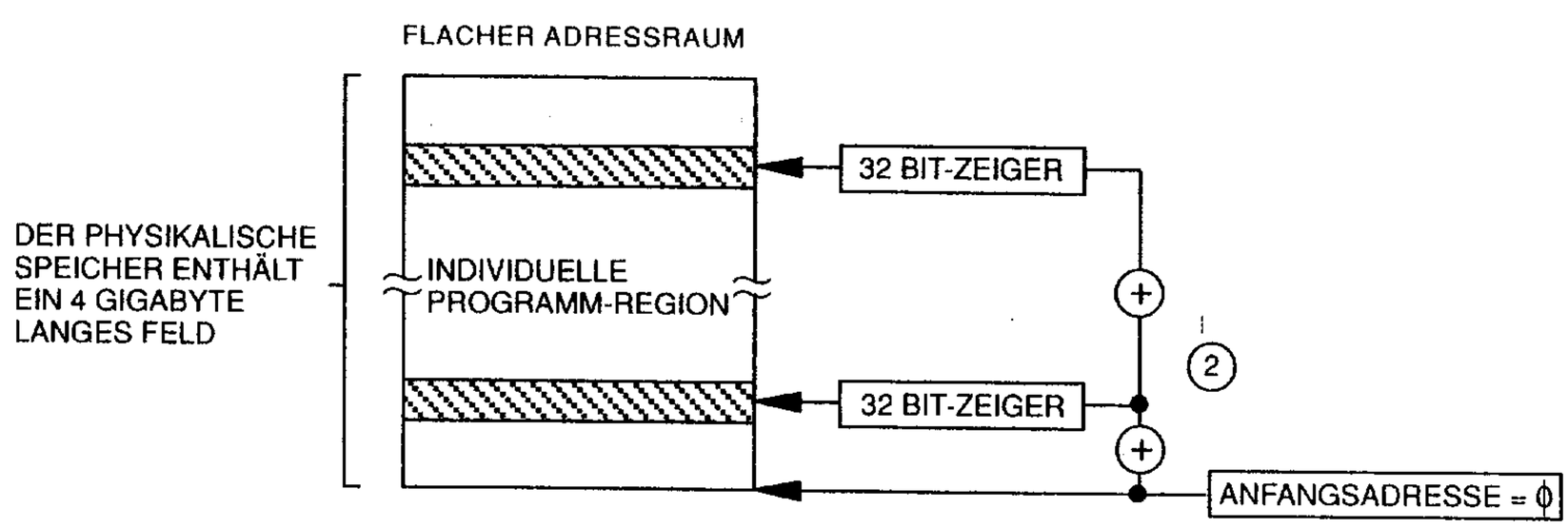
Ein solches Speicher-Organisations-Modell ist im folgenden Bild illustriert:



Das Bild zeigt, daß im flachen Adreßraum entweder die Code- oder die Daten- oder die Stack-Region eines Programms aufbewahrt ist.

Belegen die individuellen Programm-Regionen jeweils Felder mit einer maximalen Länge von 4 Gigabytes, bedeutet dies, daß ihre Anfangs-Adressen ***Null*** sein müssen ①.

Um den flachen Adreßraum zu adressieren, sind 32 Bit-Zeiger erforderlich, die mit Ordinal-Zahlen im Bereich von 0 ... $2^{32} - 1$ initialisiert sein können. Damit kann jede gewünschte Byte-Position erreicht werden ②.



1.1.2 Das Modell des segmentierten Adreßraums

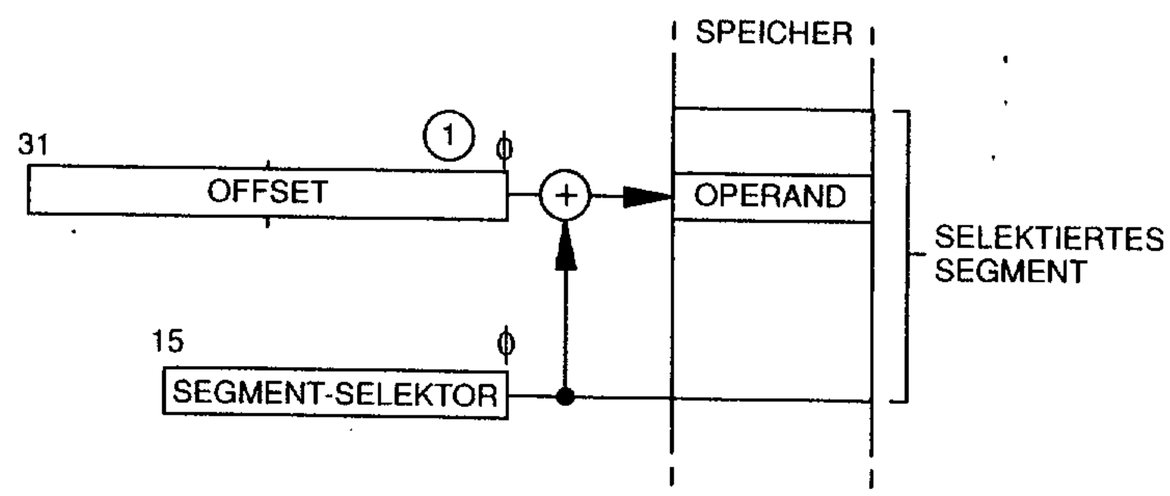
In einem segmentierten Speicher sieht ein Anwender-Programm den Adreßraum als sogenannten ***„logischen" Adreßraum.*** Dieser Raum ist viel größer als der flache Adreßraum und kann bis zu 2^{46} Bytes (64 Terabytes) enthalten.

Der Prozessor bildet den 64-Terabyte logischen Adreßraum auf den physikalischen Adreßraum von bis zu 4 Gigabytes ab, in dem er seinen Adreß-Übersetzungsmechanismus benützt.

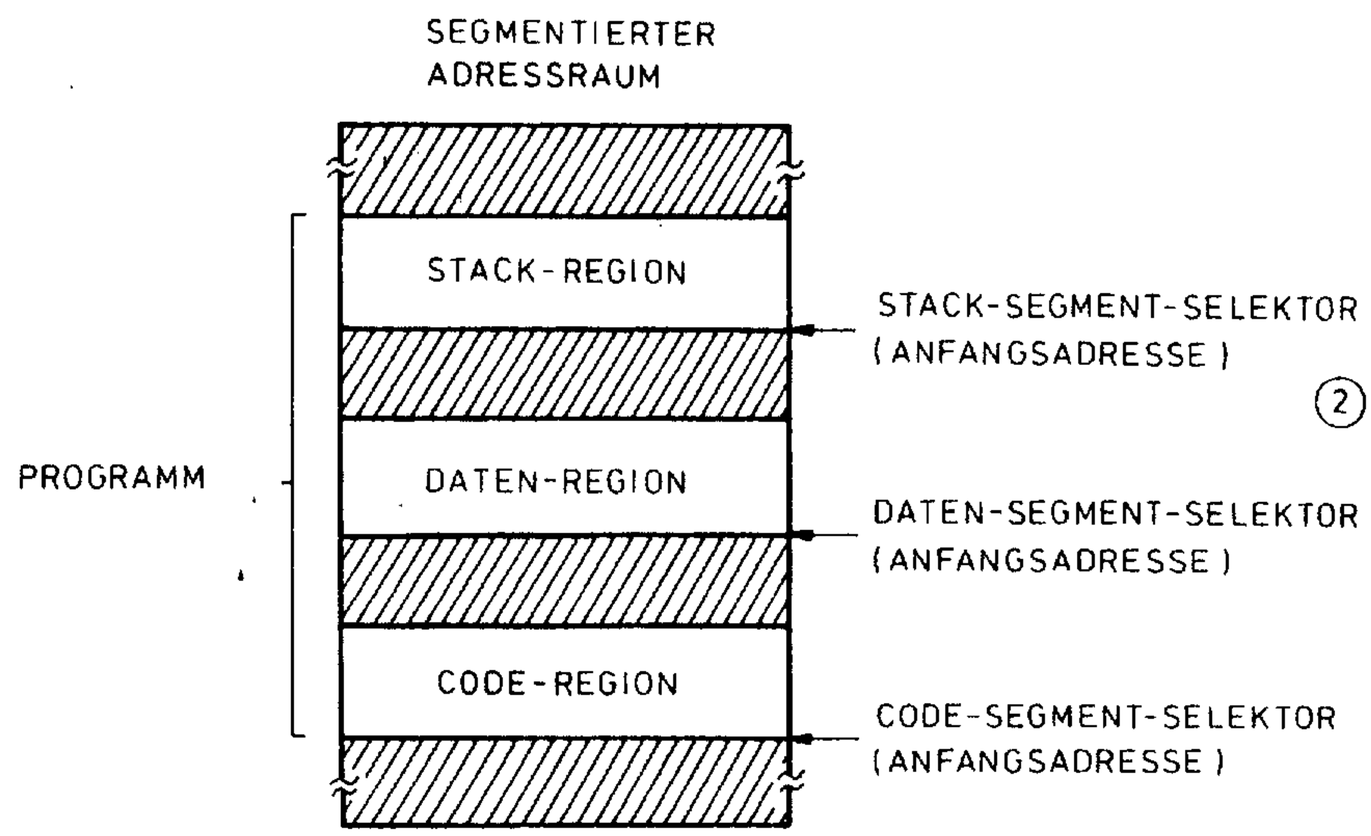
Anwender-Programmierer betrachten den logischen Adreßraum des 80486 als eine Ansammlung von bis zu 16384 unterschiedlichen Teilräumen, die jeweils individuelle Längen haben können.
Jeder dieser Teilräume wird als ***Segment*** bezeichnet, das einen individuellen „Markierer" hat. Ein Segment ist ein zusammenhängender Adreßraum mit einer variablen Länge im Bereich 0 ... 2^{32} Bytes (4 Gigabytes).

Um den logischen Adreßraum zu adressieren, sind Zeiger erforderlich, die aus folgenden zwei Komponenten bestehen:

- ***Segment-Selektor*** (Markierer), dessen 16-Bit-Felder das Segment identifiziert
- ***Offset,*** dessen 32-Bit-Ordinalwert die Byte-Position eines bestimmten Operanden innerhalb des Segments festlegt ①.



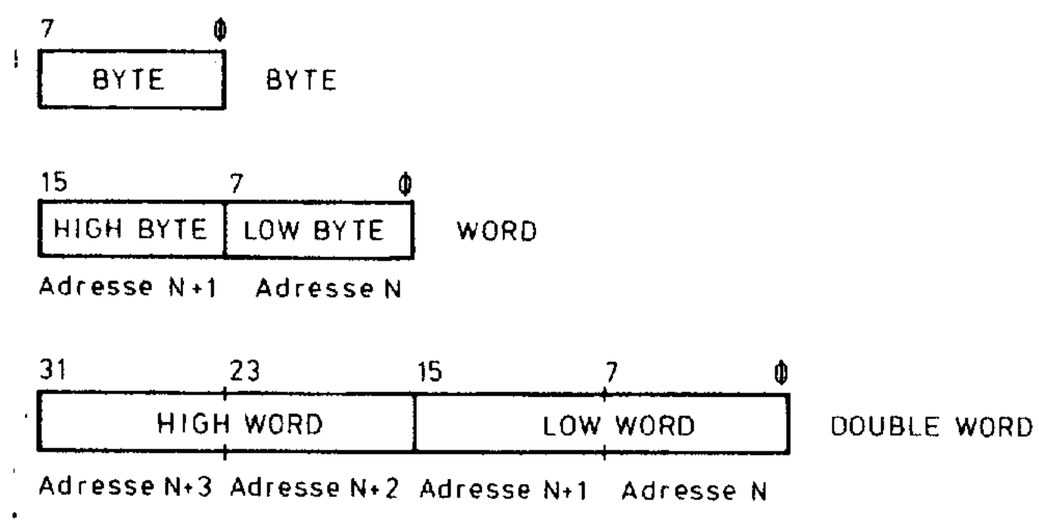
Während der Ausführung des Programms benützt der Prozessor die Segment-Selektoren, um die physikalische Anfangsadresse des benützten Segments festzulegen ②.



Die Markierung der Segmente durch individuelle Selektoren bewirkt zusätzlich eine strenge Tennung der einzelnen Programm-Regionen.

1.2 Daten-Typen

Die fundamentalen Daten-Typen des 80486 sind ***Bytes, Words, Doublewords,*** deren Aufbau das folgende Bild zeigt:



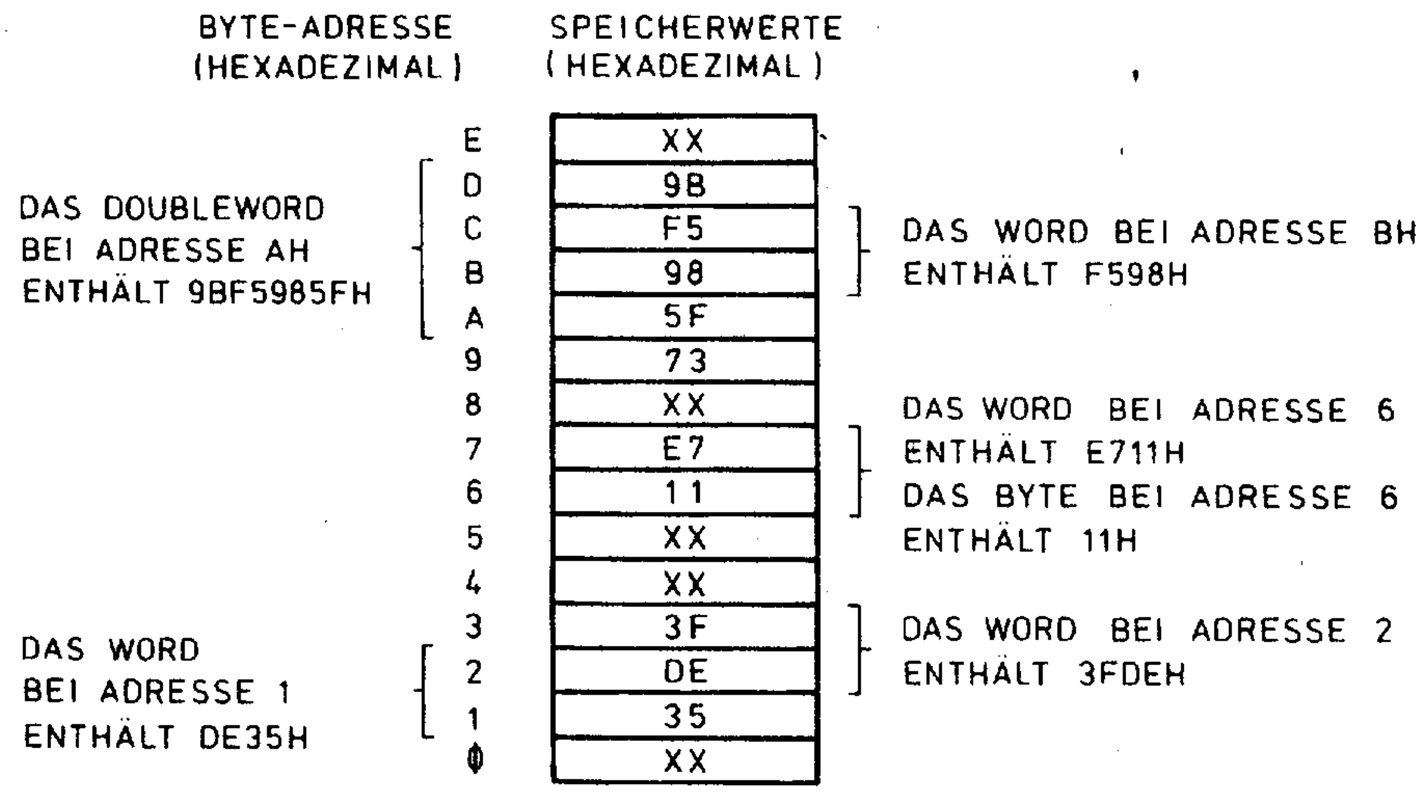
Ein ***Byte*** besteht aus 8 zusammenhängenden Bits mit den Nummern 0 ... 7 und ist an einer beliebigen logischen Adresse positioniert.

Ein ***Word*** besteht aus 2 zusammenhängenden Bytes und beginnt an einer beliebigen Byte-Adresse. Somit enthält ein Word 16 Bits (0 ... 15). Das Byte, das Bit 0 des Words enthält, wird als ***Low-Byte;*** und das Byte, das Bit 15 enthält, wird als ***High-Byte*** bezeichnet.
Jedes Byte innerhalb eines Words hat siene eigene Adresse. Dabei ist die Adresse des Bytes, das Bit 0 enthält, identisch mit der Adresse des Words.

Ein ***Doubleword*** besteht aus 2 zusammenhängenden Words und beginnt an einer beliebigen Byte-Adresse. Somit enthält ein Doubleword 32 Bits (0 ... 31). Das Word, das

Bit 0 des Doubleword enthält, wird als ***Low-Word;*** und das Word, das Bit 31 enthält, wird als ***High-Word*** bezeichnet.
Jedes Byte innerhalb eines Doublewords hat seine eigene Adresse. Dabei ist die Adresse des Bytes, das Bit 0 enthält, identisch mit der Adresse des Doublewords.

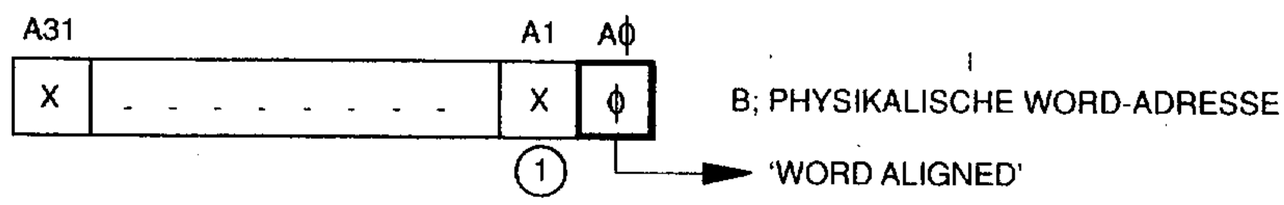
Das folgende Bild zeigt Bytes, Words und Doublewords im Speicher und illustriert die Anordnung von Bytes innerhalb von Words und Doublewords.



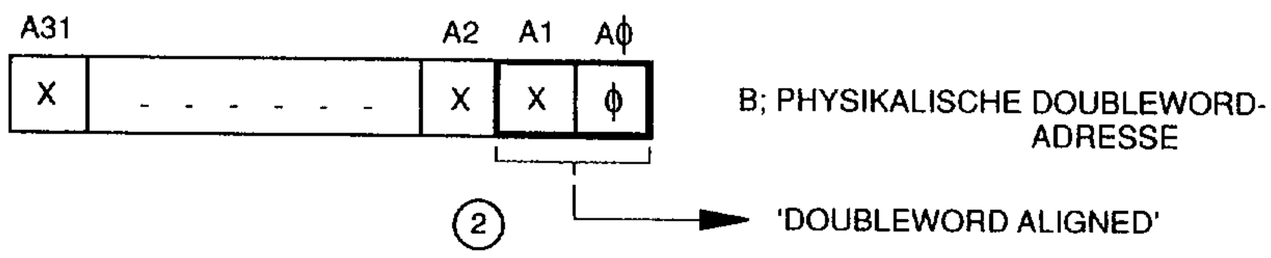
Es ist ersichtlich, daß die angegebenen fundamentalen Datentypen an jeder beliebigen Byte-Adresse beginnen können.

Dies bedeutet, daß Words und Doublewords ***nicht*** an ihren natürlichen Adreßgrenzen ausgerichtet sein müssen.

Für ***Words*** sind dies alle Adressen, die durch 2 dividierbar sind (Adreßbit A0 = 0) ①. Liegen Word-Operanden an solchen Adressen, werden sie als „Word aligned" bezeichnet.



Für ***Doublewords*** sind dies alle Adressen, die durch 4 dividierbar sind (Adreß-Bits A0 = 0, A1 = 0) ②. Liegen Doubleword Operanden an solchen Adressen, werden sie als „Doubleword aligned" bezeichnet.

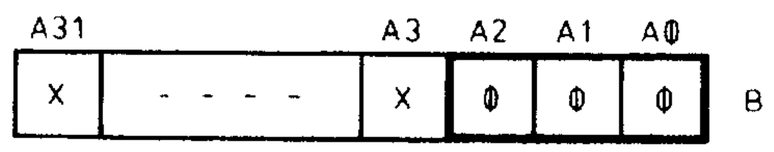


Durch die Möglichkeit, Words und Doublewords an beliebige Byte-Adressen zu positionieren, wird ein Maximum an Flexibilität in Datenstrukturen erreicht. Denn solche Datenstrukturen, die auch als ***„Records"*** bekannt sind, enthalten naturgemäß eine Mischung von Bytes, Words und Doublewords.

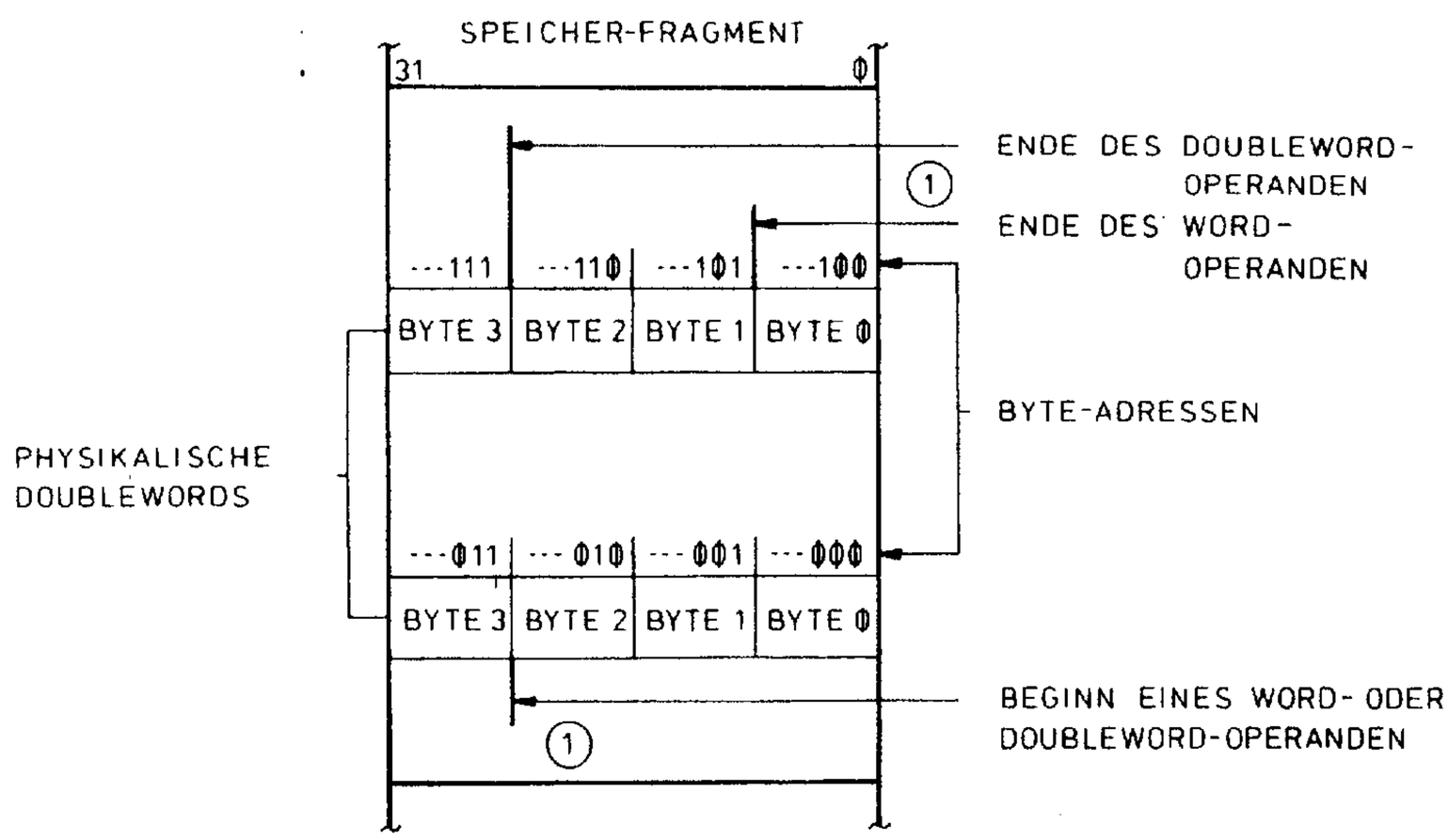
Wenn eine Hardware-Konfiguration mit einem 32-Bit-Datenbus benützt wird, werden die Operanden in ***einem*** oder in ***mehreren Transfer-Zyklen*** zwischen der CPU und dem Speicher bzw. I/O-Ressourcen übertragen.
Die Anzahl der Transfer-Zyklen ist dabei abhängig

- vom Typ des Operanden (Byte, Word oder Doubleword)
- von der physikalischen Position des Operanden im Speicher- bzw. I/O-Raum.

Das folgende Bild und die nachstehende Tabelle sollen dies verdeutlichen. Das Bild illustriert ein 32-Bit Speicher-Fragment, in dem ab folgender physikalischen Basis Adresse



an beliebigen Byte-Adressen Byte-, Word und Doubleword-Operanden hinterlegt sind.



Immer dann, wenn ein Word- oder Doubleword-Operand auf 2 aufeinanderfolgende physikalische Doublewords aufgeteilt ist, hat dies einen direkten Einfluß auf die Anzahl der erforderlichen ***Transfer-Zyklen***. Eine solche Aufteilung ist im Bild angegeben ①.

Wieviele Transfer-Zyklen, abhängig vom Operanden-Typ und seiner Byte-Adresse, erforderlich sind, ist in der folgenden Tabelle zusammengefaßt:

Typ des Operanden	Byte	Word				Doubleword			
Byte-Länge des Operanden	1	2				4			
physikalische Byte-Adresse A1/A0	XX	00	01	10	11	00	01	10	11
Transfer-Zyklen über den 32-Bit Datenbus	B	W	W	W	HB LB	D	HB L3	HW LW	H3 LB

Schlüssel: B = 1-Byte-Transfer
W = 1-Word-Transfer
H = Höherwertiger Teil
L = Niederwertiger Teil
D = 1-Doubleword-Transfer
3 = 3-Byte-Transfer

Wie die Tabelle zeigt, ist nur ***1 Transfer-Zyklus*** (B oder W) erforderlich, wenn sich der Operand innerhalb eines physikalischen Doublewords aufhält. Ist der Operand auf 2 benachbarte physikalische Doublewords aufgeteilt, sind ***2 Transfer-Zyklen*** erforderlich. Dabei werden als ***erstes*** die höherwertigen Bytes des Operanden (HB, HW oder H3) übertragen.

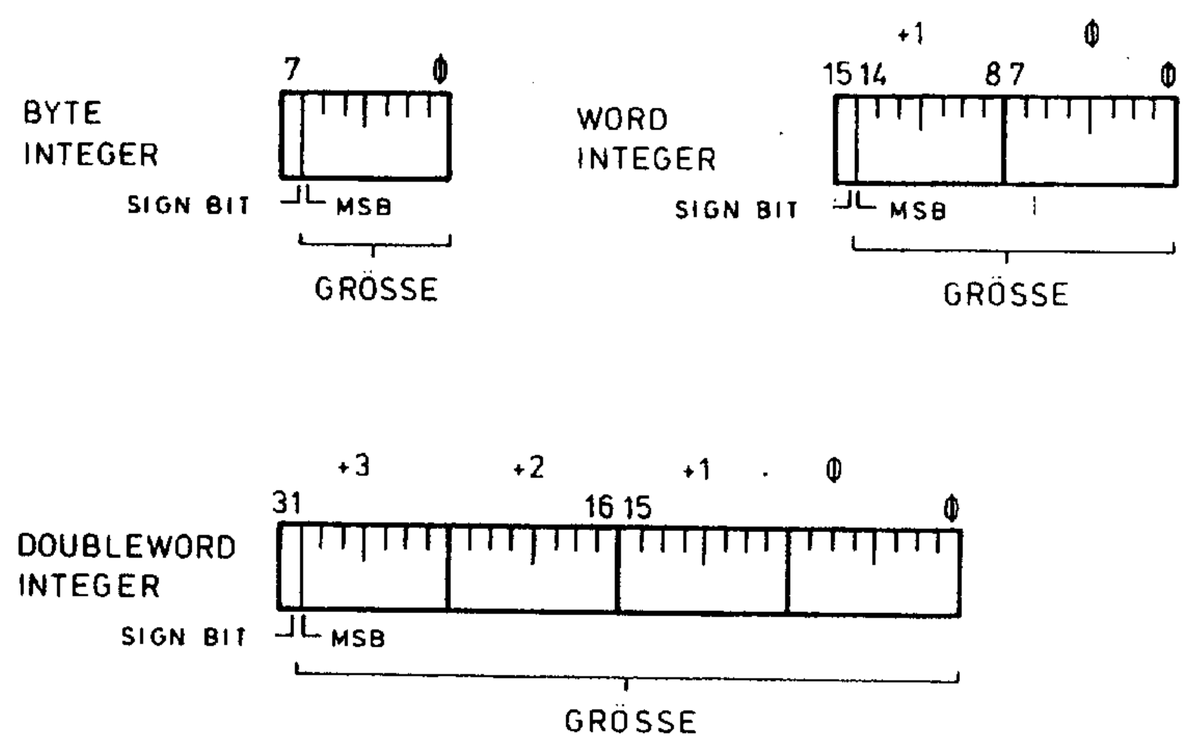
Für minimale Ausführungszeiten sollten, wenn möglich, Words und Doublewords an ihren natürlichen Adreßgrenzen beginnen. Dies sind für Words Adressen, die durch 2 teilbar sind, und für Doublewords solche, die durch 4 teilbar sind.

Obwohl Bytes, Words und Doublewords die fundamentalen Datentypen von 80486-Operanden sind, unterstützt der Prozessor weitere Interpretationen dieser Operanden.

Die folgenden zusätzlichen Datentypen, die der 80486 ebenfalls unterstützt, werden im allgemeinen von höheren Programmiersprachen benützt:

Integer:

- Ist eine vorzeichenbehaftete (signed) Binärzahl, die in einem 32-Bit-Doubleword oder in einem 16-Bit-Word oder in einem 8-Bit-Byte enthalten sein kann.

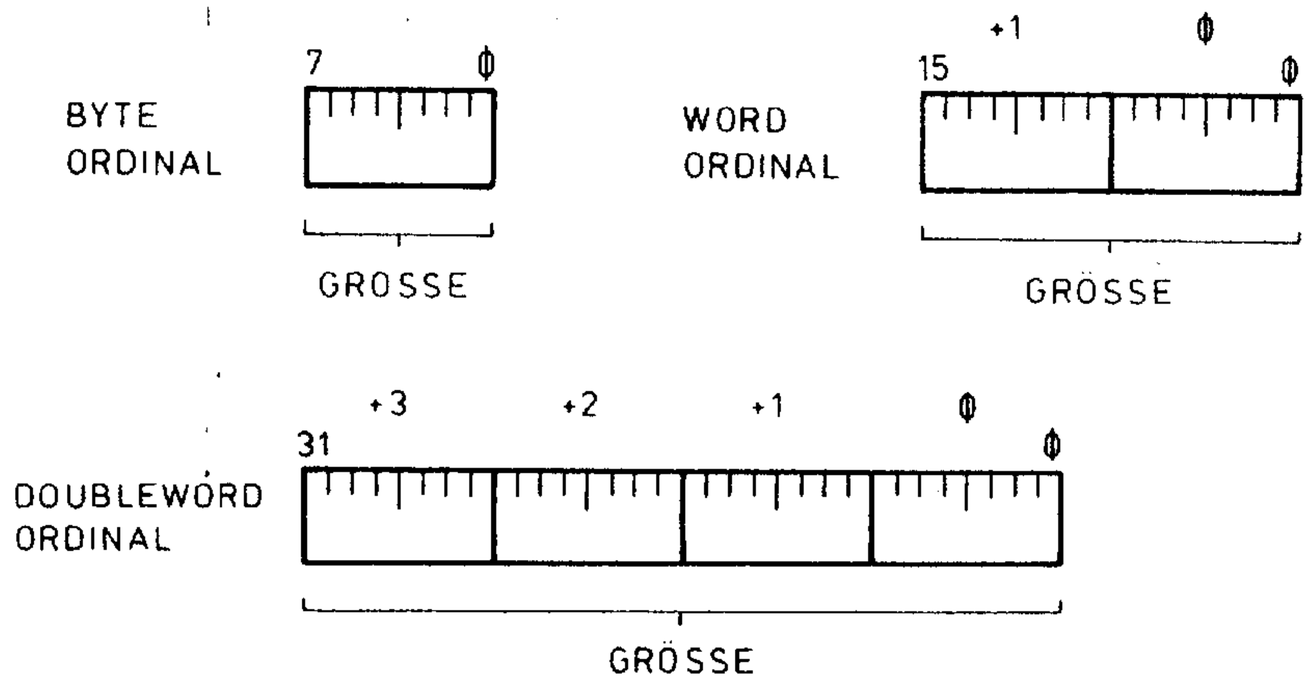


- Das Vorzeichenbit (Sign Bit) ist im Bit 7 eines Bytes, im Bit 15 eines Words oder im Bit 31 eines Doublewords untergebracht.
  Das Vorzeichenbit hat den Wert ***0*** für ***positive*** Integerzahlen und den Wert **1** für ***negative*** Integerzahlen. Dabei werden negative Integerzahlen in der Zweierkomplement-Notation dargestellt.
- Das das ***M***ost ***S***ignificant ***B***it (MSB) für das Vorzeichen benützt wird, liegt die

Größe einer 8-Bit-Integerzahl im Bereich – 128 ... + 127, die Größe einer 16-Bit-Integerzahl im Bereich – 32768 ... + 32767 und die Größe einer 32-Bit-Integerzahl im Bereich -2^{31} ... $+2^{31}-1$.

Ordinal:

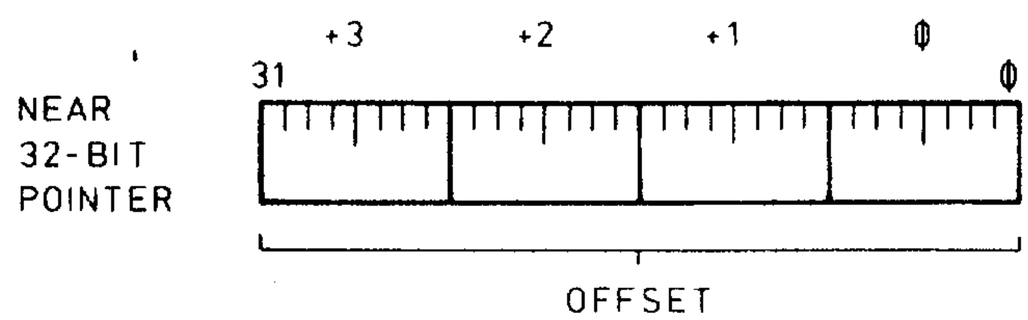
- ist eine vorzeichenlose (unsigned) Binärzahl, die in einem 32-Bit-Doubleword oder in einem 16-Bit-Word oder in einem 8-Bit-Byte enthalten sein kann.



- Bei diesem Datentyp werden ***alle*** im Operanden enthaltenen Bits zur Bestimmung seiner Größe benützt. Die Größe einer 8-Bit-Ordinalzahl liegt im Bereich 0 ... 255, die Größe einer 16-Bit-Ordinalzahl im Bereich 0 ... 65535 und schließlich die Größe einer 32-Bit-Ordinalzahl im Bereich 0 ... $2^{32}-1$.

Near Pointer:

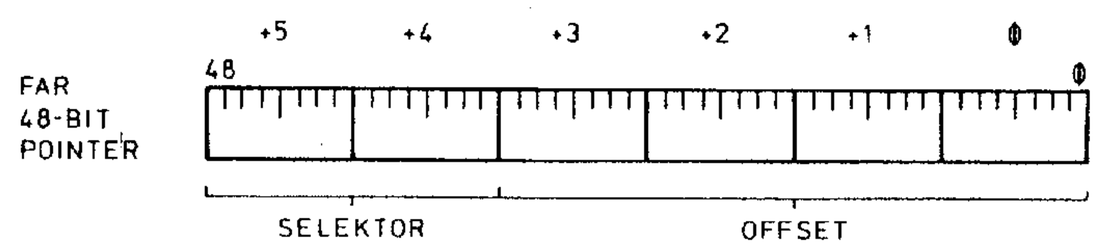
- ist eine 32-Bit logische Adresse. Ein „Near Pointer" ist identisch mit dem Offset innerhalb eines Segments und enthält daher einen 32-Bit-Ordinalwert.



- Ein „Near Pointer" kann sowohl im flachen als auch im segmentierten Speicherorganisations-Modell benützt werden.

Far Pointer:

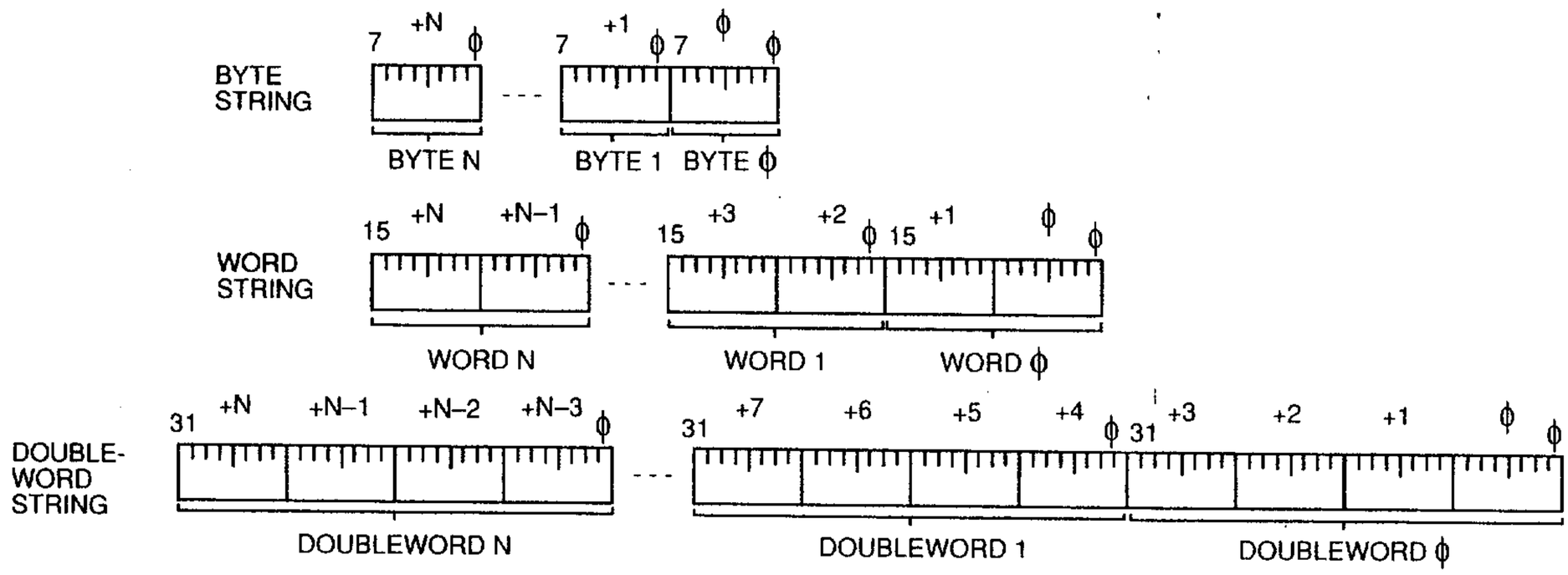
- ist eine 48-Bit logische Adresse, die aus 2 Komponenten besteht: einer 16-Bit Segment-Selektor-Komponente und einer 32-Bit Offset-Komponente.



- „Far Pointer“ werden von Anwender-Programmierern nur dann benützt, wenn der System-Entwickler eine segmentierte Speicherorganisation vorgesehen hat.

String:

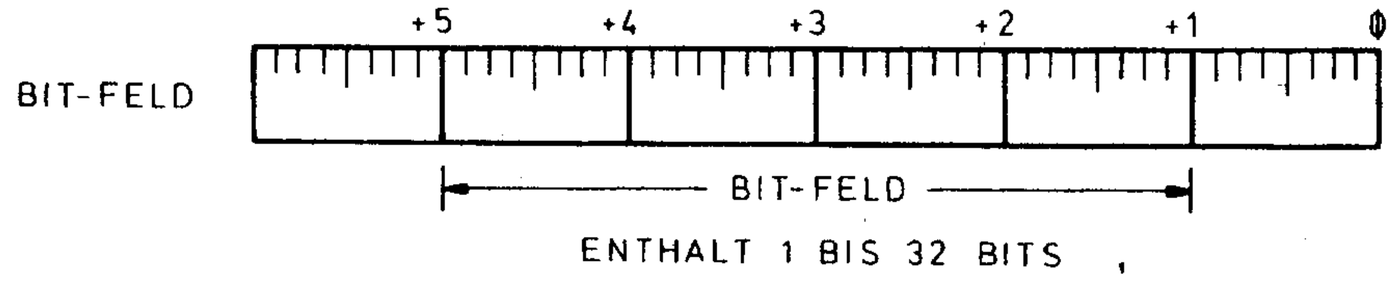
- ist eine fortlaufende Sequenz von Bytes, Words oder Doublewords



- In einem String können zwischen 0 ... $2^{32} - 1$ Bytes (4 Gigabytes) enthalten sein.

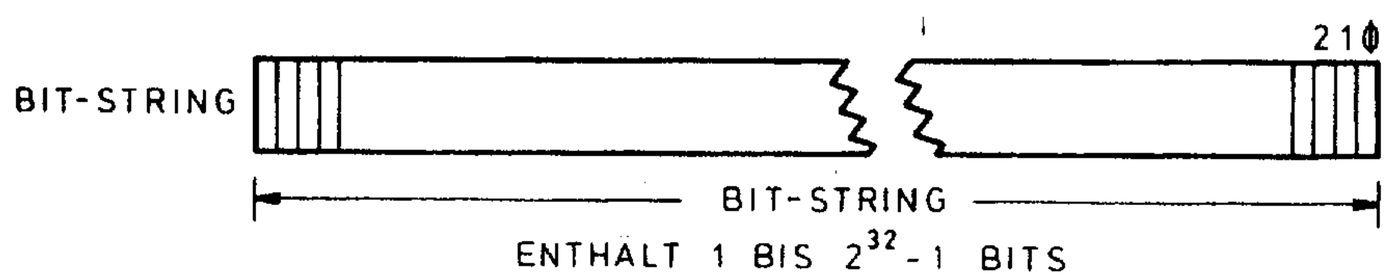
Bit-Feld:

- ist eine fortlaufende Sequenz von Bits. Ein Bit-Feld kann an einer beliebigen Bit-Position innerhalb eines beliebigen Bytes beginnen und bis zu 32 Bits enthalten.



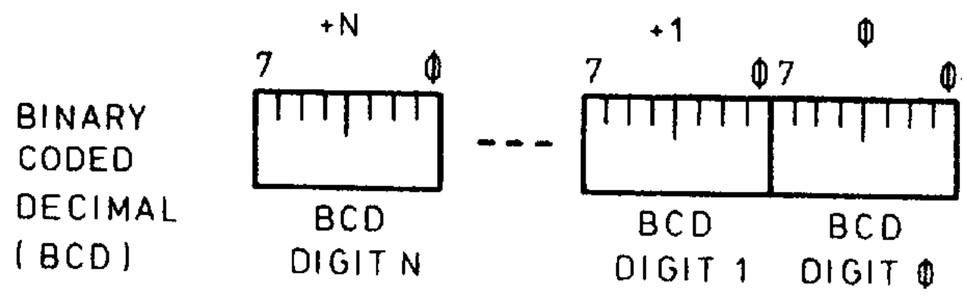
Bit String:

- ist eine fortlaufende Sequenz von Bits. Ein Bit-String kann an einer beliebigen Bit-Position innerhalb eines beliebigen Bytes beginnen und bis zu $2^{32} - 1$ Bits enthalten.



BCD:

- ist die binär-codierte Darstellung einer Dezimalzahl (***B***inary ***C***oded ***D***ecimal) im Bereich 0 ... 9. Eine einzelne BCD-Zahl wird als vorzeichenlose Byte-Einheit (Digit) gespeichert und daher als ***ungepackt*** bezeichnet.



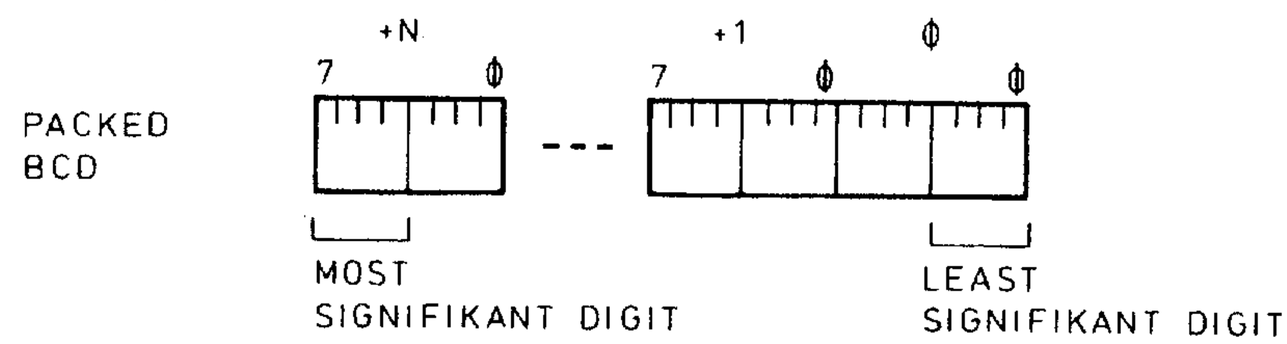
- Die Größe einer solchen Zahl wird duch das niederwertige Halbbyte bestimmt.

Dabei werden die hexadezimalen Werte 0 ... 9 als Dezimalzahlen interpretiert.

- Für Multiplikationen und Divisionen muß das höherwertige Halbbyte Null sein. Für Additionen und Subtraktionen darf das obere Halbbyte beliebige Werte enthalten.

Packed BCD:

- ist die binär-codierte Darstellung von zwei Dezimalzahlen, die jeweils im Bereich 0 ... 9 liegen.
  Zwei einzelne BCD-Zahlen, jede in einem Halbbyte, werden vorzeichenlos als Byte-Einheit (2 Digits) gespeichert und daher als ***gepackt*** bezeichnet.

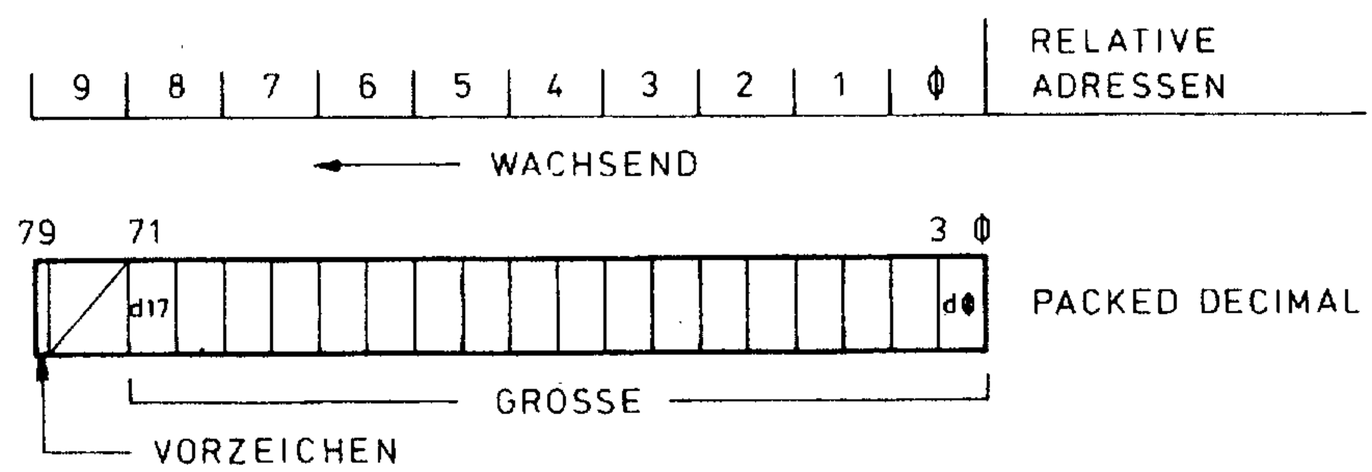


- Das Digit im höherwertigen Halbbyte wird als ***M***ost ***S***ignificant ***D***igit ***(MSD)*** und das im niederwertigen Halbbyte als ***L***east ***S***ignificant ***D***igit ***(LSD)*** bezeichnet. Da in jedem Halbbyte Werte von 0 ... 9 gültig sind, kann ein gepacktes Dezimal-Byte im Bereich 00 ... 99 liegen.

Die auf dem Chip integrierte Floating-Point-Einheit (FPU) unterstützt noch weitere Datentypen:

18-Digit Packed BCD:

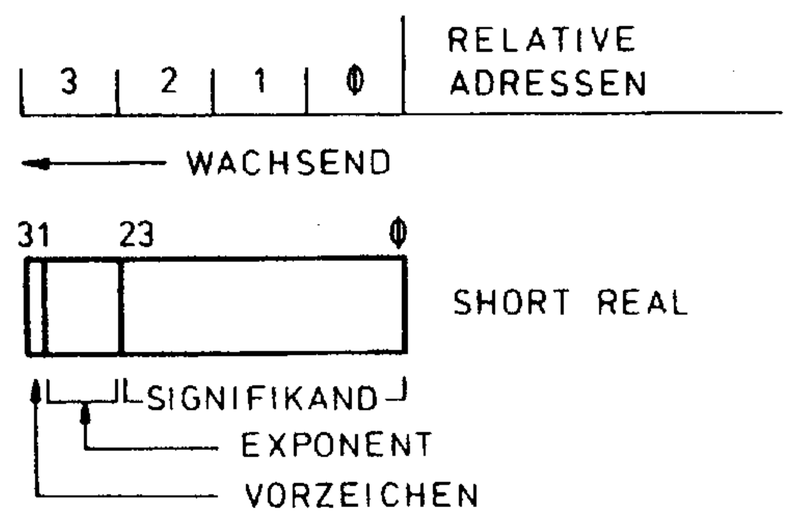
- ist eine 80-Bit vorzeichenbehaftete gepackte Dezimalzahl und wird in einem Feld von 10 zusammenhängenden Bytes gespeichert.
  In diesem Datentyp stehen 18 Halbbytes zur Speicherung von 18 dezimalen Digits (d0 ... d17) zur Verfügung.



- Die Bits 72 ... 78 sind ohne Bedeutung, während Bit 79 als Vorzeichen interpretiert wird: 0 = positiv, 1 = negativ.
  Negative Zahlen werden ***nicht*** in der Zweierkomplement-Notation dargestellt. Sie unterscheiden sich von positiven nur durch das Vorzeichen. Ihr Zahlenbereich liegt zwischen $-99 \ldots 99 \leqslant \times \leqslant +99 \ldots 99$.

Short Real:

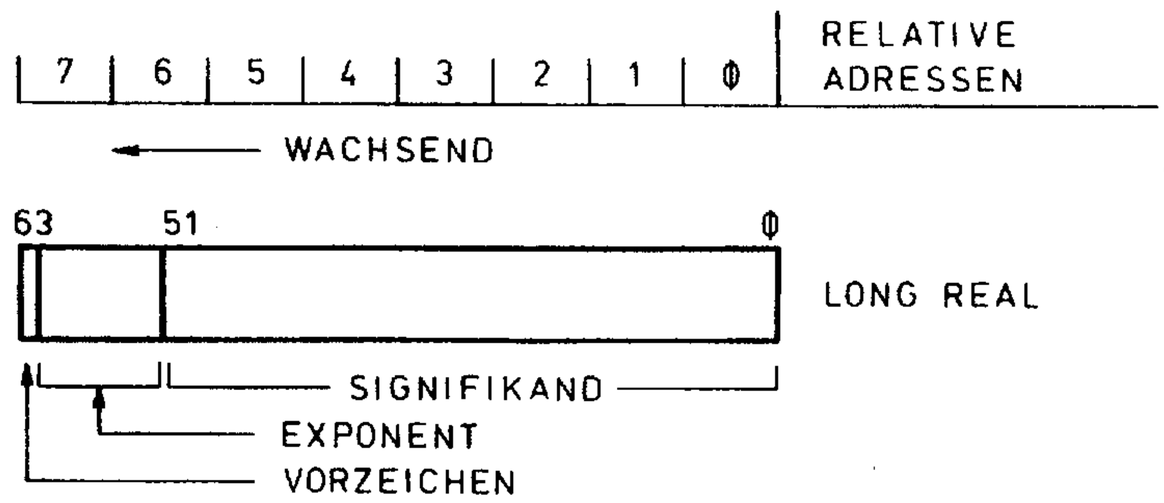
- hat eine Länge von 32 Bits und wird daher in einem Doubleword gespeichert.



Ihr Zahlenbereich liegt zwischen
$1.2 \times 10^{-38} \leq | \times | \leq 3.4 \times 10^{38}$.

Long Real:

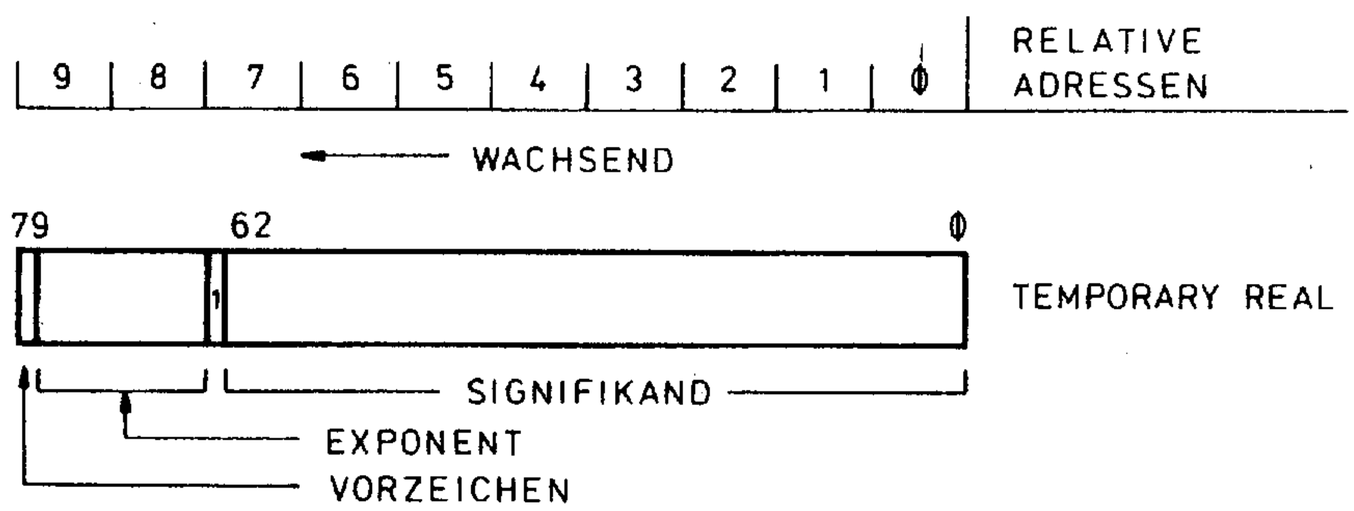
- hat eine Länge von 64 Bits und wird daher in einem Quadword gespeichert.



Ihr Zahlenbereich liegt zwischen
$2.3 \times 10^{-308} \leq | \times | \leq 1.7 \times 10^{308}$.

Temporary Real:

- hat eine Länge von 80 Bits und wird in einem Feld von 10 zusammenhängenden Bytes gespeichert.



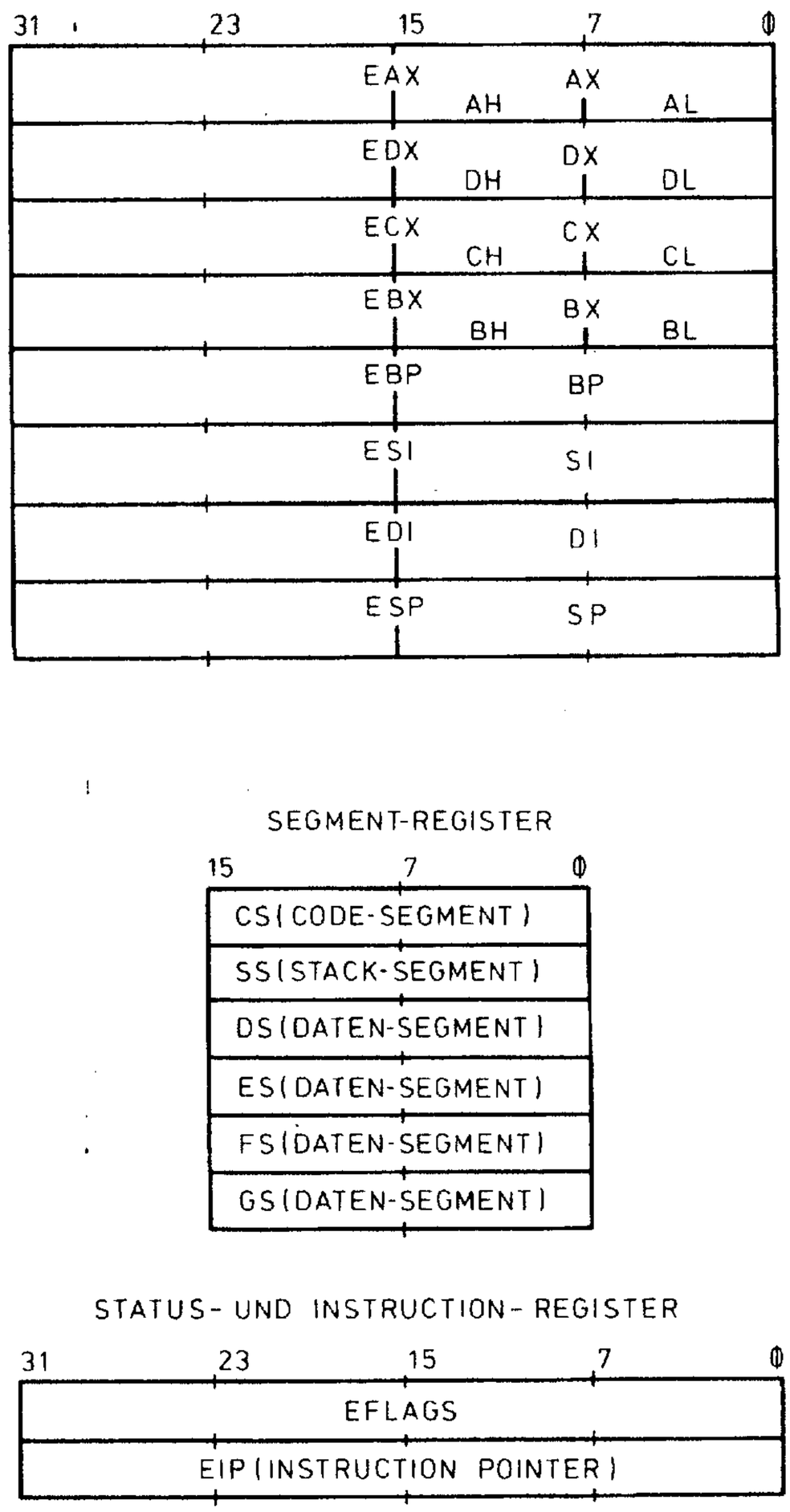
Ihr Zahlenbereich liegt zwischen
$3.4 \times 10^{-4932} \leq | \times | \leq 1.1 \times 10^{4932}$.

Alle Datentypen, die von der FPU unterstützt werden, sind vom amerikanischen Verband IEEE als Standard 754 definiert.

1.3 Register

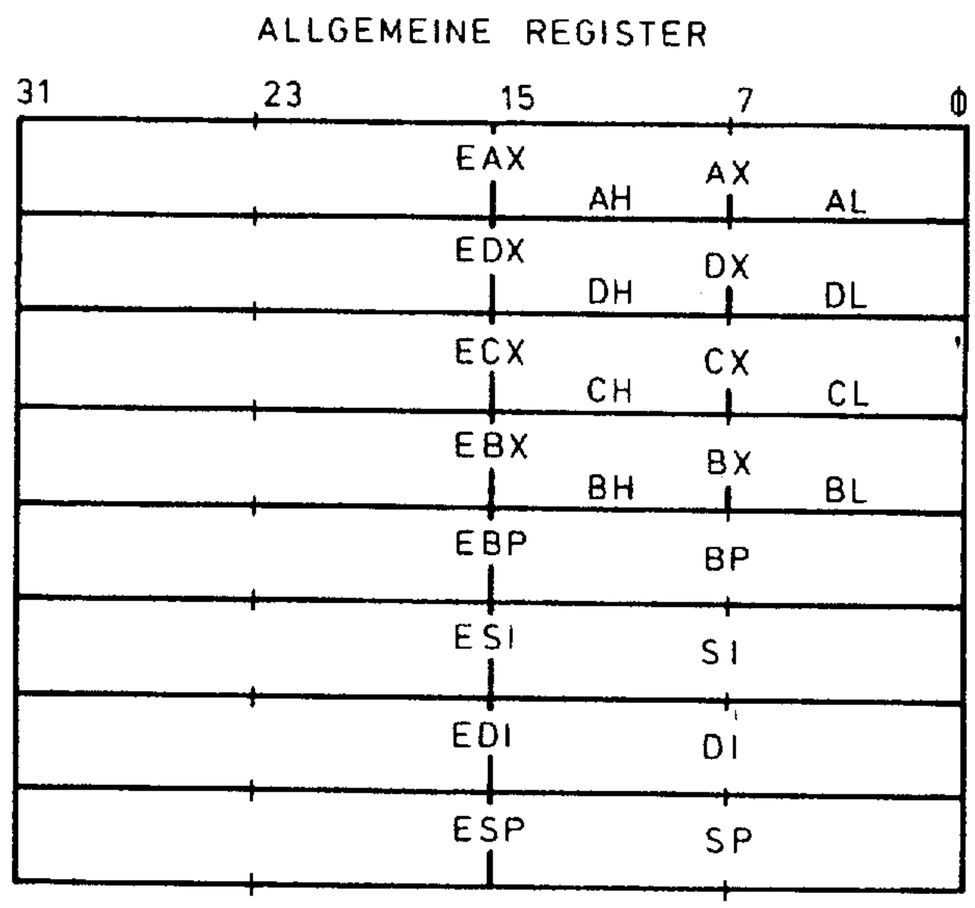
Die Integer-Einheit (IU) des 80486 enthält insgesamt 16 Register, die für den Anwender-Programmierer von Interesse sind. Das folgende Bild zeigt diese Register im Überblick. Dabei ist ersichtlich, daß sie in ***drei*** Kategorien eingeteilt sind:

1. Allgemeine Register
2. Segment-Register
3. Status- und Instruction-Register



1.3.1 Allgemeine Register

Die allgemeinen Register des 80486 sind die acht 32-Bit-Register EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI.



Sie enthalten primär Operanden für logische und arithmetische Operationen. Alle Register, außer ESP, können als ***Index-Register*** für die effektive Adreßberechnung von Speicher-Operanden benützt werden. Im Gegensatz dazu stehen bei den 16-Bit-Vorgängern nur DI und SI als Index-Register zur Verfügung.

Wie das angegebene Bild zeigt, haben die niederwertigen Worte der 32-Bit-Register eigene Namen und können als individuelle Einheiten behandelt werden. Diese ***Wortregister*** sind mit AX, BX, CX, DX, BP, SI, DI, IP bezeichnet. Dadurch ist der Umgang mit 16-Bit-Daten für die Kompatibilität zu den 16-Bit-Vorgängern gewährleistet.

Weiterhin ist im Bild zu erkennen, daß auch jedes Byte der 16-Bit-Register AX, BX, CX und DX einen eigenen Namen hat und als individuelle Einheit behandelt werden kann. Diese ***Byte-Register*** sind mit AH, BH, CH, DH (High-Bytes) und AL, BL, CL, DL (Low-Bytes) bezeichnet. Dadurch ist der Umgang mit „Characters" oder anderen 8-Bit-Daten gewährleistet.

1.3.2 Segment-Register

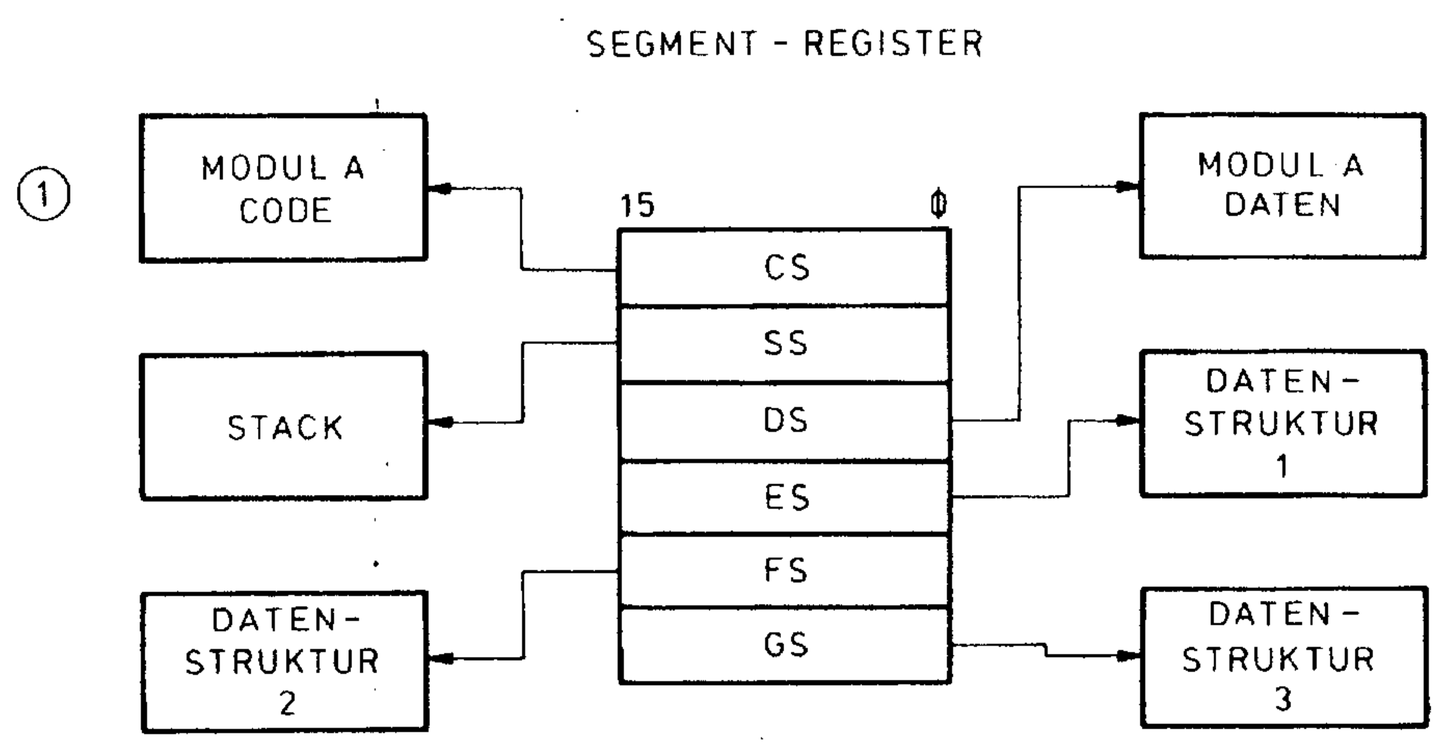
Ein vollständiges Programm besteht aus mehreren unterschiedlichen Modulen, wobei jedes dieser Module Befehle und Daten enthält. Während des Programmablaufs werden aber ***nicht alle Module,*** sondern nur eine jeweils notwendige Teilmenge benützt.

Diese Tatsache unterstützt die 80486-Architektur durch einen Mechanismus, der den direkten Zugriff auf die Befehle und Daten des augenblicklichen aktuellen Moduls erlaubt.

Hierzu stehen ***6 Speicher-Segmente*** für den unmittelbaren Zugriff während des Pro-

grammablaufs zur Verfügung. Um die augenblicklich aktuellen Segmente zu identifizieren, benützt der 80486 die ***16-Bit-Segment-Register*** CS, DS, SS, ES, FS, GS. Jedes dieser Register spezifiziert eine besondere Art von Segment.

Wie das folgende Bild zeigt, wird der Inhalt eines Segments durch Mnemoniks wie ***Code, Daten, Stack*** charakterisiert.

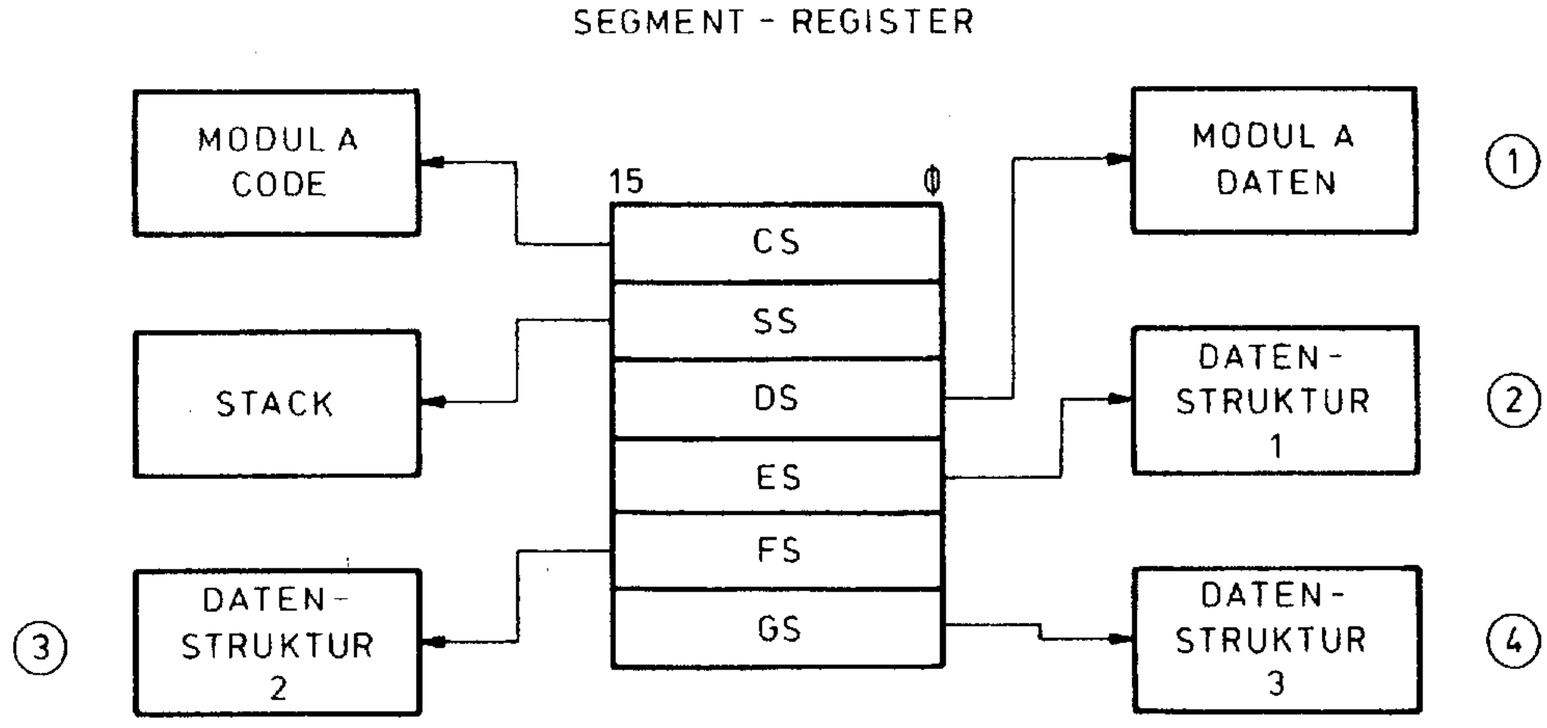


Das Segment, das die augenblicklich auszuführenden Befehle eines Moduls enthält z. B. von Modul A ①, wird als ***augenblickliches Code-Segment*** bezeichnet und durch das Segment-Register CS spezifiziert.

Sollte das augenblickliche Code-Segment z. B. Prozedur-Aufrufe enthalten, so ist für die Bearbeitung der lokalen Parameter eine Speicher-Region erforderlich, die als ***Stack*** bezeichnet wird. Alle Stack-Operationen benützen das Segment-Register SS, wobei SS die Position des Stack im Speicher festlegt.

Die Segment-Register DS, ES, FS, GS erlauben die Spezifikation von ***vier Daten-Segmenten,*** die für das augenblicklich ausgeführte Programm adressierbar sind. Im Vergleich zu den 16-Bit-Vorgängern stehen beim 80486 zwei zusätzliche Daten-Segmente zur Verfügung, die durch FS, GS identifiziert werden.

Die Zugriffsmöglichkeit auf ***vier separate Daten-Regionen*** erhöht die Flexibilität eines Programms im Umgang mit unterschiedlichen Typen von Daten-Strukturen.



BEISPIEL:

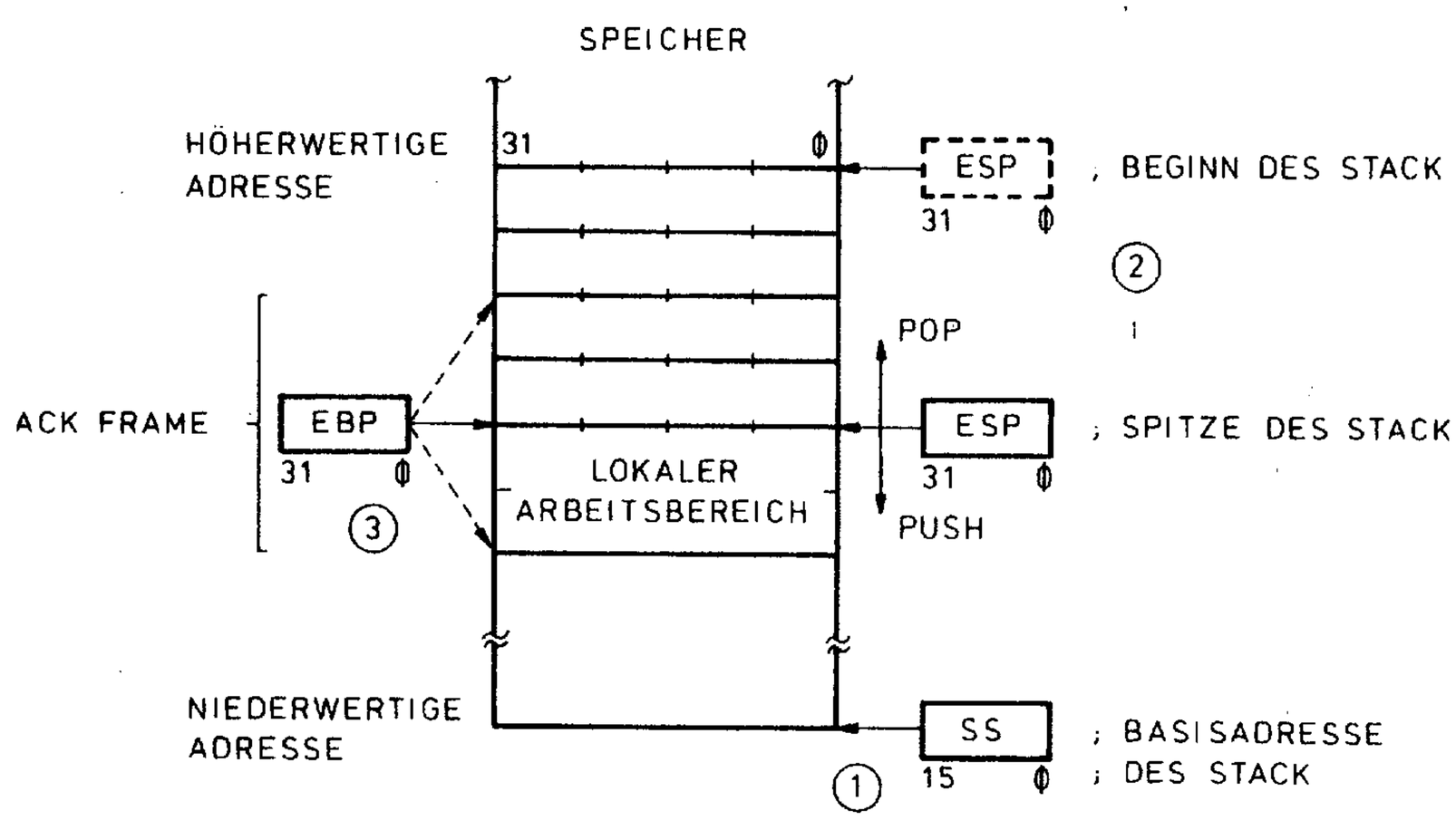
- Ein Segment-Register (z. B. DS) kann die Daten-Struktur des augenblicklich aktiven Code-Segments eines Moduls (z. B. von Modul A) identifizieren ①.
- Ein anderes Segment-Register (z. B. ES) zeigt auf die Daten eines in höherer Programmier-Sprache geschriebenen Moduls ②.
- Ein weiteres Segment-Register (z. B. FS) kann dynamisch erzeugte Daten-Strukturen identifizieren. Dem PASCAL-Programmierer sind solche Daten-Strukturen als ***Listen*** bekannt ③.
- Schließlich kann noch ein letztes Segment-Register (z. B. GS) eine Daten-Struktur identifizieren, die von mehreren Programmen gemeinsam benützt wird ④.

Sollte es erforderlich sein, daß ein Programm zusätzliche Segmente benötigt, so können während des Programmablaufs die Segment-Register problemlos umgeladen werden. Dadurch wird erreicht, daß die neuen Initialisierungs-Werte in DS, ES, FS oder GS noch weitere Daten-Segmente identifizieren.

1.3.3 Implementierung des Stack

Alle Stack-Operationen werden durch folgende 3 Register unterstützt:

- Stack-Segment-Register SS ①
- Stack-Pointer-Register ESP ②
- Stack-Frame-Pointer-Register EBP ③



1.3.3.1 Das Stack-Segment-Register SS

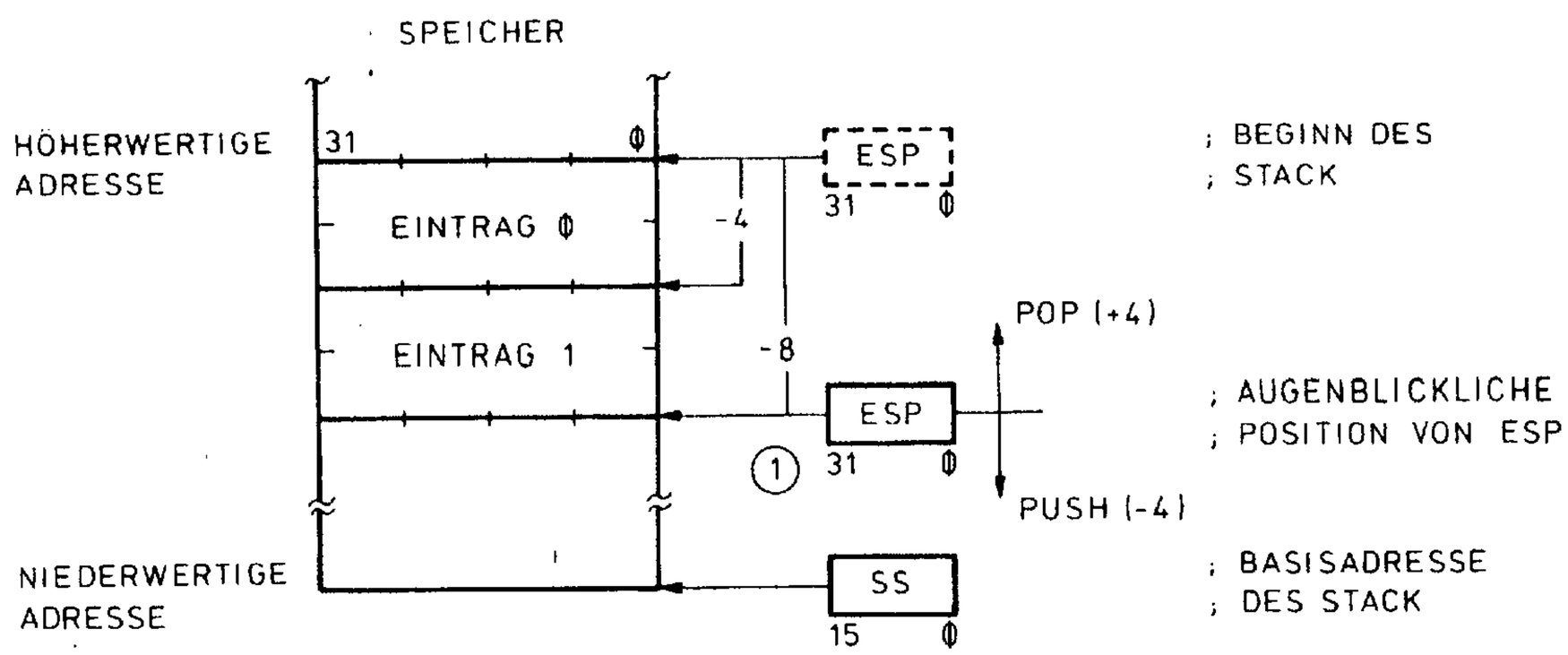
Ein Stack hält sich im Speicher auf und wird durch das SS-Register identifiziert, d. h. SS legt die Basis-Adresse des Stack fest ①. Jedes 80486-System kann eine Vielzahl von Stacks enthalten. Welcher Stack aber im Augenblick benützt wird, hängt vom Initialisie-

rungs-Wert im SS-Register ab. Dies bedeutet, daß zu einer bestimmten Zeit immer nur ein Stack direkt adressierbar ist. Dieser Stack wird dann als ***augenblicklicher Stack*** bezeichnet.

Alle vom Prozessor ausgeführten Stack-Operationen benützen automatisch das SS-Register.

1.3.3.2 Das „Stack-Pointer "-Register ESP

ESP zeigt immer auf den letzten Eintrag im Stack und wird implizit von PUSH- und POP-Befehlen bei Prozedur-Aufrufen, beim Verlassen von Prozeduren und bei Interrupt-Operationen benützt.



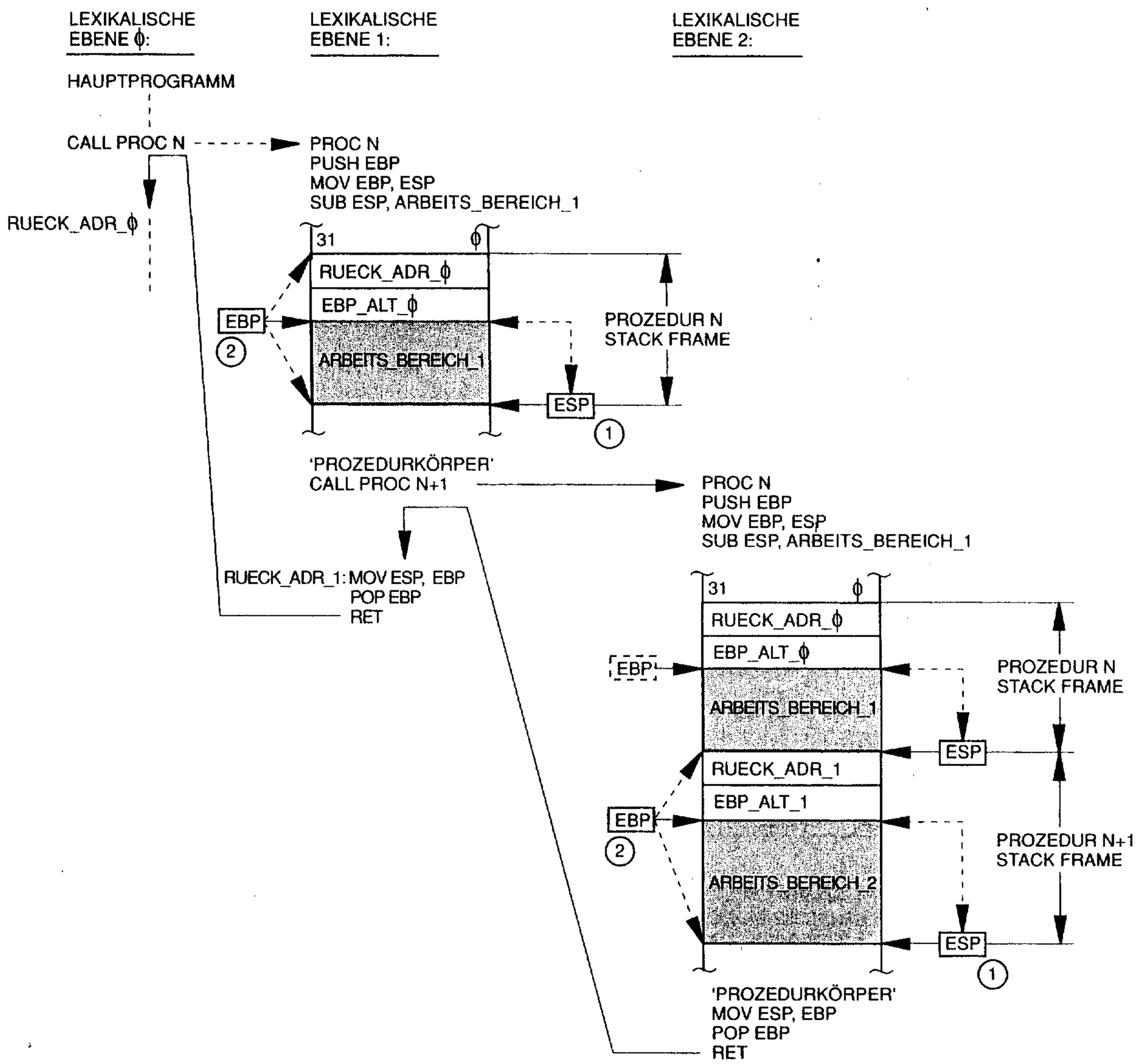
Wenn ein Eintrag in den Stack erfolgt (PUSH-Operation), dekrementiert der 80486 ESP mit 4 (ESP – 4) und schreibt den Eintrag an die augenblickliche Position von ESP ①. Diese Position wird als „*T*op *O*f *S*tack" ***(TOS)*** bezeichnet.

Wenn ein Eintrag aus dem Stack entfernt wird (POP-Operation), liest der 80486 den Eintrag von der augenblicklichen ESP-Position und inkrementiert anschließend ESP mit 4 (ESP + 4).

Wie das angegebene Bild illustriert, wächst der Stack im Speicher in Richtung niederwertige Adressen.

1.3.3.3 Das „Stack Frame Pointer"-Register EBP

Das EBP-Register kann benützt werden, um in eleganter Weise Daten-Strukturen, Variablen und dynamisch zugewiesene Arbeitsbereiche im Stack zu adressieren. Dies soll am folgenden Bild erläutert werden.



Es existieren zwei Prozeduren PROC N und PROC N+1, in denen ***lokale Variablen*** bearbeitet werden. Auf lokale Variablen kann ***nur*** in der ***augenblicklichen Prozedur*** zugegriffen werden, das heißt, sie werden gegen den Zugriff externer Programmteile abgegrenzt.

Zu diesem Zweck wird ESP so verschoben, daß im Stack genügend Raum zur Bearbeitung lokaler Variablen entsteht. Die Größe des Arbeitsbereichs kann dabei abhängig von der Aufgabe in der jeweiligen Prozedur ***dynamisch*** erfolgen.

Gemäß dem angegebenen Bild wird PROC N mit

SUB ESP,ARBEITS_BEREICH_1

ein anderer lokaler Arbeitsbereich festgelegt als in PROC N+1 mit

SUB ESP,ARBEITS_BEREICH_2

①. In beiden Fällen kennzeichnet ESP die Spitze des Stack ① und EBP den Beginn des Arbeitsbereichs ②.

Liegen ***verschachtelte (nested) Prozeduren*** wie im Beispiel vor, kennzeichnet ESP außerdem die Spitze der zur augenblicklichen Prozedur gehörenden Stackbelegung ***(„Stack***

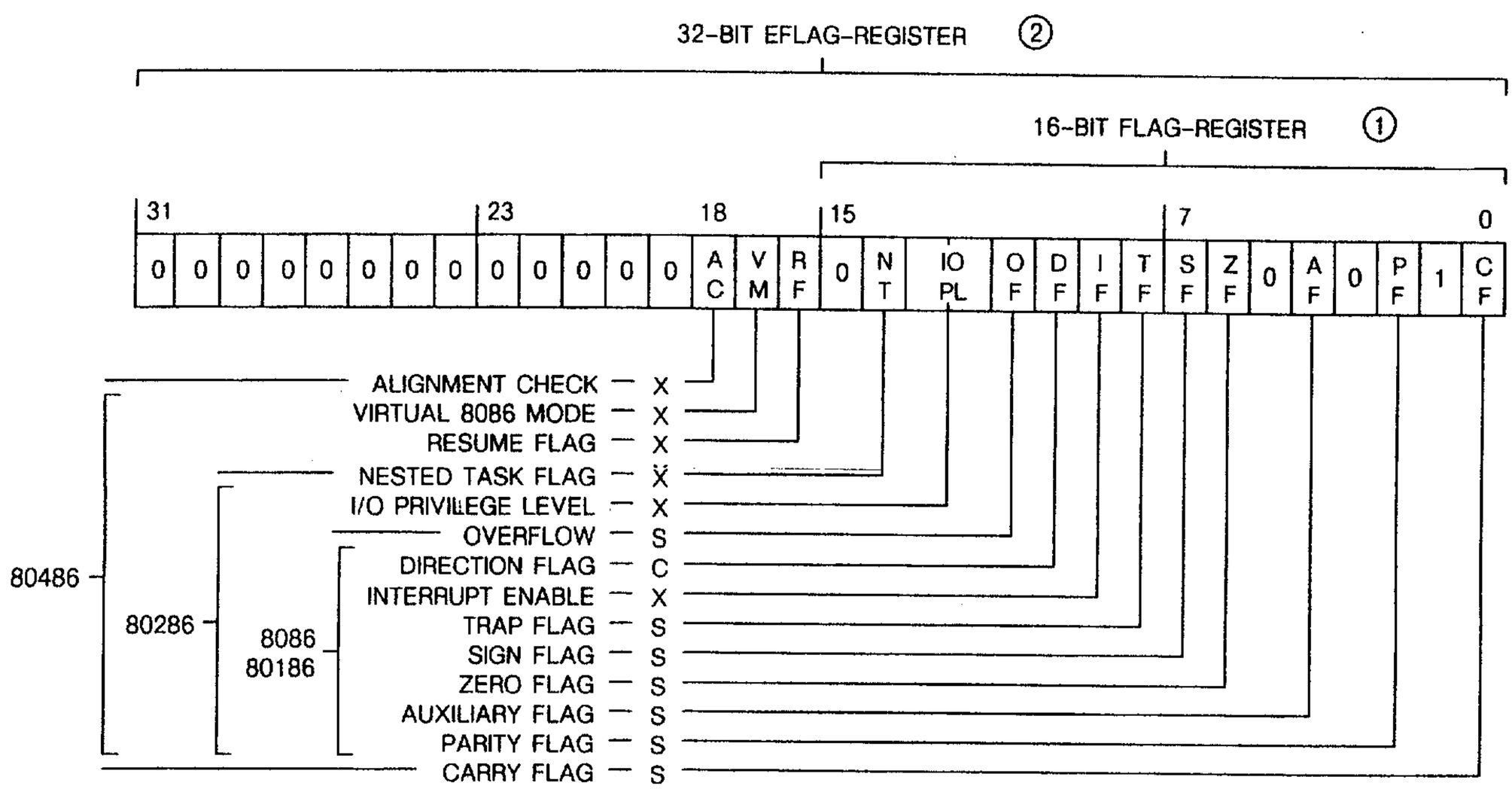
Frame"). Das heißt, ESP ist nur im „Stack Frame" der augenblicklichen Prozedur-Ebene (im Beispiel in der Ebene 2) aktuell.

Innerhalb des aktuellen „Stack Frame" kann der lokale Arbeitsbereich mit EBP erreicht werden, z. B. mit MOV [EBP–4] ,EAX ②. Damit wird EBP als sogenannter „Stack Frame Pointer" benützt.

Duch diese Technik sind die „Stack Frames" der verschiedenen Prozedur-Ebenen (lexikalische Ebenen) und damit ihre lokalen Arbeitsbereiche gegeneinander abgegrenzt.

1.3.4 Das Flag-Register

Das Flag-Register des 80486 ist ein 32-Bit-Register, das als ***EFLAG*** bezeichnet wird. Das folgende Bild zeigt die Position der individuellen Flags innerhalb dieses Registers:



Es ist ersichtlich, daß die EFLAGs in folgende 3 Bit-Felder unterteilt sind:

- Das Bit-Feld 0 ... 11, das mit den Flags der 8086- bzw. 80186-CPUs identisch ist.
- Das Bit-Feld 0 ... 15, das den Flags der 80286-CPU entspricht.
- Das Bit-Feld 0 ... 18, das von der 80486-CPU benützt wird.

Die niederwertigen 16 Bits des EFLAG-Registers können als Einheit betrachtet werden und sind nützlich für die Ausführung von 8086-, 80186- oder 80286-Code ①.

Das komplette EFLAG-Register wird nur vom 80486 benützt, wobei einige Bit-Positionen reserviert sind und daher nicht definiert werden dürfen ②.

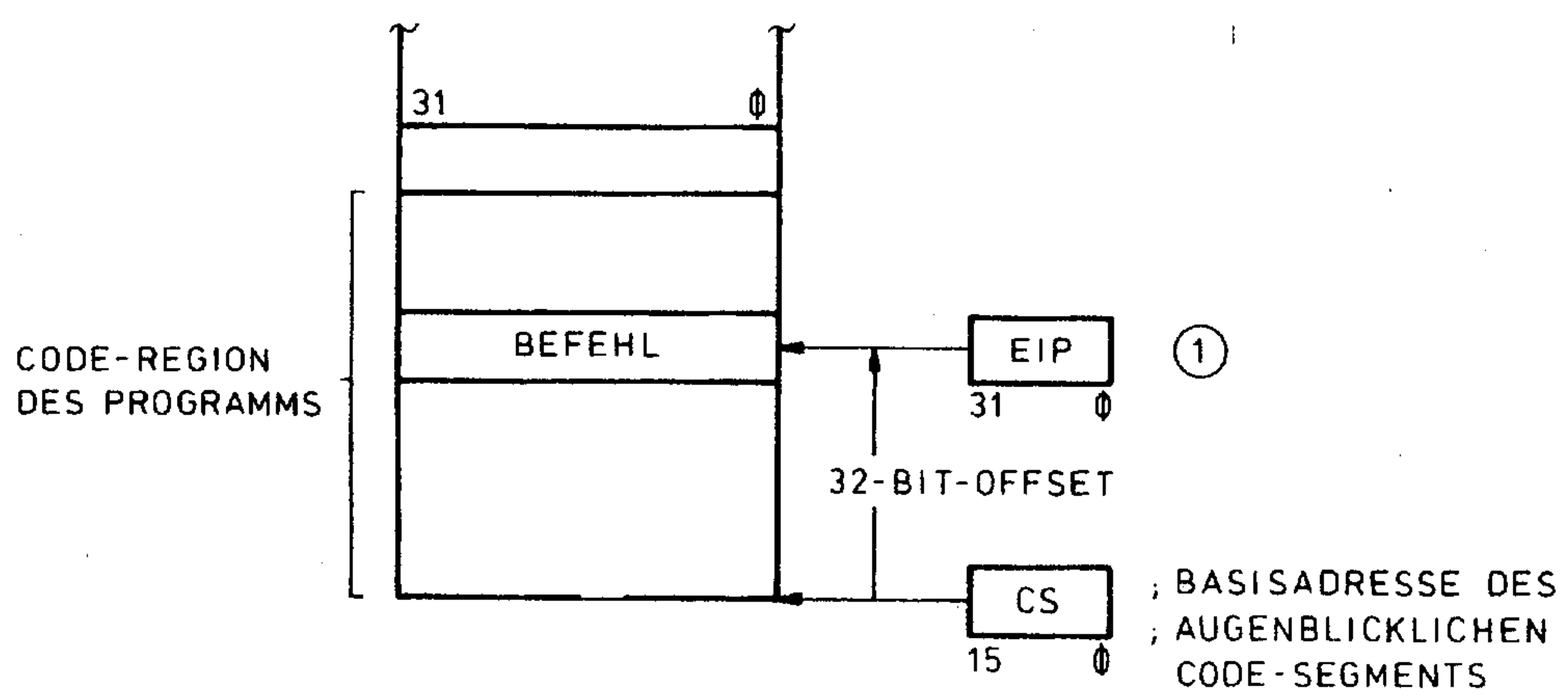
Aus der Sicht des Anwender-Programmierers sind die EFLAGs in 3 Funktionsgruppen eingeteilt:

- Status-Flags (S)
- Control-Flags (C)
- System-Flags (X)

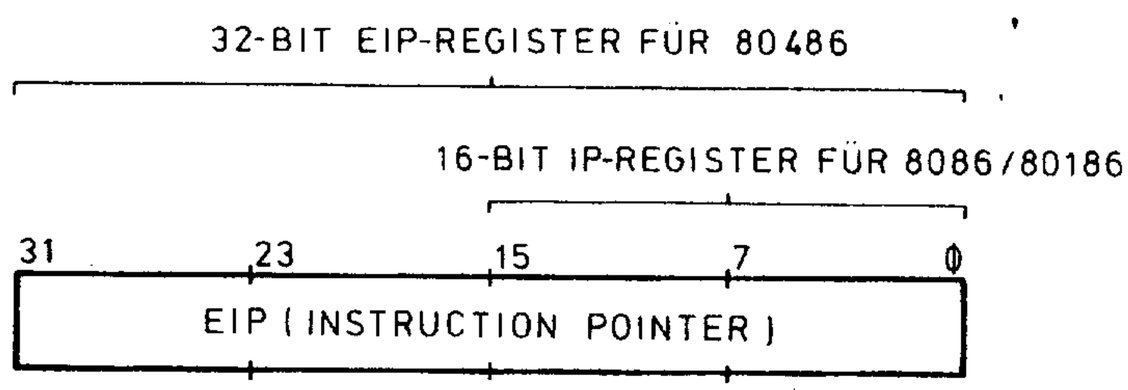
Im Laufe dieses Buches sollen nur die System-Flags erläutert werden, da sie die gegenüber dem 8086 oder 80186 die neuartige Architektur des 80486 unterstützen.

1.3.5 Der „Instruction Pointer" EIP

Der „Instruction Pointer" EIP enthält einen 32-Bit-Ordinal-Wert, der den Offset eines Befehls relativ zur Basis des augenblicklichen Code-Segments darstellt ①.



Wie das folgende Bild zeigt, werden die niederwertigen 16 Bits von EIP als IP bezeichnet und können vom Prozessor als eine Einheit behandelt werden.



Dieses Leistungsmerkmal gibt dem 80486 die Möglichkeit, Befehle auszuführen, die für den 8086, 80186 oder 80286 entwickelt worden sind.

1.4 „Real Mode"-Architektur

Wenn der 80486 zurückgesetzt (RESET) oder seine Stromversorgung eingeschaltet wird, befindet er sich im sogenannten „Real Mode". Im „Real Mode" hat der 80486 die gleiche Basis-Architektur wie der 8086. Allerdings ist jetzt ***zusätzlich*** der Zugriff auf den 32-Bit-Registersatz des 80486 erlaubt.

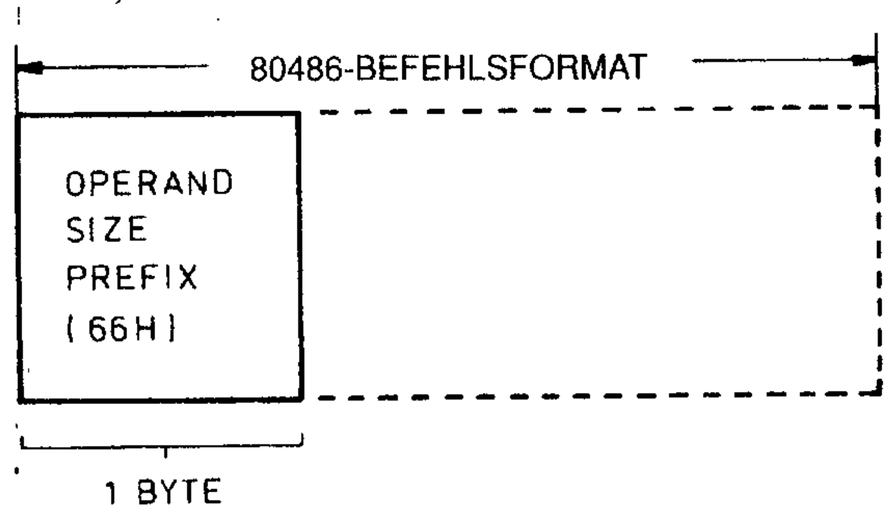
Weiterhin können alle ***neuen anwender-orientierten Befehle*** des 80486 im „Real Mode" benützt werden. Dies sind

1. die neuen Befehle, die bereits vom 80186 und 80286 eingeführt worden sind, wie zum Beispiel
   - String-I/O
   - ENTER und LEAVE
   - BOUND usw.
2. die neuen Befehle, die nur der 80486 zur Verfügung stellt, wie zum Beispiel
   - Lade SS-, FS- und GS-Register
   - Einzelbit-Befehle
   - Bit-Abtastung usw.

Genauso wie beim 8086 haben im „Real Mode" die Register-Operanden und die effektiven Adreßwerte eine voreingestellte Länge von 16 Bits.
Damit aber die 32-Bit-Register und die 32-Bit-Adressierungsmechanismen benützt werden können, stellt der 80486 eine sogenannte ***„Operand Size Prefix"*** (Code = 66H) und eine ***„Address Length Prefix"*** (Code = 67H) im Befehlsformat zur Verfügung.

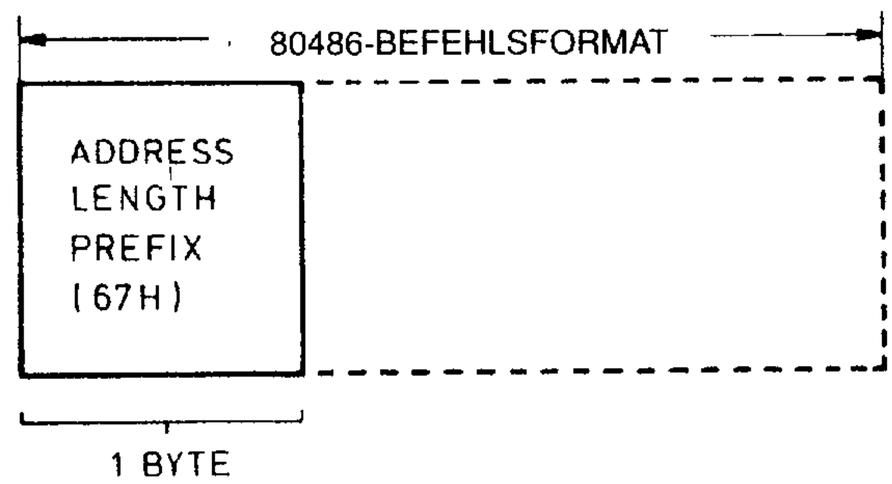
Immer dann, wenn im „Real Mode" ein Befehl ausgeführt wird, der explizit ein 32-Bit-Register oder eine 32-Bit effektive Adresse benützt, wird dem Befehl die entsprechende „Prefix" vorangestellt.

Beispiel 1: **MOV EAX,5**



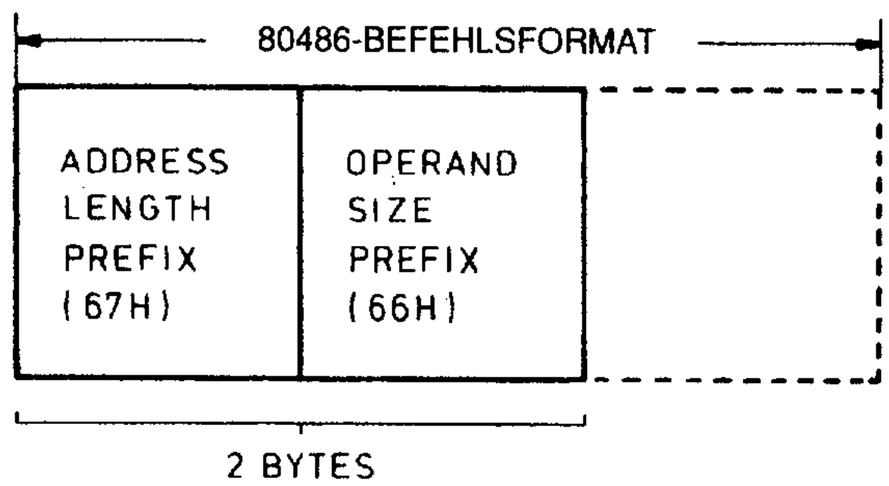
Hier wird dem Befehl die „Operand Size Prefix" vorangestellt, da er explizit das 32-Bit Register EAX benützt.

Beispiel 2: **MOV AX, [EBX]**



Hier wird dem Befehl die „Address Length Prefix" vorangestellt, da er explizit den 32-Bit-Adreßzeiger [EBX] benützt.

Beispiel 3: **MOV EAX, [EBX]**



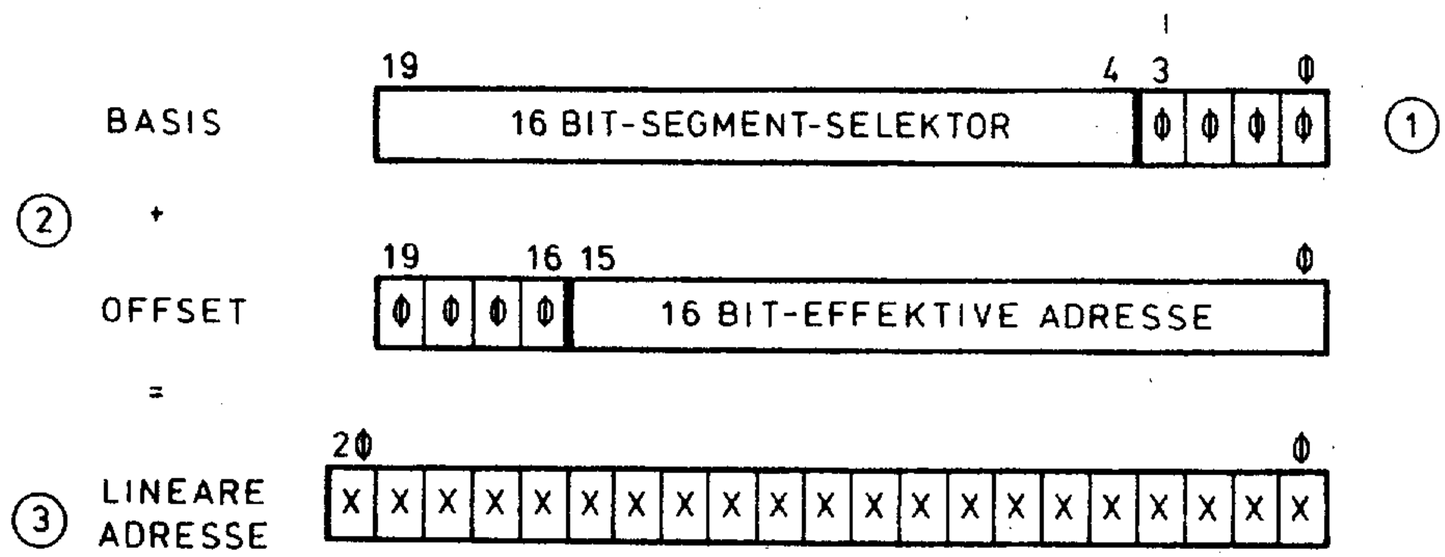
Hier wird dem Befehl die „Operand Size Prefix" ***und*** die „Adress Length Prefix" vorangestellt, da er explizit sowohl das 32-Bit-Register EAX als auch den 32-Bit Adreßzeiger [EBX] benützt.

Genauso wie beim 8086, beträgt im 80486-Real-Mode die maximale Segment-Länge 64K Bytes. Aus diesem Grund muß ein 32-Bit-Adreßzeiger einen Wert haben, der ***nicht*** größer als 0000FFFFH ist.

Da also der 4 Gigabyte physikalische Adreßraum im „Real Mode" nicht ausgenützt werden kann, ist diese Betriebsart für andere Zwecke vorgesehen. So wird der „Real Mode" primär dazu benützt, den Prozessor für die Protected Mode-Operationen vorzubereiten.

1.4.1 Speicheradressierung im „Real Mode"

Im „Real Mode" ist die maximale Speichergröße auf 1 MByte + 64 KByte begrenzt. Die Segment-Zuweisung erfolgt dabei in der gleichen Weise wie beim 8086. Um die physikalische Basisadresse des Segments zu bilden, wird der 16-Bit-Wert in einem Selektor um 4 Bits nach links verschoben ①. Danach werden die höherwertigen 4 Bits einer effektiven Adresse um 4 Nullen erweitert und zur physikalischen Basisadresse addiert ②. Die daraus resultierende Adresse wird als ***lineare Adresse*** bezeichnet, die mit der physikalischen Adresse einers Speicher-Operanden identisch ist ③.



Wie das angegebene Bild zeigt, kann die lineare Adresse bis zu 21 signifikante Bits (0 ... 20) haben. Dieses Leistungsmerkmal hat der 8086 nicht.

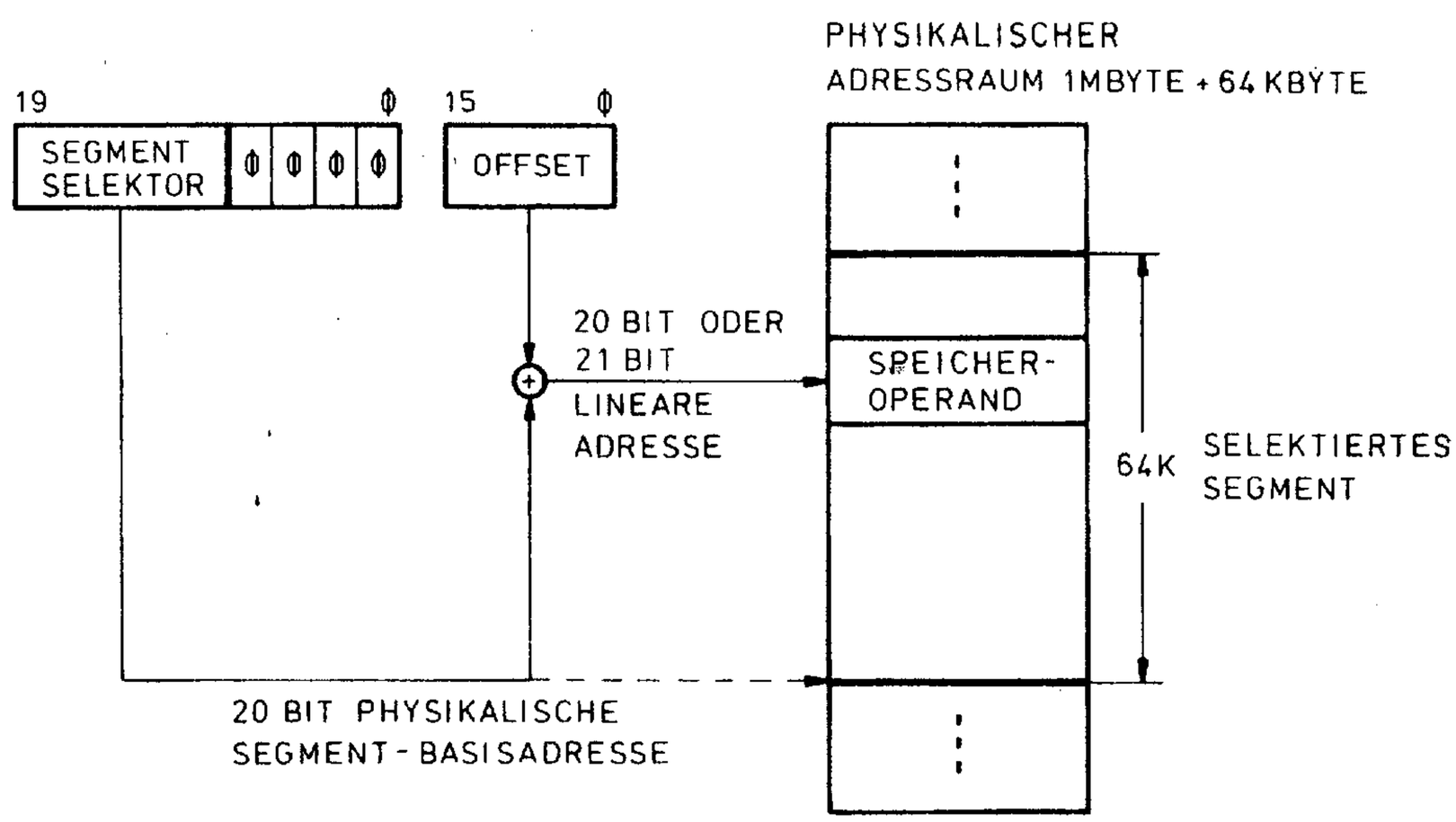
Beispiel:

Mit einem Selektor-Wert von 0FFFFH und einem Offset von 0FFFFH ergibt sich im „Real Mode" der 21-Bit lineare Adreßwert 10FFEFH (1 MByte + 65519 Bytes).

Basis	FFFF0H
+	
Offset	0FFFFH
=	
lineare Adresse	10FFEFH

Im Gegensatz dazu kann die lineare Adresse des 8086 nur bis zu 20 signifikante Bits (0 ... 19) enthalten. Für das angegebene Beispiel würde dies bedeuten, daß der 8086 das höchstwertige Bit der linearen Adresse abschneidet und anschließend die 20-Bit lineare Adresse 0FFEFH liefert.

Wie der angegebene Adressier-Mechanismus im „Real Mode" benützt wird, um einen Speicher-Operanden zu adressieren, zeigt das folgende Bild.



- Der Segment-Selektor liefert die 20-Bit physikalische Basis-Adresse des Segments, in dem sich der Speicher-Operand aufhält.
- Der 16-Bit-Offset gibt die Position des Speicher-Operanden ***relativ*** zur Segment-Basis an.
- Die 20-Bit lineare Adresse gibt die physikalische Position des Speicher-Operanden im 1 MByte-Adreßraum an.
- Die 21-Bit lineare Adresse gibt die physikalische Position des Speicher-Operanden im 64 K-Bereich jenseits vom 1MByte-Adreßraum an.

1.5 Adressierungsarten

Um Operanden zu spezifizieren, stellt der 80486 insgesamt 11 verschiedene Adressierungsarten zur Verfügung. Diese Adressierungsarten sind so optimiert, daß sie die Vielzahl von Daten-Referenzen, wie sie in höheren Programmiersprachen (z. B. C-386) gebraucht werden, wirksam unterstützen.

1.5.1 Register-Adressierung

Bei dieser Adressierungsart sind die Operanden in einem der folgenden Register untergebracht:

- In einem 32-Bit allgemeinen Register wie EAX, EBX, ECX, EDX, ESI, EDI, ESP oder EBP.
- In einem 16-Bit allgemeinen Register wie AX, BX, CX, DX, SI, DI, SP oder BP.
- In einem 8-Bit allgemeinen Register AH, BH, CH, DH, AL, BL, CL oder DL.

Codierbeispiele:

```
XCHG EDX,EDI      ; Vertausche 32-Bit-Register
XCHG DX,DI        ; Vertausche 16-Bit-Register
XCHG DL,CH        ; Vertausche 8-Bit-Register
```

1.5.2 Unmittelbare Adressierung

Bestimmte Befehle benützen Operanden, die im Befehl selbst enthalten sind. Solche Operanden werden als ***unmittelbare Operanden*** bezeichnet und können 32 Bits, 16 Bits oder 8 Bits lang sein.

Codierbeispiele:

Um die Variablen MUSTER_1, MUSTER_2 und MUSTER_3 zu testen, können folgende unmittelbare Operanden als Masken benützt werden.

```
TEST MUSTER_1,0EF000FF1H ; Benütze eine 32-Bit Maske
TEST MUSTER_2,0FED8H     ; Benütze eine 16-Bit Maske
TEST MUSTER_3,1EH        ; Benütze eine 8-Bit Maske
```

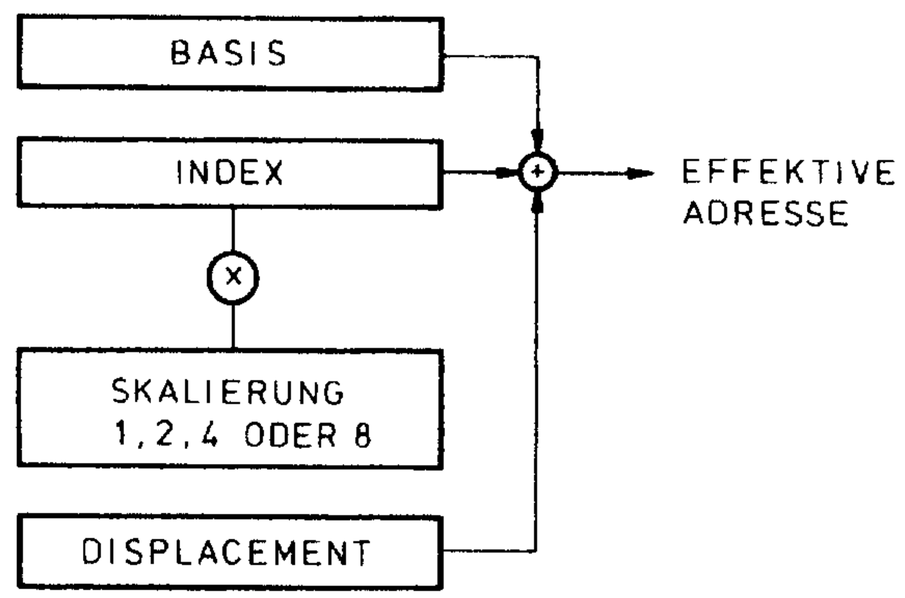
1.5.3 Speicheradressierung

Um die ***effektive Adresse*** eines Speicher-Operanden zu spezifizieren, werden die verbleibenden 9 Adressierungsarten benützt.

Die effektive Adresse stellt den resultierenden Offset eines Speicher-Operanden dar und ergibt sich durch beliebige Kombinationen der folgenden 4 Adreß-Elemente:

- Basis
- Index
- Skalierung
- „Displacement“

Die Bildung der effektiven Adresse ist im folgenden Bild illustriert:

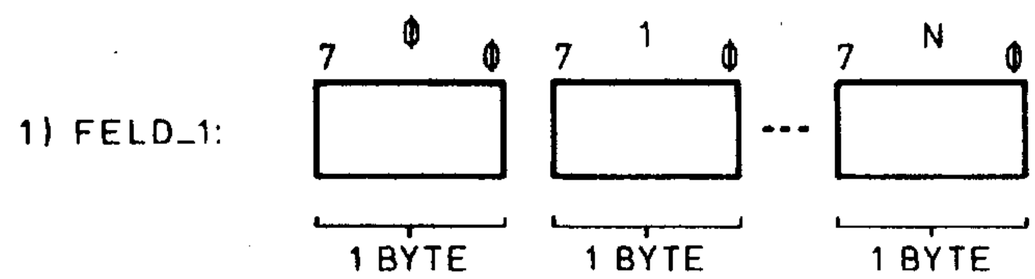


Basis:
Als Basis kann der Inhalt eines der folgenden allgemeinen Register benützt werden: EAX, ECX, EDX, EBX, ESP, EBP, ESI und EDI.
Das Basis-Register wird im allgemeinen dazu benützt, um den Beginn eines Feldes zu kennzeichnen.

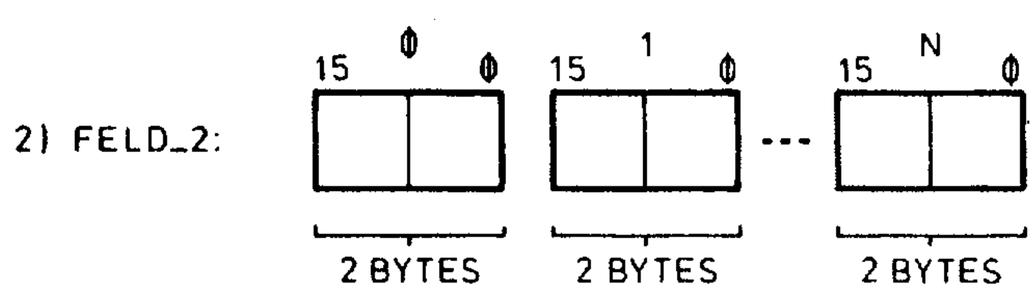
Index:
Als Index kann der Inhalt eines der folgenden allgemeinen Register benützt werden: EAX, ECX, EDX, EBX, EBP, ESI, EDI (***nicht*** ESP).
Das Index-Register wird im allgemeinen dazu benützt, um auf die einzelnen Elemente eines Feldes oder eines Strings zuzugreifen.

Skalierung:
Die Skalierung kann nur im Zusammenhang mit einem Index benützt werden. Dabei ist es möglich, den Wert eines Index-Registers mit einem der folgenden Skalier-Faktoren zu multiplizieren: 1, 2, 4 oder 8. Dieses Leistungsmerkmal ist speziell bei der Bearbeitung von Feldern nützlich. Jedes Feld ist aus einer begrenzten Anzahl von Elementen zusammengesetzt. Dabei ist jedes der Elemente durch eine spezifische Byte-Länge gekennzeichnet. Diese Länge bestimmt den sogenannten ***Typ*** der im Feld enthaltenen Elemente.

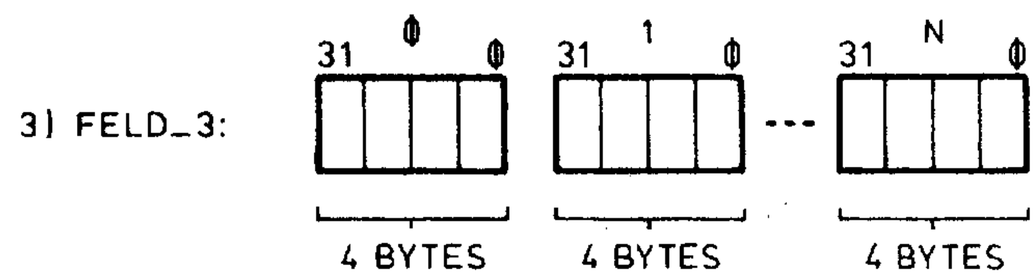
BEISPIELE:



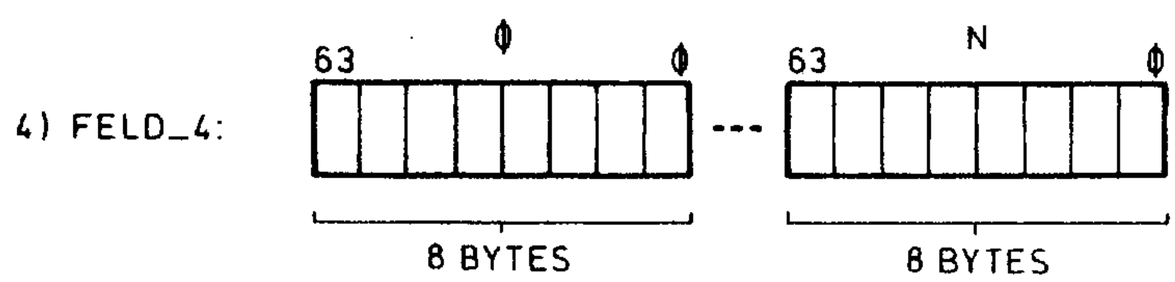
Dieses Feld enthält Elemente vom Typ ***Byte*** (1 Byte).



Dieses Feld enthält Elemente vom Typ ***Word*** (2 Bytes).



Dieses Feld enthält Elemente vom Typ ***Doubleword*** (4 Bytes).

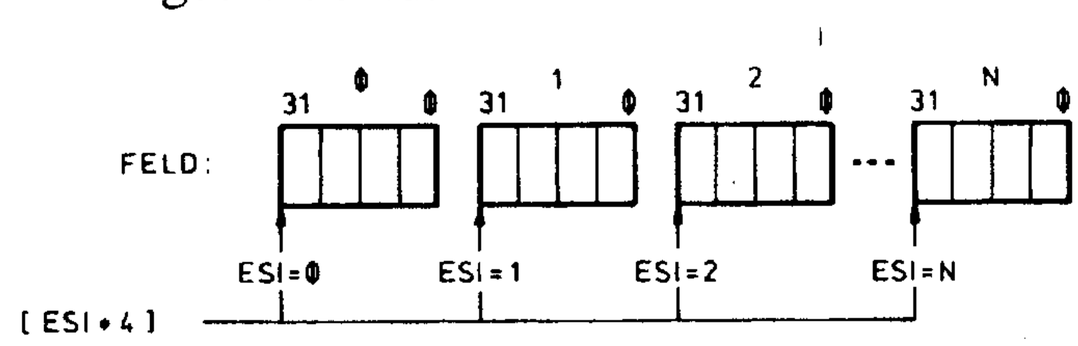


Dieses Feld enthält Elemente vom Typ ***Quadword*** (8 Bytes).

Um die einzelnen Elemente eines Feldes nacheinander zu erreichen, kann eines der oben angegebenen Index-Register zusammen mit einem Skalier-Faktor benützt werden.

BEISPIEL:

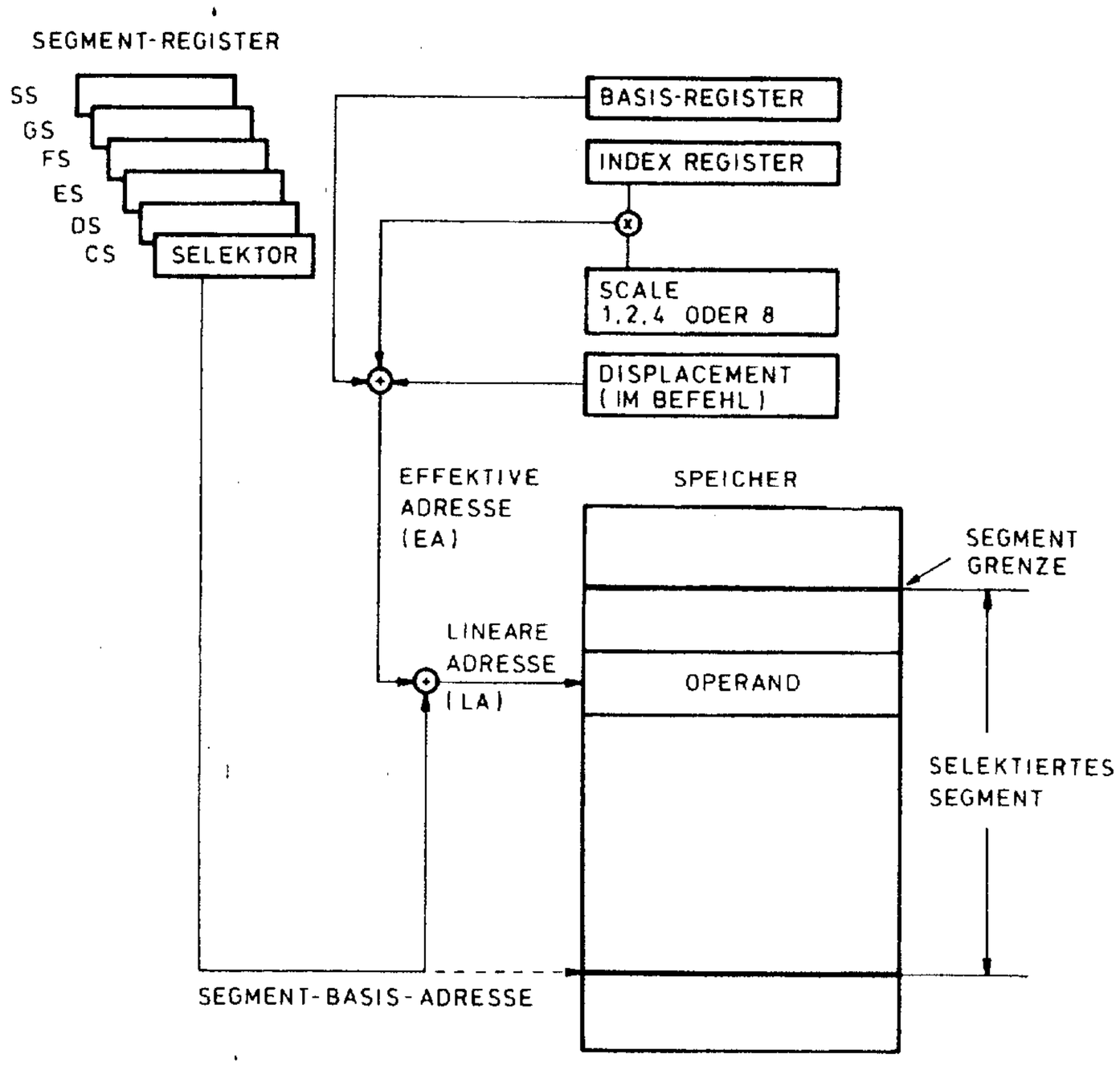
In einem Programm soll ein Feld bearbeitet werden, dessen Elemente aus 4 Bytes (Typ Doubleword) zusammengesetzt sind.



Wird nun zum Beispiel das Index-Register ESI und der Skalier-Faktor 4 (wegen Typ „Doubleword") benützt, so kann durch fortlaufendes Inkrementieren von ESI der Zeiger [ESI x 4] die Position der Einzelelemente erreichen und damit das gesamte Feld in bequemer Weise abtasten ①.

„Displacement“:
Das „Displacement“ ist ein 8-Bit- oder 32-Bit-Wert, der dem Befehl unmittelbar folgt. Dabei ist das Displacement identisch mit dem Offset einer Variablen relativ zum Beginn eines der Daten-Segmente (32-Bit-Displacement) oder relativ zum Beginn eines Daten-RECORDs (8 Bit-Displacement). 16 Bit-Displacements können ebenfalls benützt werden, wenn dem Befehl eine „Adress Length Prefix“ vorangestellt wird.

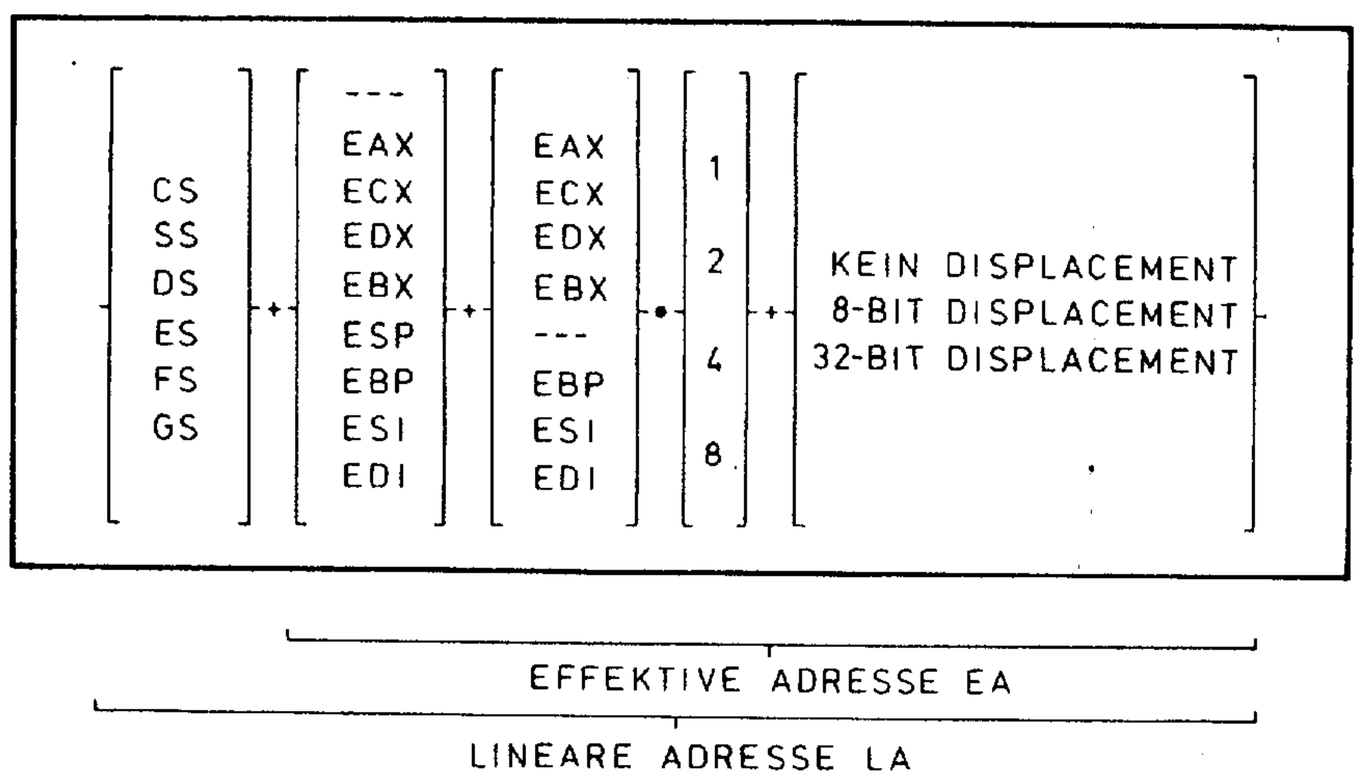
Das folgende Bild zeigt eine Analyse aller 80486-Speicher-Adressierungsarten:



Es ist ersichtlich, daß die effektive Adresse ***EA*** und die lineare Adresse ***LA*** nach folgenden Formeln berechnet werden:

- EA = BASIS-REGISTER + (INDEX-REGISTER × SCALE) + DISPLACEMENT
- LA = SEGMENT-BASIS-ADRESSE + EA

Abhängig davon, wie die Adreß-Elemente kombiniert werden, ergeben sich die bereits erwähnten 9 Adressierungsarten. Das angegebene Bild illustriert komplett die Kombinationsmöglichkeiten:



Welche Adressierungsformen sich dabei ergeben, zeigt beispielhaft die folgende Auflistung:

1.5.3.1 Direkt

Die effektive Adresse des Operanden ist im Befehl als 8-, 16- oder 32 Bit-Displacement enthalten.

Codierbeispiel:

MOV EBX,DWORD PTR [1000]
EA

1.5.3.2 Register-Indirekt

Die effektive Adresse des Operanden ist in einem Basis- ***oder*** Index-Register enthalten.

Codierbeispiel:

MOV EAX, [EDI]
EA

1.5.3.3 Based

Die effektive Adresse des Operanden wird gebildet, indem der Inhalt eines Basis-Registers zu einem Displacement addiert wird.

Codierbeispiel:

MOV EAX, [ESI + 10]
EA

1.5.3.4 Indexed

Die effektive Adresse des Operanden wird gebildet, indem der Inhalt eines Index-Registers zu einem „Displacement“ addiert wird.

Codierbeispiel:

MOV EAX,FELD [EDI]

EA

1.5.3.5 Scaled Indexed

Die effektive Adresse des Operanden wird gebildet, indem der Inhalt eines Index-Register mit einem Skalier-Faktor multipliziert und das Ergebnis zu einem „Displacement“ addiert wird.

Codierbeispiel:

MOV EAX,FELD [ESI ×2]

EA

1.5.3.6 Based Indexed

Die effektive Adresse des Operanden wird gebildet, indem der Inhalt eines Basis-Registers zum Inhalt eines Index-Registers addiert wird.

Codierbeispiel:

MOV EAX, [EDI][EBX]

EA

1.5.3.7 Based Scaled Indexed

Die effektive Adresse des Operanden wird gebildet, indem der Inhalt eines Index-Registers mit einem Skalier-Faktor multipliziert und das Ergebnis zum Inhalt eines Basis-Registers addiert wird.

Codierbeispiel:

MOV EAX, [EDX ×4][EBX]

EA

1.5.3.8 Based Indexed mit „Displacement“

Die effektive Adresse des Operanden wird gebildet, indem ein Displacement zu den Inhalten eines Index- und eines Basis-Registers addiert wird.

Codierbeispiel:

$$\mathbf{MOV\ EAX,\ \underbrace{[ESI][EDI + 5000]}_{EA}}$$

1.5.3.9 Based Scaled Indexed mit „Displacement“

Die effektive Adresse des Operanden wird gebildet, indem der Inhalt eines Index-Registers mit einem Skalier-Faktor multipliziert und das Ergebnis zum Inhalt eines Basis-Registers und zu einem Displacement addiert wird.

Codierbeispiel:

$$\mathbf{MOV\ EAX,\underbrace{FELD[ESI \times 4][ECX + 50]}_{EA}}$$

1.6 Segment-Register-Auswahl

Um in einem 80486-System den Speicher zu organisieren, wird eine Datenstruktur benützt, die als Segment bezeichnet wird. Dabei gibt es 3 Haupttypen von Segmenten:

- Code-Segment
- Daten-Segment
- Stack-Segment

Jedes dieser Segmente kann eine variable Länge im Bereich 1 Byte ... 4 Gigabytes haben.

Da Segmente die logische Struktur eines Programms widerspiegeln, müßten normalerweise in allen Befehlen die benützten Segment-Selektoren explizit spezifiziert werden. Aus Gründen der Geschwindigkeit und der Kompaktheit des Codes wählt aber der Prozessor bei bestimmten Speicherzugriffen die notwendigen Segment-Selektoren selbständig aus. Dies bedeutet, daß der Selektorteil ***nicht*** in jeder Adreßinformation enthalten sein muß.

Im allgemeinen benützen

- Daten-Referenzen den Selektor im DS-Register
- Stack-Referenzen den Selektor im SS-Register
- Befehle den Selektor im CS-Register.

Welche Segment-Register, abhängig vom Typ des Speicher-Zugriffs, der Prozessor automatisch (implizit) benützt, ist in der folgenden Tabelle zusammengefaßt:

Typ des Speicherzugriffs	Implizit benütztes Segment-Register	Mögliche Segment Override Prefixes
Code Fetch	CS	keine
Ziel von PUSH- und PUSHA-Befehlen	SS	keine
POP- und POPA-Befehle	SS	keine
Effektive Adressen, die folgende Basis-Register benützen:		
[EAX]	DS	CS, SS, ES, FS, GS
[EBX]	DS	CS, SS, ES, FS, GS
[ECX]	DS	CS, SS, ES, FS, GS
[EDX]	DS	CS, SS, ES, FS, GS
[EBX]	DS	CS, SS, ES, FS, GS
[ESI]	DS	CS, SS, ES, FS, GS
[EDI]*	DS	CS, SS, ES, FS, GS
[EBP]	SS	CS, DS, ES, FS, GS
[ESP]	SS	CS, DS, ES, FS, GS

***HINWEIS:**
Die STOS- und MOVS-Befehle benützen (E)DI als Basis-Register und ES für das Zielsegment. Eine „Segment Override Prefix" ist hierfür ***nicht*** möglich.

Muß während des Programmablaufs der Prozessor auf Segmente zugreifen, die er ***nicht*** implizit benützt, kann er spezielle ***Befehls-Prefix-Elemente*** verwenden. Eine Prefix bewirkt, daß für einen einzelnen Speicherzugriff die voreingestellte Segmentauswahl überschrieben wird („Segment Override"). Welche Prefix-Elemente möglich sind, ist ebenfalls in der angegebenen Tabelle zusammengefaßt.

1.7 Ausnahmen und Interrupts

Der 80486 kennt 2 Mechanismen, die die Ausführung eines Programms unterbrechen können: Interrupts und Ausnahmen.

1. ***Interupts*** sind ***asynchrone Ereignisse,*** die typischerweise von externen Geräten verursacht werden.

2. ***Ausnahmen*** sind ***synchrone Ereignisse,*** die durch bestimmte Bedingungen während einer Befehlsausführung vom Prozessor „entdeckt" werden.

Interrupts und Ausnahmen bewirken, daß das aktuelle Progamm kurzzeitig verlassen wird, um ein Programm höherer Priorität auszuführen.

Dabei sind Interrupts von der augenblicklichen Programmausführung völlig unabhängig, Ausnahmen dagegen reproduzierbar. Dies bedeutet, daß ein Programm, das eine Ausnahme verursacht, wiederholbar ist.

Die folgende Tabelle (Seite 32) zeigt eine Zusammenfassung aller 80486-Ausnahme-Bedingungen, die zur Unterbrechung eines Programms führen. Aus der Tabelle geht hervor, welche Ausnahmen für Anwender-Programmierer und welche für System-Programmierer von Interesse sind. An dieser Stelle und in den nachfolgenden Abschnitten 1.7.1 ... 1.7.8 sollen zunächst kurz die für den Anwender-Programmierer interessanten Ausnahmen beschrieben werden.

1.7.1 Divisionsfehler

Diese Ausnahme kann während der Ausführung eines ***DIV-*** (vorzeichenlose Division) oder eines ***IDIV-Befehls*** (Integer-Division) vorkommen, wenn z. B. durch 0 dividiert wird oder sich ein Quotienten-Überlauf ergibt. Da der EIP-Wert auf das erste Byte des Divisions-Befehls zeigt, muß vor der Rückkehr in die unterbrochene Prozedur der Rücksprung-Zeiger inkrementiert werden.

1.7.2 Einzelschritt

Der Prozessor verursacht eine Einzelschritt-Ausnahme nach jedem Befehl, wenn das TF-Bit (Trap Flag) gesetzt ist. Wenn er nun eine Interrupt-Prozedur aufruft, werden die EFLAGs in den Stack gerettet und das TF-Bit im CPU-Register zurückgesetzt, so daß es in der Handler-Prozedur zu keiner Einzelschritt-Ausnahme kommen kann. Erst wenn der Prozessor mit IRET den „Handler" verläßt, werden die EFLAGs und damit IF = 1 vom Stack in die CPU zurückgeschrieben. Die Handler-Prozedur für diese Ausnahme gehört typischerweise zu einem ***Debug-System,*** das den schrittweisen Test eines Programms erlaubt.

VEKTOR-NUMMER	AUSNAHME-BEDINGUNG	
0	DIVISIONS-FEHLER	ANWENDER-PROGRAMMIERER
1	EINZEL-SCHRITT	
2	NMI-INTERRUPT	SYSTEM-PROGRAMMIERER
3	BREAKPOINT-INTERRUPT (INT 3-BEFEHL)	ANWENDER-PROGRAMMIERER
4	OVERFLOW (INTO-BEFEHL)	
5	FELDGRENZE ÜBERSCHRITTEN (BOUND-BEFEHL)	
6	UNDEFINIERTER OPERATIONS-CODE	
7	PROZESSOR-ERWEITERUNG NICHT VORHANDEN	
8	DOPPEL-FEHLER	SYSTEM-PROGRAMMIERER
9	VOM 80486 NICHT BENUTZT	
10	UNGÜLTIGES 'TASK'-STATUS-SEGMENT	
11	SEGMENT NICHT VORHANDEN	
12	STACK-SEGMENT-ÜBERLAUF VORHANDEN	
13	ALLGEMEINE SCHUTZVERLETZUNG	
14	PAGE-FEHLER	
15	RESERVIERT	
16	FEHLER DER PROZESSOR-ERWEITERUNG	ANWENDER-PROGRAMMIERER
17	ALIGNMENT CHECK	SYSTEM-PROGRAMMIERER
18-31	RESERVIERT	

1.7.3 Breakpoint

Diese Ausnahme wird vom ***INT 3-Befehl*** verursacht und bietet die Möglichkeit, irgendwo in einem ausführbaren Segment einen Abbruchpunkt zu definieren. Dabei kann ein ***Debug-System*** in einem ausführbaren Segment einen INT 3-Befehl plazieren. Dieser Befehl verhindert dann die normale Ausführung des Segments, so daß der „Debugger" bei kritischen Punkten eines Programms Register-Inhalte, Variablen usw. anzeigen kann. Der EIP-Wert zeigt auf den nächsten Befehl. Hat der „Debugger" den INT 3-Befehl durch einen gültigen OpCode ersetzt, muß er vor der Rückkehr EIP mit 1 dekrementieren.

1.7.4 Overflow

Diese Ausnahme kommt vor, wenn das ***OF-Bit*** (Overflow-Flag) gesetzt ist und der Prozessor einen INTO-Befehl ausführt. Der EIP-Wert zeigt auf den nächsten Befehl.

1.7.5 Feldgrenze überschritten

Diese Ausnahme kommt immer dann vor, wenn der Prozessor den ***BOUND-Befehl*** ausführt und dabei einen Operanden jenseits der spezifischen Feldgrenze findet. Der EIP-Wert zeigt auf das erste Byte des fehlerhaften Befehls.

1.7.6 Undefinierter Operations-Code

Diese Ausnahme wird verursacht, wenn der Prozessor im Befehls-Strom einen undefinierten Operations-Code entdeckt. So kann es z. B. vorkommen, daß ein JMP-Befehl konstante Daten in einem ausführbaren Segment benützt.

1.7.7 Prozessor-Erweiterung nicht vorhanden

Diese Ausnahme kommt vor, wenn die Befehle der numerischen Prozessor-Erweiterung von einem Software-Emulator emuliert werden.

1.7.8 Fehler der Prozessor-Erweiterung

Diese Ausnahme wird von der numerischen Prozessor-Erweiterung verursacht, wenn sie auf eine der folgenden Bedingungen trifft:

- Ungültige Operation
- Overflow
- Division durch 0
- Underflow
- Denormalisierter Operand
- Unexaktes Ergebnis

1.8 Die physikalische Abbildung logischer Adressen

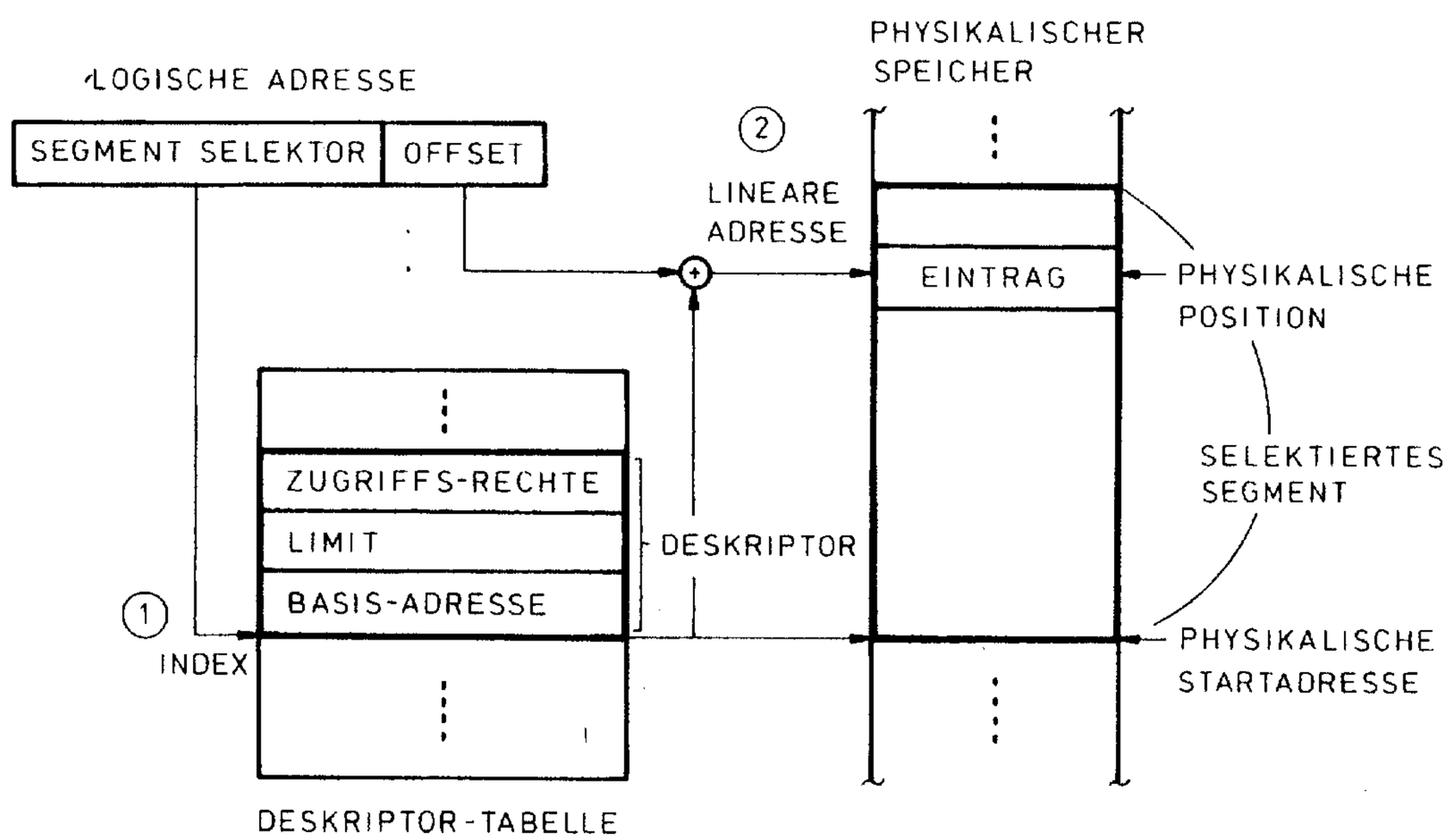
Die segmentierte Architektur des 80486 ermöglicht im sogenannten ***„Protected Mode"*** des Prozessors die Abbildung des logischen Adreßraums auf den physikalischen Adreßraum. Allerdings wird nicht der gesamte logische Adreßraum eines Programms, so wie ihn der Programmierer sieht, auf den physikalischen Adreßraum abgebildet, sondern nur ein logischer Teilraum, das sogenannte ***Segment***.

Die Abbildung erfolgt in diesem Fall über eine ***Segment-Tabelle,*** in der ***Segment-Deskriptoren*** für jedes Segment gespeichert sind.

Ein Segment-Deskriptor enthält die physikalische Startadresse, die Länge und die Zugriffsrechte eines Segments. Die Segment-Selektor-Komponente wird als Index benützt, um einen Segment-Deskriptor in der Segment-Tabelle auszuwählen ①.

Zur Startadresse des Segments, die im Segment-Deskriptor gespeichert ist, wird anschließend die effektive Adresse (Offset) addiert. Die resultierende lineare Adresse entspricht der physikalischen Position eines Eintrags im ausgewählten Segment ②.

Das angegebene Bild verdeutlicht diesen Mechanismus. Die Hardware kann nun bequem bei jedem Zugriff auf das Segment den Offset der logischen Adresse und den Typ des Zugriffs mit dem Limit-Wert und dem Zugriffsrecht im Deskriptor vergleichen. Dadurch verhindert sie z. B., daß in ein ***nur lesbares Segment*** jenseits der Segment-Obergrenze geschrieben wird.



1.9 Segment-Längen bei 32-Bit- und 16-Bit-Adressierung

Die lineare Adresse eines 80486-Operanden wird entweder mit

- 32 Bit-Offsets
  oder
- 16 Bit-Offsets

gebildet. Ausgewählt wird eine dieser Optionen durch das ASM386/486-Segment-Attribut USE. Dabei kann USE entweder USE16 oder USE32 sein. Jedes dieser Attribute hat direkten Einfluß sowohl auf die Größe der Adressen als auch auf die Größe der Operanden.

BEISPIEL:

```
CODE_1 SEGMENT USE32
        .
        .
        .
CODE_1 ENDS
```

Hier ist das logische ASM386/486-Segment CODE_1 als USE32-Segment spezifiziert. Dies bedeutet, daß in diesem Segment als ***Voreinstellung*** 32 Bit-Offsets generiert und 32 Bit-Operanden benützt werden.

BEISPIEL 2:

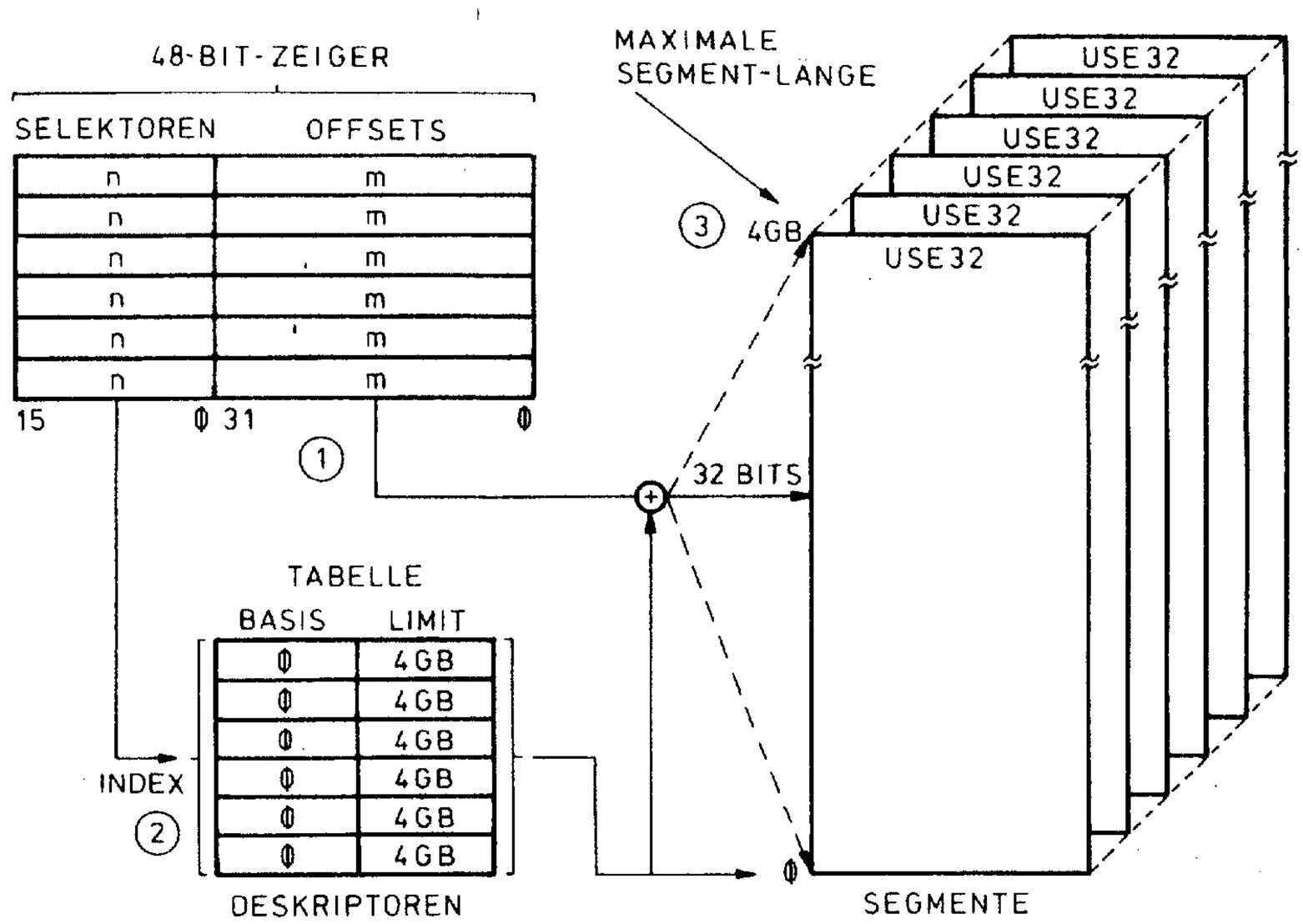
```
CODE_2 SEGMENT USE16
        .
        .
        .
CODE_2 ENDS
```

Hier ist das logische ASM386/486-Segment CODE_2 als USE16-Segment spezifiziert. Dies bedeutet, daß in diesem Segment als ***Voreinstellung*** 16 Bit-Offsets generiert und 16 Bit-Operanden benützt werden.

Die maximale Länge eines individuellen Segments wird durch die Wahl des USE-Attributs bestimmt. So kann ein USE16-Segment bis 64 KBytes (2^{16} Bytes) lang sein, während ein USE32-Segment bis zu 4 Gigabytes (2^{32} Bytes) lang sein kann.

Die folgenden beiden Bilder illustrieren die Unterschiede zwischen USE32- und USE16-Segmenten.

1.9.1 USE32-Segment

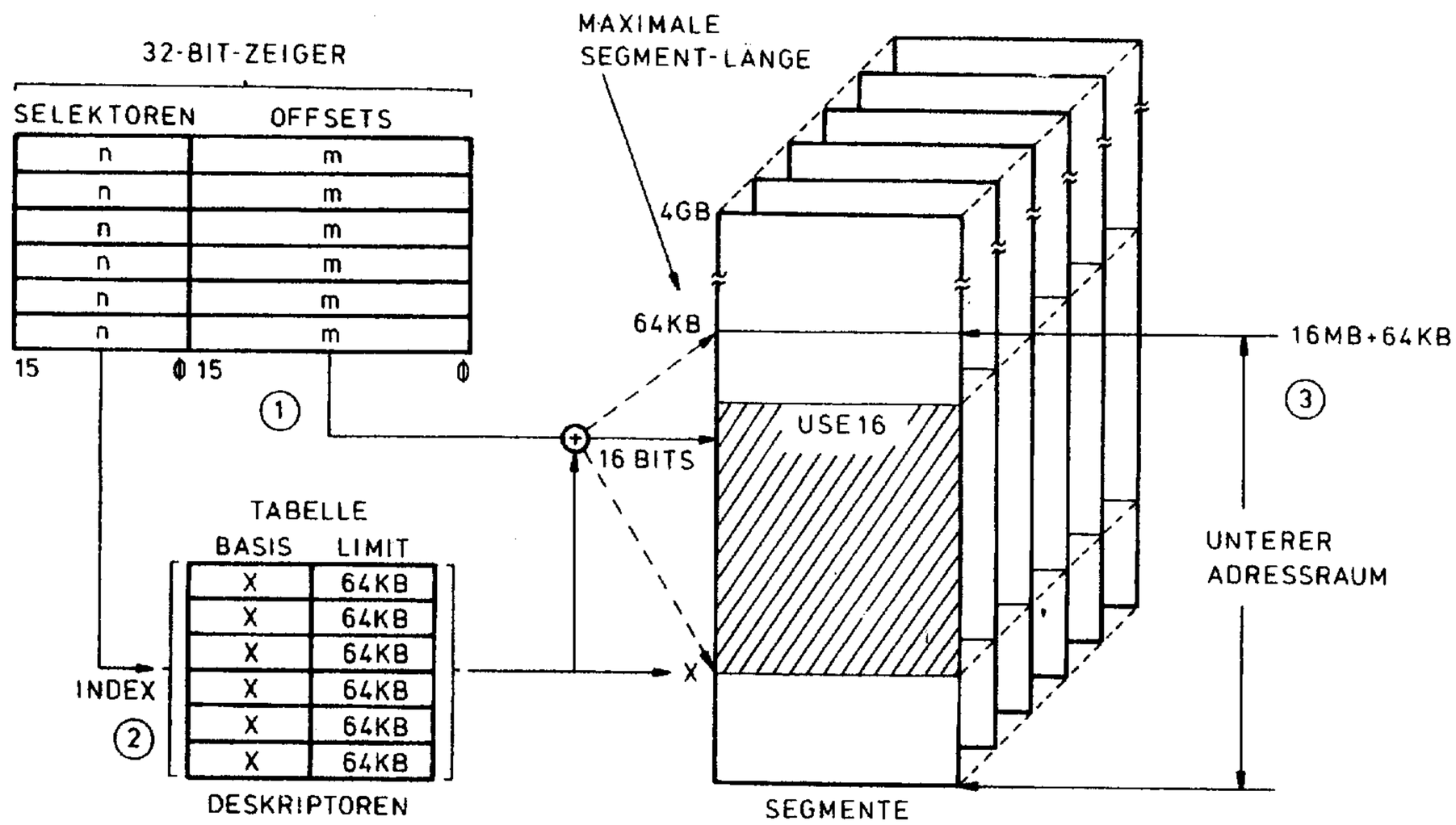


In einem USE32-Segment wird im Normalfall als logische Adresse der 48-Bit-Zeiger ***Selektor:Offset*** benützt. Dabei enthält der Selektor eine 16-Bit-Ordinalwert und der Offset einen 32-Bit-Ordinalwert ①. Der Selektor-Wert wird als Index benützt, der in einer Tabelle den Deskriptor eines Segments auswählt ②.

Die Basis-Adresse eines USE32-Segments wird durch einen ***32-Bit-Wert*** im Basis-Feld des ausgewählten Deskriptors bestimmt. Nur wenn das Basis-Feld den Wert 0 enthält, kann der 32-Bit-Offset ein Segment mit einer maximalen Länge von 4 Gigabytes adressieren ③.

Sollten in einem USE32-Segment Befehle mit 16-Bit-Offsets benützt werden, stellt ASM386/486 dem Befehl die ***„Adress Length Prefix“*** 67 H voran. In diesem Fall können dann im 4 Gigabyte physikalischen Adreßraum nur maximal 64 KByte adressiert werden.

1.9.2 USE16-Segment



In einem USE16-Segment wird im Notfall als logische Adresse der 32-Bit-Zeiger ***Selektor:Offset*** benützt. Dabei enthält der Selektor einen 16-Bit-Ordinalwert und der Offset ebenfalls einen 16-Bit-Ordinalwert ①. Der Selektor-Wert wird als Index, der in einer Tabelle den Deskriptor eines Segments auswählt ②.

Die Basis-Adresse X eines USE16-Segments wird durch einen ***24 Bit-Wert*** im Basis-Feld des ausgewählten Deskriptors bestimmt. Da das Basis-Feld nur Werte im Bereich 0 ... 00FFFFFFH annehmen kann, liegen USE16-Segmente im unteren (16 MByte + 64 KByte)-Raum des 4 Gigabyte physikalischen Adreßraums ③.

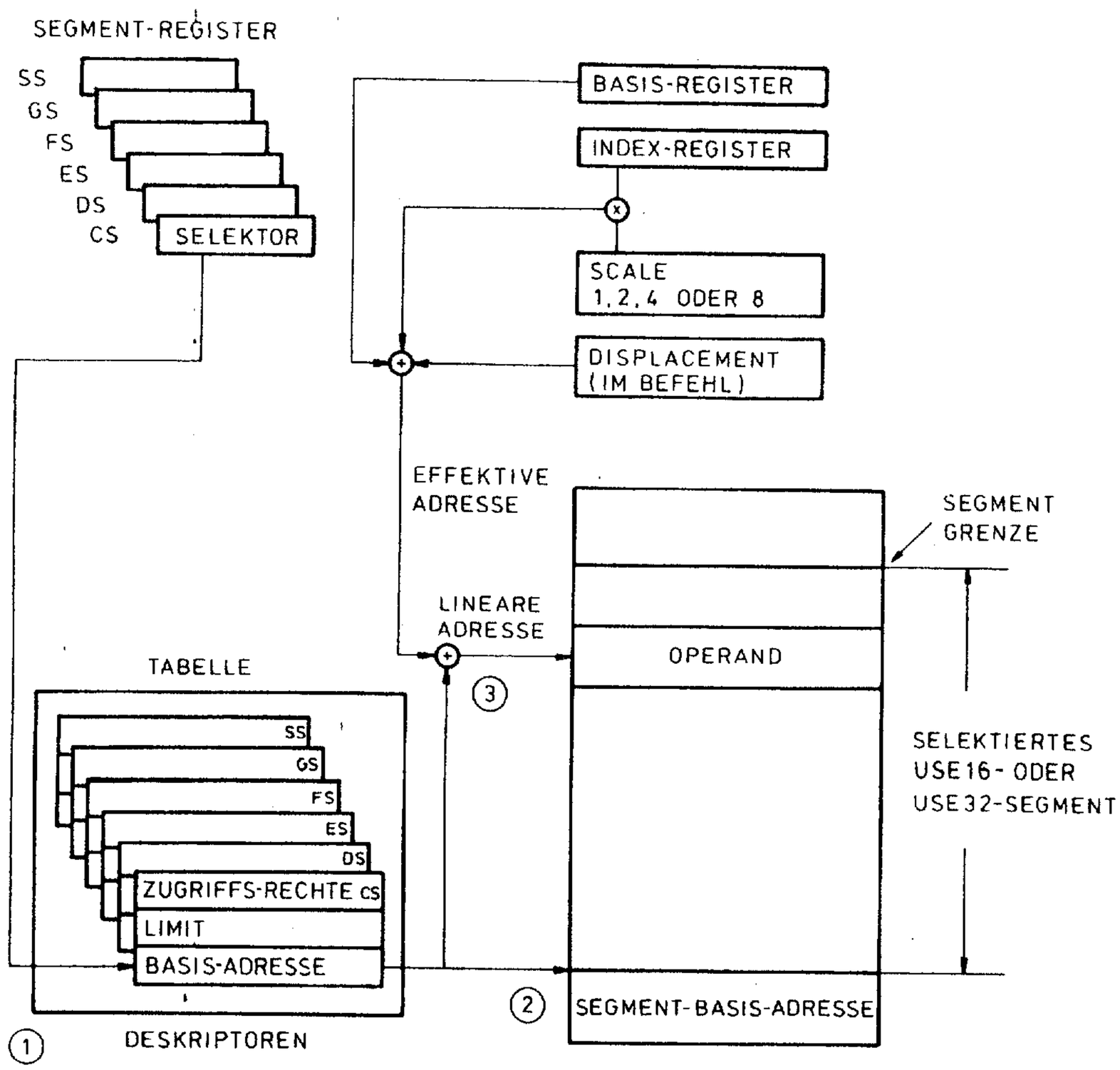
Sollten in einem USE16-Segment Befehle mit 32-Bit-Offsets benützt werden, stellt ASM386/486 dem Befehl die ***„Address Length Prefix“*** 67H voran.
Um in diesem Fall die 64 KByte-Segment-Obergrenze ***nicht*** zu überschreiten, darf der 32-Bit-Offset nur den Maximalwert 0000FFFFH liefern.

Wenn im „Protected Mode“ des Prozessors Deskriptoren benützt werden, um die logischen USE16- oder USE32-Programm-Segmente auf bestimmte physikalische Speicherbereiche abzubilden, werden die effektiven Adressen logischer Operanden in exakt gleicher Weise gebildet, wie sie im Abschnitt „Speicheradressierung“ bereits beschrieben worden sind.

Die Inhalte der Segment-Register SS, GS, FS, ES, DS und CS werden allerdings nicht mehr als physikalische Basis-Adresse der Segmente interpretiert. Statt dessen enthalten die Segment-Register Selektoren, die einen bestimmten Deskriptor in einer Deskriptor-Tabelle auswählen ①.

Anschließend liefert der ausgewählte Deskriptor die physikalische Basis-Adresse des Segments, in dem sich der logische Operand aufhält ②.

Die lineare Adresse des Operanden ergibt sich auch hier durch einfache Addition der physikalischen Segment-Basis-Adresse und der effektiven Adresse ③.



1.10 Das „Paging"-Konzept

Im „Protected Mode" des 80486 ist der ***logische Adreßraum,*** wesentlich größer als der ***physikalische Adreßraum***. Dies bedeutet, daß jedem Benutzer ein „unechter", sogenannter ***virtueller Adreßraum*** zugewiesen werden kann, der von 0 ... 64 Terabytes reicht. Virtuelle Adressen können also Werte annehmen, die wesentlich über dem obersten Wert des physikalischen Speichers von 4 Gigabytes liegen.

Der virtuelle Adreßraum eines Benutzers muß von einer ***„Memory Management"-Einheit (MMU)*** auf den physikalischen Speicherbereich abgebildet werden. Zu diesem Zweck stehen geeignete Deskriptoren zur Verfügung. Da der virtuelle Adreßraum größer ist als die Ausdehnung des physikalischen Arbeitsspeichers, muß ein sekundäres Speichermedium (z. B. Massenspeicher) zur Verfügung stehen, um den virtuellen Adreßraum aufzunehmen.

Bei jeder Adreßreferenz wird nun geprüft, ob sich der benötigte Programmbereich bereits im physikalischen Speicher befindet. Ist dies nicht der Fall, wird ein Transfer vom Massenspeicher in den Arbeitsspeicher veranlaßt.

Beim Verwalten eines virtuellen Speichers gibt es verschiedene Verfahren, die Speicherzuteilung abzuwickeln. Dabei zeigt die Erfahrung, daß immer nur für eine kurze Zeit die CPU einen kleinen Teil des Programms bearbeiten muß. Dies bedeutet, daß in den ***seltensten Fällen*** die gesamte Code- und Daten-Region eines Programms im physikalischen Arbeitsspeicher benötigt wird.

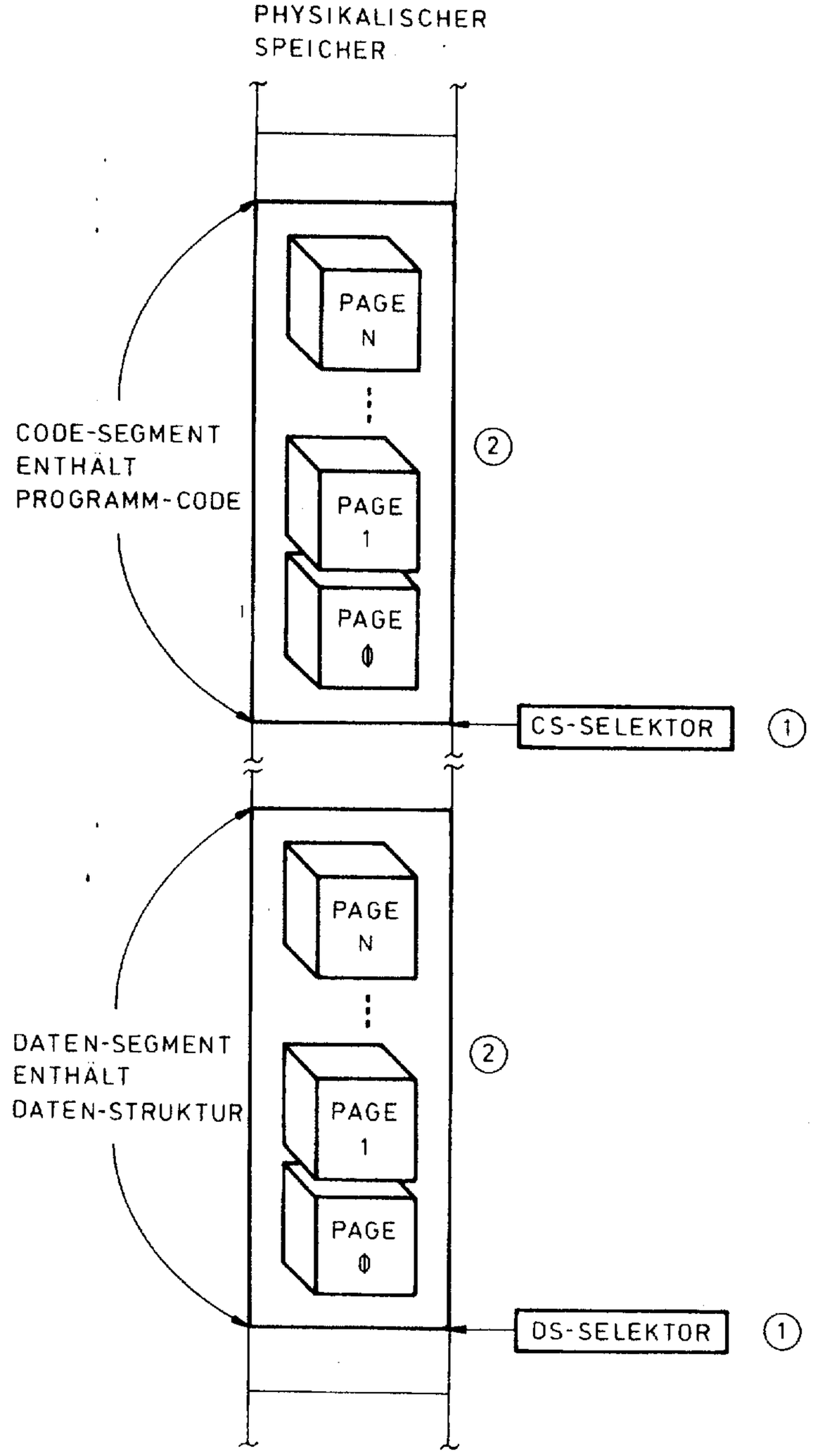
Würden alle Teile eines Programms in den Arbeitsspeicher geladen und wieder ausgelagert („swapping"), liegen die Nachteile auf der Hand:

- Es würde Speicherplatz vergeudet für Progammteile, die momentan ***nicht*** bearbeitet werden.
- Es wäre zusätzliche Zeit für das Einlagern nichtbenötigter Programmteile erforderlich.

Zur Lösung dieser Probleme stellt der 80486 einen Mechanismus zur Verfügung, mit dessen Hilfe der physikalische Raumbedarf eines Segments im Speicher ***verfeinert*** werden kann.

Dieser Mechanismus unterteilt ein physikalisches Segment in viele kleine Abschnitte mit jeweils starrer Länge, die als ***Pages*** bezeichnet werden.

Dabei haben die Pages keine direkte Beziehung zur logischen Struktur eines Programms. Während ein Segment-Selektor als logischer Name eines Programm-Codes oder einer Daten-Struktur betrachtet werden kann ①, korrespondiert eine Page immer nur mit einem kleinen Teil des Programm-Codes oder der Daten-Struktur ②.

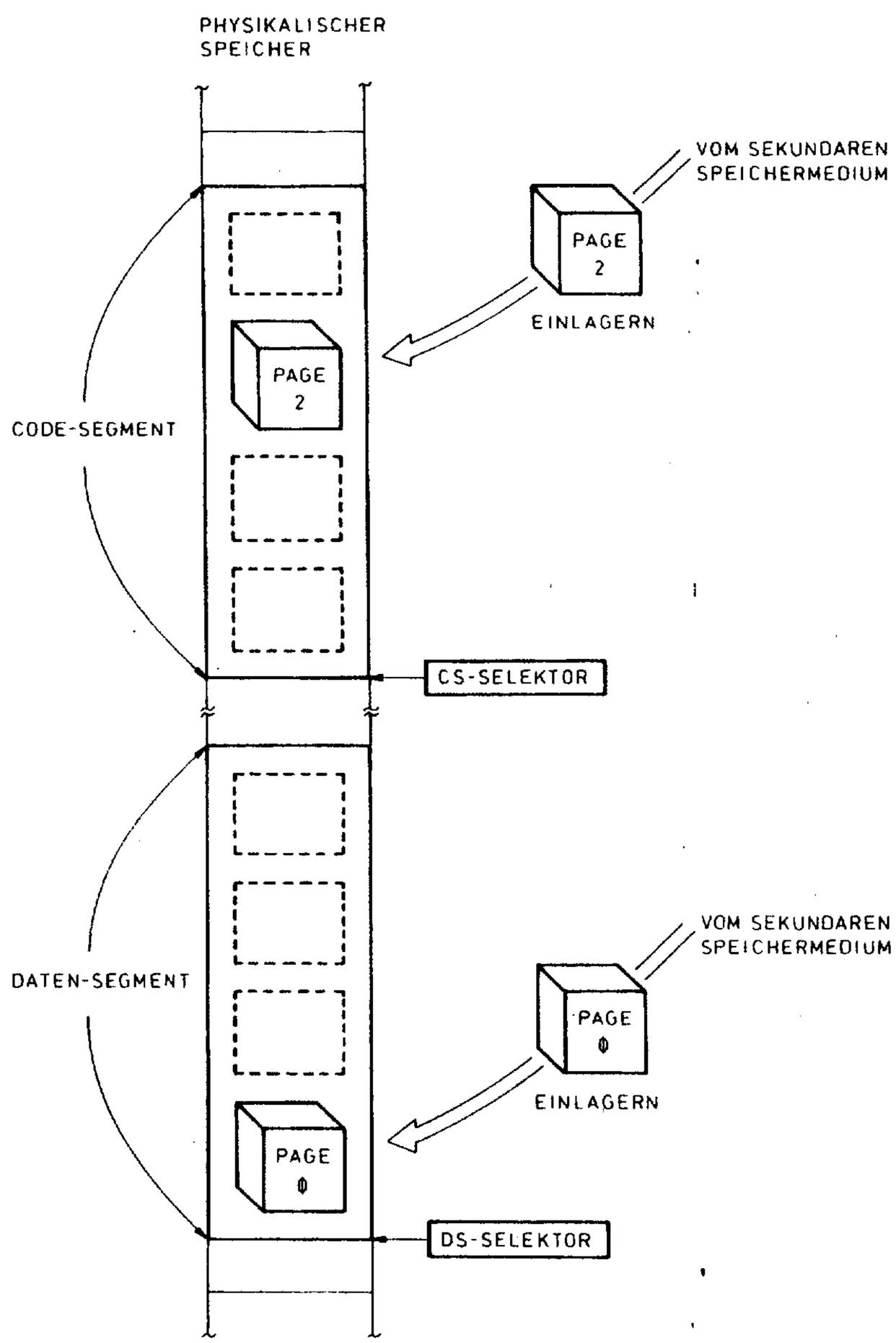


Das Paging wird also benützt, um den physikalischen Speicher zu verwalten, während die Segmentierung die wirkungsvolle Behandlung der logischen Struktur eines Programms unterstützt.

Wenn nun während der Programmausführung eine logische Adresse, so wie sie der Programmierer sieht, von der MMU auf den physikalischen Speicher abgebildet wird, überprüft der 80486, ob sich die zugehörige Page bereits im physikalischen Arbeitsspeicher befindet.

Wenn nicht, muß eine Betriebssystem-Routine die erforderliche Page vom sekundären Massenspeicher lesen und in den physikalischen Arbeitsspeicher einlagern.

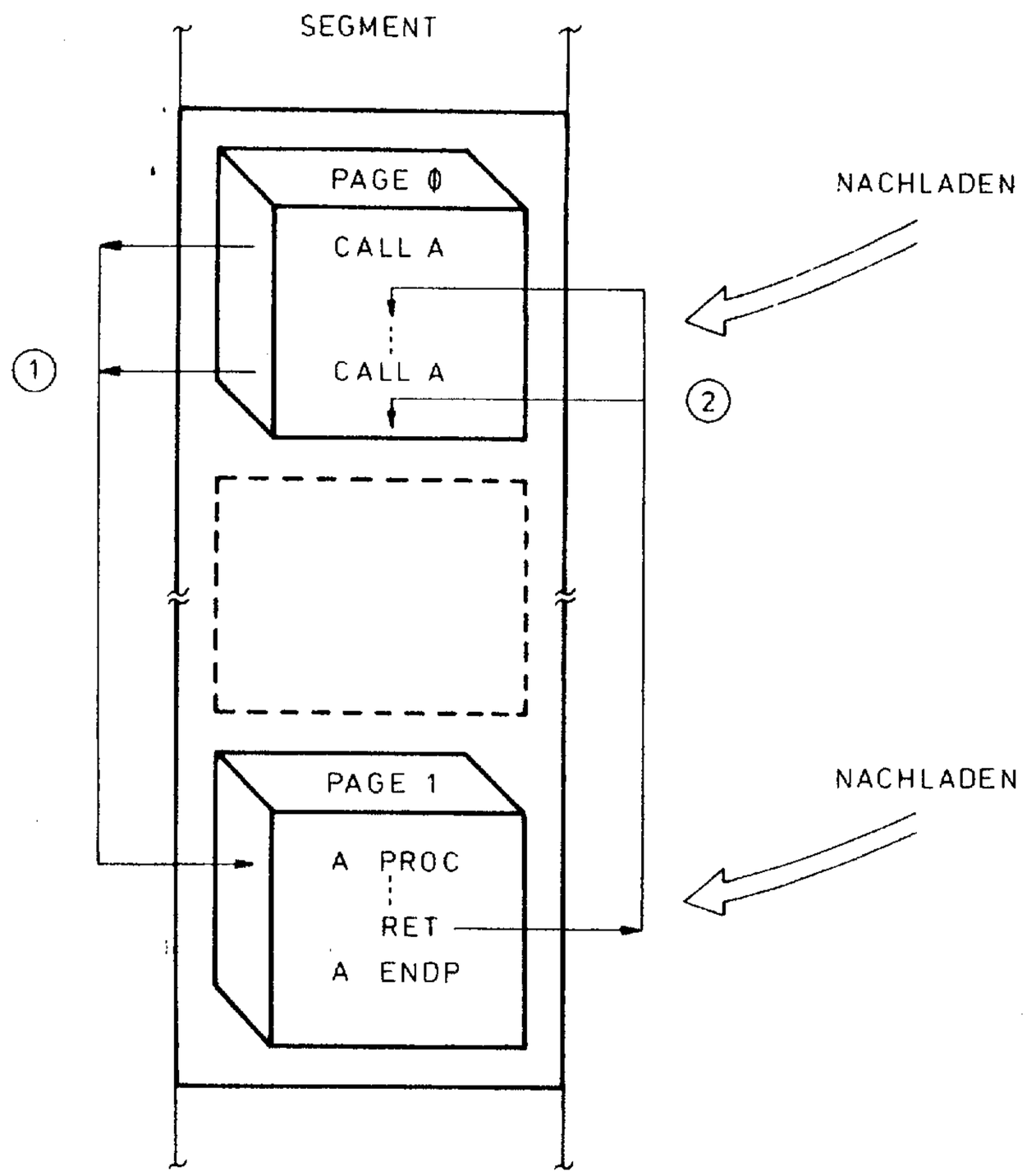
Diesen Vorgang verdeutlicht das folgende Bild:



Da also Pages nur auf Anforderung eingelagert werden, wird der Vorgang manchmal als ***„Demand Paging"*** bezeichnet.

Obwohl durch den Paging-Mechanismus der physikalische Raum eines Segments verkleinert werden kann, treten auch hier einige Probleme auf. Ist nämlich z. B. die letzte Seite (Page) ***nicht*** komplett ausgefüllt, hat man auch beim Paging Zeit und Speicherplatz verschwendet.

Außerdem kann bei einer ungünstigen Aufteilung eines Programms auf verschiedene Seiten das berüchtigte *„Seitenflattern“* auftreten. Dies soll das folgende Bild verdeutlichen:

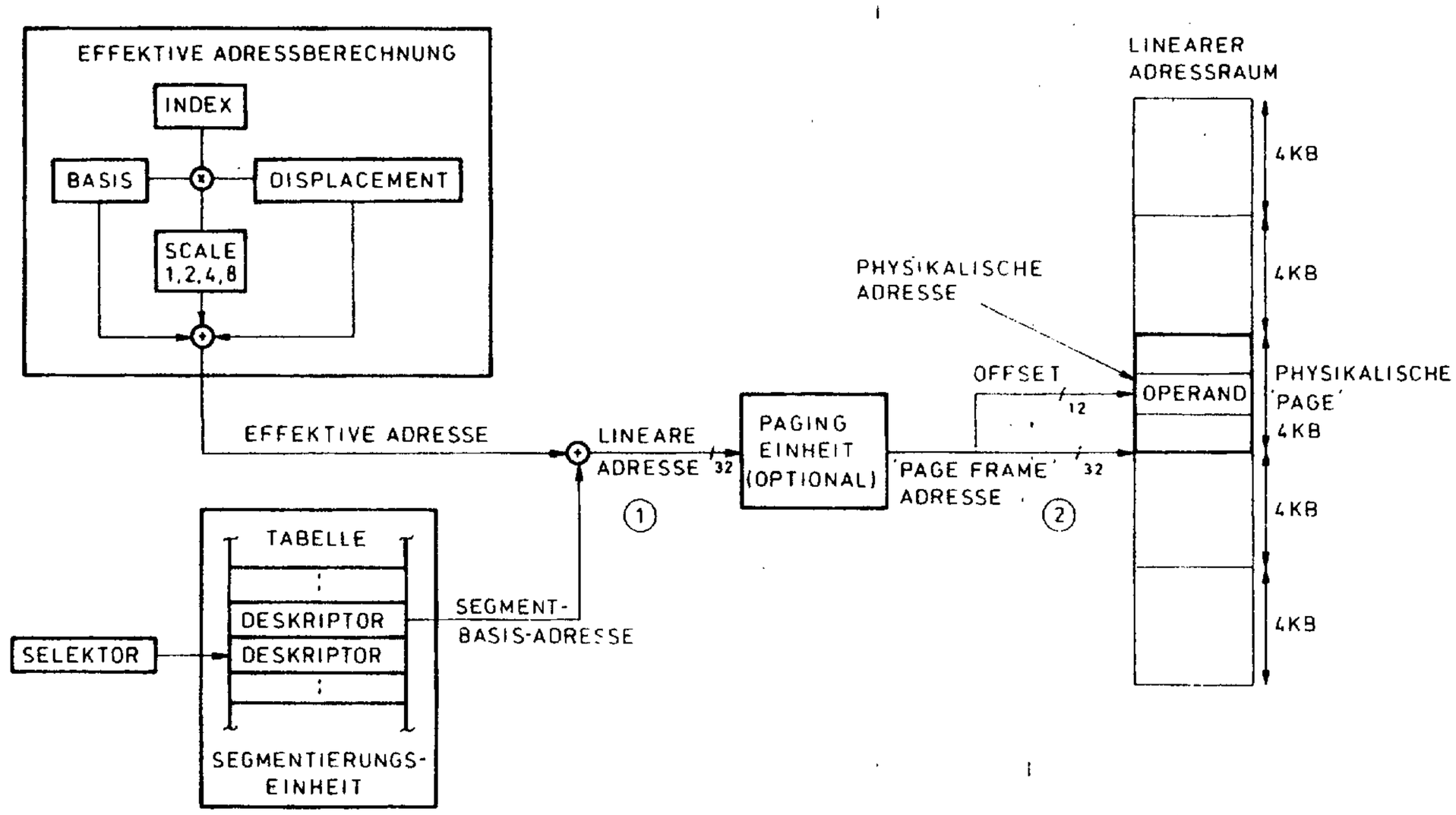


Es zeigt, daß in der PAGE 0 mehrmals mit CALL A die Prozedur A aufgerufen wird, die sich in der PAGE 1 aufhält ①. Umgekehrt erfolgt nach jedem Aufruf der Prozedur A am Ende mit RET die Rückkehr zur PAGE 0 ②. Es wird also andauernd zwischen den Pages hin und her gesprungen. Dies führt dazu, daß die beteiligten Pages ständig vom Massenspeicher nachgeladen werden müssen („Seitenflattern“).

Um einen Kompromiß zwischen der möglichst verlustlosen Speicheraufteilung und der Reduzierung der Wahrscheinlichkeit des Seitenflatterns einzugehen, hat man eine Seitengröße von 4096 Bytes (4 KBytes) gewählt.

Das beschriebene Paging-Konzept ist ***optional*** und steht nur im „Protected Mode“ des Prozessors zur Verfügung. Wenn es benützt wird, um die sehr großen Segmente (maximal 4 Gigabytes) des 80486 zu managen, übersetzt der Paging-Mechanismus des Prozessors die von der Segmentierungs-Einheit gelieferte ***lineare*** Adresse in eine ***physikalische*** Adresse.

Das folgende Bild illustriert den Zusammenhang zwischen „Paging" und „Segmentierung":



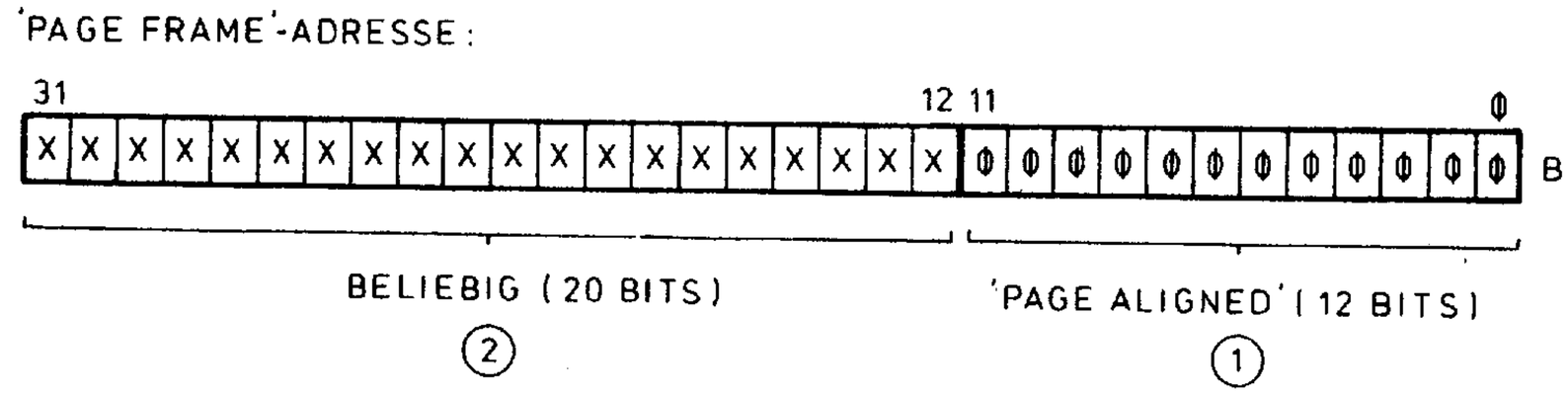
Ist der Paging-Mechanismus freigegeben, gibt die CPU die 32-Bit lineare Adresse eines Operanden, die aus der Segment-Basis-Adresse und der effektiven Adresse zusammengesetzt ist, ***nicht*** mehr an den Adreßbus aus. Statt dessen wird die lineare Adresse der Paging-Einheit zugeführt ①, die daraus zwei Adreßkomponenten ableitet, und zwar

- eine 32-Bit Page-Frame-Adresse und
- einen 12-Bit-Offset ②.

Dabei legt die Page-Frame-Adresse die physikalische Basis-Position einer Page innerhalb des linearen Adreßraums fest.

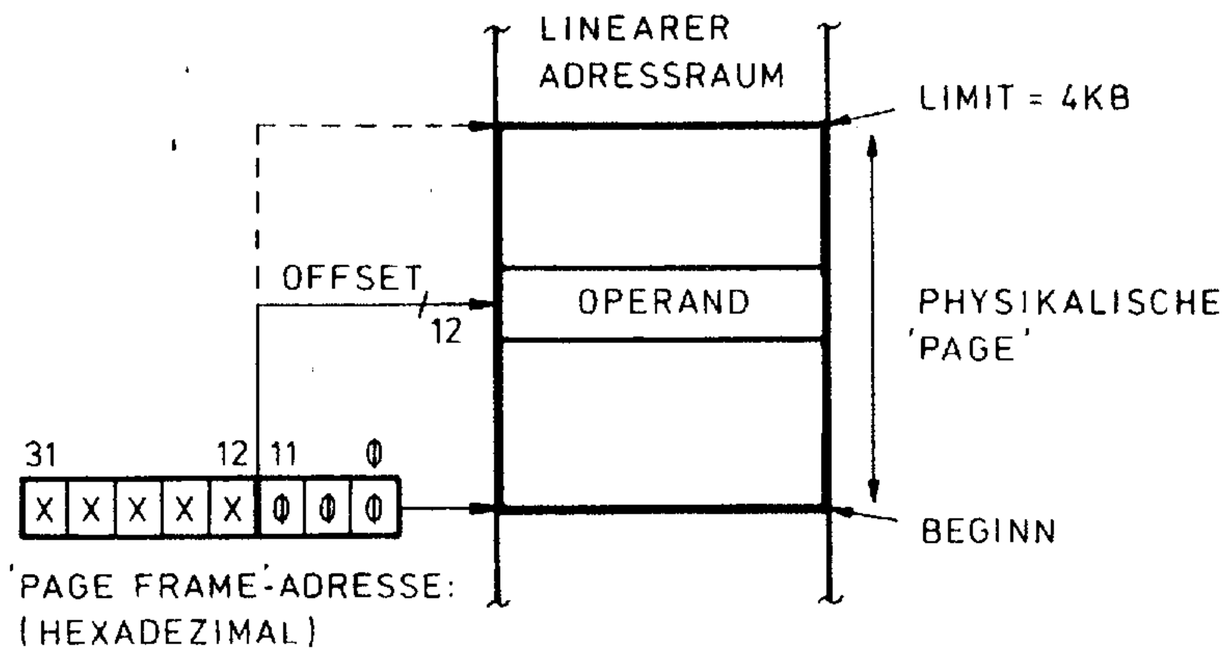
Da alle Pages eine Länge von 4 KByte haben, müssen sie an Adressen ausgerichtet sein, die durch 4 KByte teilbar sind. Die Ausrichtung eines Daten-Blocks an solche Adressen wird als ***„page aligned"*** bezeichnet.

Um dies zu erreichen, liefert die Paging-Einheit für jede Page eine 32-Bit-Frame-Adresse, deren niederwertige 12 Bits (0 ... 11) mit Nullen aufgefüllt sind ①.



Da die Adress-Bits 12 ... 31 beliebig sein können, kann man die logischen Seiten eines Programms auf irgendeine physikalische Seite im Speicher abbilden ②.

Um nun die physikalische Position eines Operanden innerhalb einer 4-KByte-Page zu erreichen, ist nur noch ein Offset von 12 Bits im Bereich von 000 ... FFFH erforderlich. Dies verdeutlicht das folgende Bild.:



1.11 Befehlsformat

Alle Befehls-Codierungen des 80486 sind Untermengen eines allgemeinen Befehlsformats, wie es das folgende Bild zeigt:

80486-BEFEHLSFORMAT

ELEMENT	BEFEHLS-PREFIX (LOCK ODER REP)	ADDRESS SIZE PREFIX (67H)	OPERAND SIZE PREFIX (66H)	SEGMENT OVERRIDE PREFIX	OPERATIONS-CODE	MODRM BYTE	SIB BYTE	DISPLACEMENT	UNMITTEL-BARER WERT
ANZAHL VON BYTES	0 ODER 1	0 ODER 1	0 ODER 1	0 ODER 1	1 ODER 2	0 ODER 1	0 ODER 1	0, 1, 2 ODER 4	0, 1, 2 ODER 4

Es ist ersichtlich, daß die Befehle optional eine Reihe von Prefix-Bytes enthalten können, wie

- Befehls-Prefix
- „Address Size Prefix“
- „Operand Size Prefix“
- „Segment Override Prefix“

Weiterhin können die Befehle

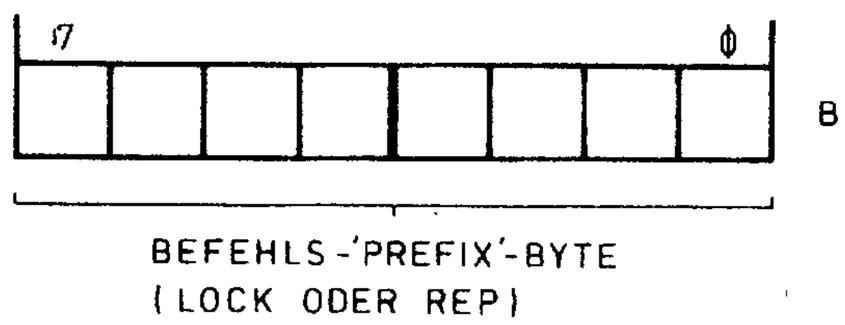
- ein oder zwei primäre Operations-Code-Bytes enthalten.

Für die verschiedenen Formen der effektiven Adressierung von Operanden steht ein ***Adreß-Spezifizierer*** zur Verfügung, der aus folgenden Bytes zusammengesetzt sein kann:

- ModRM-Byte (erstes Adressierungs-Byte)
- SIB-Byte = ***S***cale ***I***ndex ***B***ase (zweites Adressierungs-Byte)
- Displacement (bis zu 4 Bytes).

Benützt ein Befehl einen unmittelbaren Operanden (bis zu 4 Bytes), dann folgt der Wert dieses Operanden irgendeinem Displacement-Byte. Ein unmittelbarer Operand ist also immer im letzten Feld des Befehls untergebracht.

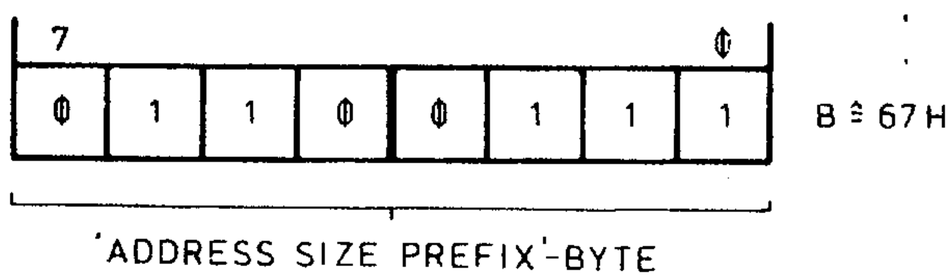
1.11.1 Befehls-Prefix



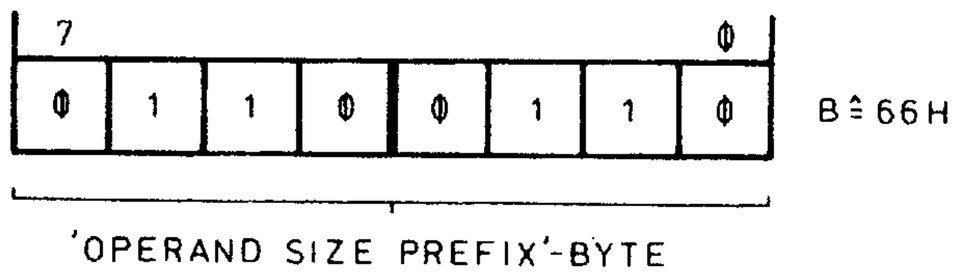
Die folgende Zusammenstellung zeigt die erlaubten Befehls-Prefix-Codierungen in ASM386/486 und ihre hexadezimalen Byte-Werte:

ASM386/486 Befehls-Prefix	Anwendung	Hexadezimaler Wert
REP	Wird nur bei String-Befehlen benützt	F3H
REPE/REPZ		F3H
REPNE/REPNZ		F2H
LOCK	Für die Dauer der Befehlsausführung in einer Multi-prozessor-Umgebung bleibt der Multibus für andere Prozessoren gesperrt	FOH

1.11.2 „Address Size Prefix“



1.11.3 „Operand Size Prefix“



Auf die Bedeutung und Wirkung diese beiden Prefix-Bytes ist bereits im Kapitel „Real Mode-Architektur“ hingewiesen worden. Ein 80486-Befehl kann jedoch diese Prefix-Bytes ***auch*** im „Protected Mode“ enthalten. Ihre allgemeine Wirkung soll nun an dieser Stelle erläutert werden.

Beide Prefix-Bytes erlauben, individuell die durch die Segment-Attribute USE16 oder USE32 voreingestellte Auswahl der Operanden- und der effektiven Adreßgrößen (16 Bits durch USE16 oder 32 Bits durch USE32) aufzuheben. Dabei ist ihre Wirkung nur bei denjenigen Befehlen gegeben, denen ein oder beide Prefix-Bytes vorangestellt sind.

Die Präsens der „Address Size Prefix" und der „Operand Size Prefix" hat also zur Folge, daß die voreingestellten Operanden- und Adreßgrößen auf „entgegengesetzte" Werte umgeschaltet werden.

BEISPIELE:

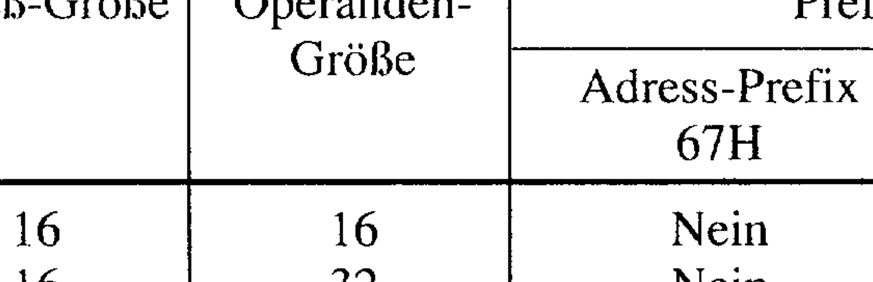
1. Ist die Operandengröße für 32-Bit-Datenoperationen voreingestellt, schaltet die „Operand Size Prefix" den Befehl auf 16-Bit-Datenoperationen um.

2. Ist die effektive Adreßgröße für 16 Bits voreingestellt, bringt die „Address Size Prefix" den Befehl dazu, daß er die 32 Bit effektive Adreßberechnung benützt.

   Die folgende Tabelle zeigt, abhängig vom gewählten USE-Attribut und den vom Programmierer explizit benützten Operanden- und Adreßgrößen, welche Prefix-Bytes derAssembler ASM 386/486 für das augenblickliche Code-Segment generiert:

USE-Attribut des augenblicklichen Code-Segments	Benützte Adreß-Größe	Benützte Operanden-Größe	Generiert ASM386/486 Prefix-Bytes?	
			Adress-Prefix 67H	Oberhand-Prefix 66H
USE16	16	16	Nein	Nein
USE16	16	32	Nein	Ja
USE16	32	16	Ja	Nein
USE16	32	32	Ja	Ja
USE32	32	32	Nein	Nein
USE32	32	16	Nein	Ja
USE32	16	32	Ja	Nein
USE32	16	16	Ja	Ja

1.11.4 „Segment Override Prefix"

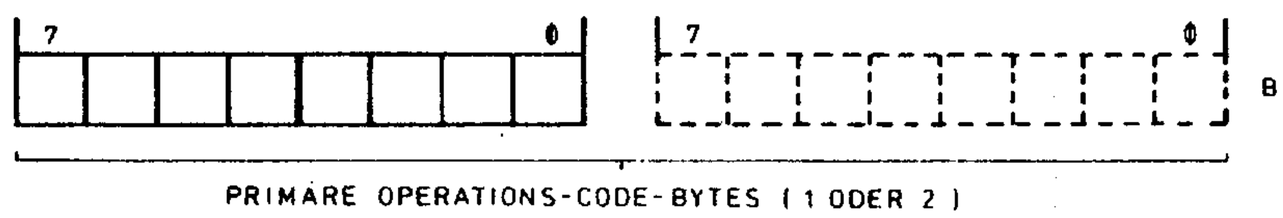


Die folgende Zusammenstellung zeigt die „Segment Override Prefix"-Codierung in ASM386/486 und ihre hexadezimalen Byte-Werte:

ASM386/486 Segment-Override-Prefix	Anwendung	Hexadezimeter Wert
CS:	Speicherzugriff relativ zum CS-Register	2EH
SS:	Speicherzugriff relativ zum SS-Register	36H
DS:	Speicherzugriff relativ zum DS-Register	3H
ES:	Speicherzugriff relativ zum ES-Register	26H
FS:	Speicherzugriff relativ zum FS-Register	64H
GS:	Speicherzugriff relativ zum GS-Register	65H

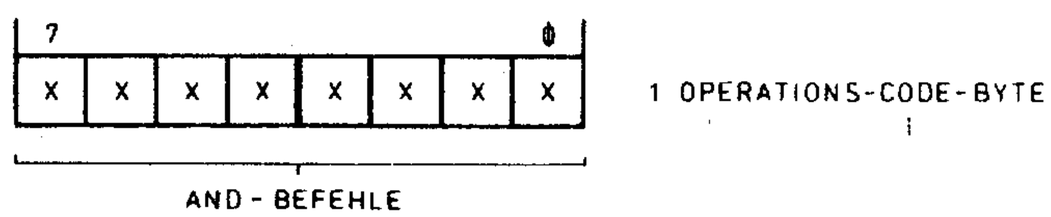
HINWEIS:
Welche „Segment Override Prefix" abhängig vom Typ des Speicherzugriffs in einem Befehl benutzt werden darf, ist im Kapitel „Segment-Register-Auswahl" beschrieben und tabellarisch zusammengefaßt.

1.11.5 Operations-Code



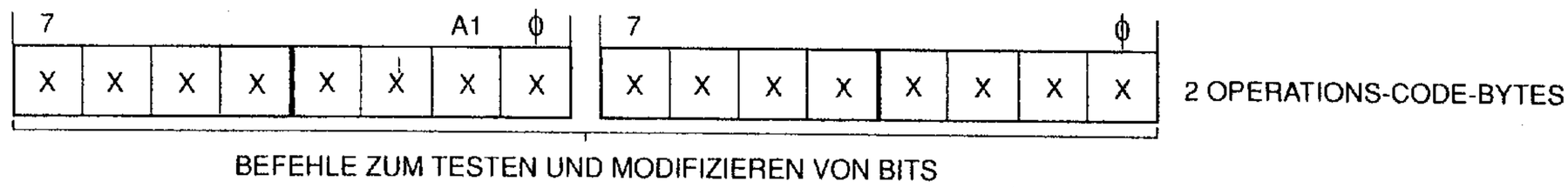
Diese Felder im 80486-Befehlsformat spezifizieren die ***Klasse der Operationen***. Wie das Bild zeigt, kann der primäre Operations-Code eines Befehls entweder aus 1 oder 2 Byte bestehen.

Beispiel 1:



Die Klasse der AND-Operationen besteht aus ***einem*** primären Operations-Code-Byte.

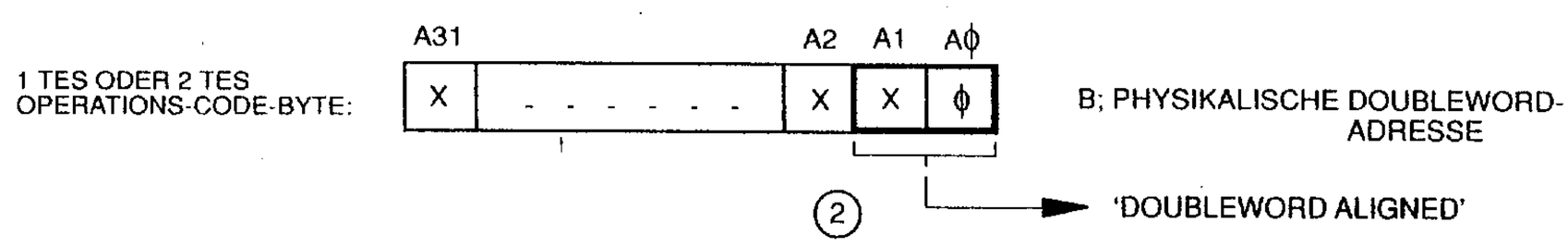
Beispiel 2:



Die Klasse der Befehle, die zum Testen und Modifizieren einzelner Bits benützt werden, besteht aus *zwei* primären Operations-Code-Bytes.

Alle Operanden, die der 80486 unterstützt, basieren auf den fundamentalen Datentypen ***Byte, Word, Doubleword.*** .

Immer dann, wenn ein Befehl auf einen Operanden trifft, muß der Prozessor den Typ des Operanden identifizieren. Zu diesem Zweck steht im ersten oder im zweiten primären Operations-Code-Byte das ***Typ-Identifikations-Bit w*** zur Verfügung. Es bestimmt, ob es sich um eine Byte- oder Word- bzw. Doubleword-Operation handelt ①.



Wie das angegebene Bild zeigt, identifiziert w = 1 sowohl Word- als auch Doubleword-Operanden. Damit aber diese beiden Operanden-Typen unterschieden werden können, wird als weiteres Identifikations-Kriterium das USE-Attribut des Segments benützt.

So gibt in einem USE32-Segment w = 1 an, daß 32-Bit-Operanden verarbeitet werden, während in einem USE16-Segment w = 1 anzeigt, daß 16-Bit-Operanden verarbeitet werden.

Sollten aber in einem USE32-Segment 16-Bit-Operanden oder umgekehrt in einem USE16-Segment 32-Bit-Operanden benützt werden, ist in beiden Fällen das w-Feld mit 1 besetzt. Die Identifikation des Operanden-Typs übernimmt nun das ***Operand Size Prefix-Byte***.

Bei Byte-Operationen ist die Identifizierung der Operanden-Größe wesentlich einfacher. Unabhängig davon, welches USE-Attribut gewählt worden ist, wird sowohl in einem USE32- als auch in einem USE16-Segment w = 0 anzeigen, daß es sich um eine Byte-Operation handelt.

Eine zusammenfassende Darstellung dieser Zusammenhänge zeigen die folgenden Tabellen:

1) **USE16-Code-Segment:**

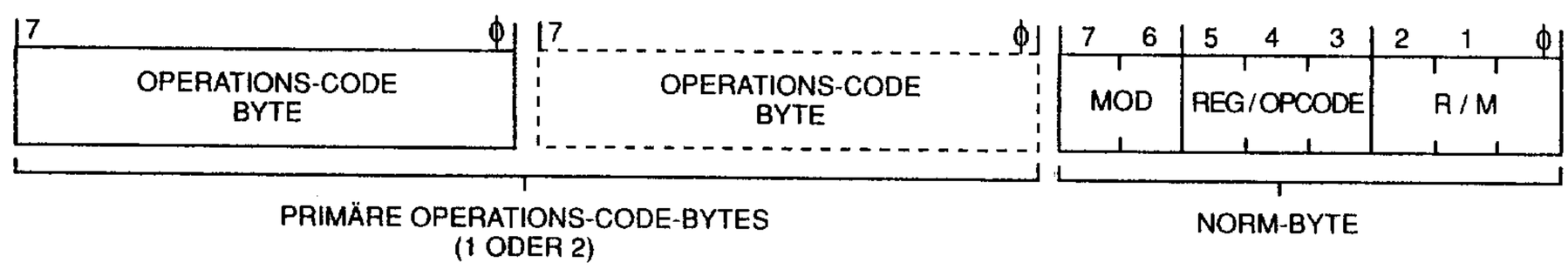
w-Feld	Operand-Prefix (66H)	Benützte Operanden-Größe (Typ) in einem USE16-Code-Segment	Codierbeispiele
0	Nein	Byte	MOV BH, BYTE PTR [BP] [DI]
1	Nein	Word	MOV DI, WORD PTR [BP] [DI]
1	Ja	Doubleword	MOV EDI, DWORD PTR [BP] {DI}

2) **USE32-Code-Segment:**

w-Feld	Operand-Prefix (66H)	Benützte Operanden-Größe (Typ) in einem USE 16-Code-Segment	Codierbeispiele
0	Nein	Byte	MOV BH, BYTE PTR [BP] [DI]
1	Ja	Word	MOV DI, WORD PTR [BP] [DI]
1	Nein	Doubleword	MOV EDI, DWORD PTR [BP} {DI}

1.11.6 Das ModRM-Byte (Modus Register/Memory)

Sehr viele 80486-Befehle beziehen sich auf Operanden, die sich im Speicher aufhalten. Um diese Operanden zu erreichen, hat der Programmierer die Auswahl aus einer Vielzahl von Adressierungsarten. Zur Spezifikation der ausgewählten Adressierungsart steht dem Prozessor das ModRM-Byte im Befehlsformat zur Verfügung.

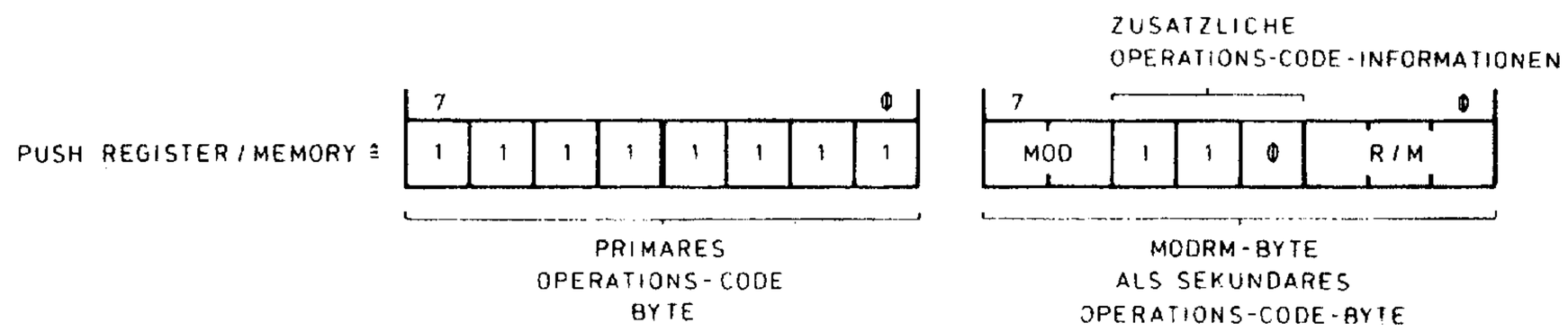


Wie das angegebene Bild zeigt, folgt das ModRM-Byte unmittelbar nach den primären Operations-Code-Byte(s).

- Das ***mod***-Feld (***mod***us), das die obersten 2 Bits im ModRM-Byte besetzt, wird immer mit dem r/m-Feld (***r***egister/***m***emory) kombiniert. Beide Felder zusammen liefern 32 mögliche Werte. Davon werden 24 Werte benützt, um die möglichen ***Speicher***-Referenzen zu spezifizieren.
  Da sich Operanden aber nicht nur im Speicher, sondern auch in Registern aufhalten, werden die restlichen 8 Werte benützt, um die möglichen ***Register***-Referenzen zu spezifizieren.
- Das ***reg***-Feld (***reg***ister) besetzt die nächsten 3 Bits im ModRM-Byte. Bei Register/Speicher-Operationen wird hier der Code des beteiligten Registers hinterlegt. Bei Register/Register-Operationen wird dieses Feld benützt, um entweder dasZiel- oder das Quellregister zu kennzeichnen.

Bei manchen Befehlen wird das reg-Feld ***nicht*** benützt, um ein Register zu spezifizieren, sondern es dient zur Aufnahme von zusätzlichen Operations-Code-Informationen. In diesem Fall kann das ModRM-Byte als ***sekundäres Operations-Code-Byte*** betrachtet werden.

BEISPIEL:



Bei diesem Befehl wir das reg-Feld des ModRMBytes als opcode Feld benützt (code = 110), um PUSH register/memory zu identifizieren.

Das ***r/m***-Feld (***r***egister/***m***emory) besetzt die letzten beiden Bits im ModRM-Byte. Bei Register/Speicher-Operationen wird es benützt, um zusammen mit dem Mod-Feld die Adressierungsarten festzulegen. Bei Register/Register-Operationen wird dieses Feld benützt, um entweder das Ziel- oder dasd Quellregister zu spezifizieren.

BEISPIELE:

1. **MOV EAX,EBX**

   → Das Quellregister ist im r/m-Feld codiert
   → Das Zielregister ist im reg-Feld codiert.

2. **MOV EAX,EBX**

   → Das Quellregister ist im reg-Feld codiert
   → Das Zielregister ist im r/m-Feld codiert.

Wie die angegebenen Beispiele zeigen, ist eine wechselseitige Identifikation de Quell- und Zielregister durch die reg- und r/m-Felder möglich.

An dieser Stelle ist nun die Frage zu beantworten, welches Identifikations-Mittel der Prozessor hat, um die gewünschte Transfer-Richtung einer Daten-Operation zu erkennen.

Dies soll am Beispiel der folgenden ASM386/486-Befehlsnotierungen geklärt werden:

	Ziel	Quelle	
1. **MOV**	**EAX,**	**[ESI]**	; Quelle ist Speicherplatz, ; Ziel ist Register
2. **MOV**	**[EDI]**	**;EAX**	; Quelle ist Register, ; Ziel ist Speicherplatz
3. **MOV**	**EAX,EBX**		; Quelle ist Register, ; Ziel ist Register

Gemäß Vereinbarung gilt, daß rechts vom Komma die ***Quelle eines Operanden*** und links vom Komma das ***Ziel eines Operanden*** zu notieren ist. Dabei kann sowohl die Quelle als auch das Ziel ein Speicherplatz oder ein Register sein.

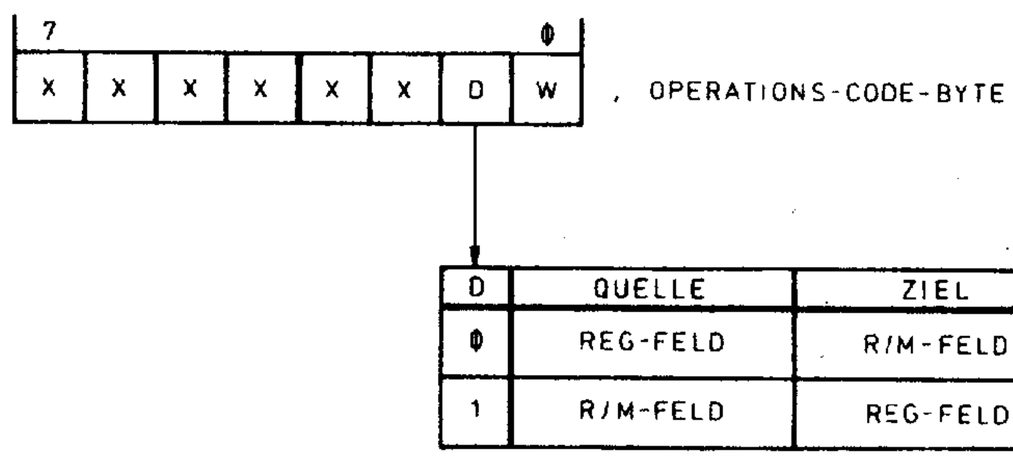
Beim ersten MOV-Befehl markiert der Zeiger [ESI] einen ***Speicher-Quelloperanden,*** der in das Zielregister EAX übertragen wird (Speicher/Register-Operation).

Beim zweiten MOV-Befehl wird das Quellregister EAX an eine Zielposition im Speicher übertragen, die durch den Zeiger [EDI] markiert ist (Register/Speicher-Operation).

Beim dritten MOV-Befehl wird das Quellregister EBX in das Zielregister EAX übertragen (Register/Register-Operation).

Wie die angegebenen Beispiele zeigen, muß der Prozessor für einen individuellen Befehl nicht nur das beteiligte Register und die benützte Adressierungsart, sondern auch die Richtung der Daten-Operation erkennen.

Zu diesem Zweck steht dem Prozessor im primären Operations-Code das ***d-Bit (direction)*** zur Verfügung. Dieses Bit hat direkten Einfluß auf die Interpretation der reg- und r/m-Felder im ModRM-Byte. Wie der Prozessor abhängig vom d-Wert diese Felder interpretiert, zeigt die folgende Tabelle:



D	QUELLE	ZIEL
0	REG-FELD	R/M-FELD
1	R/M-FELD	REG-FELD

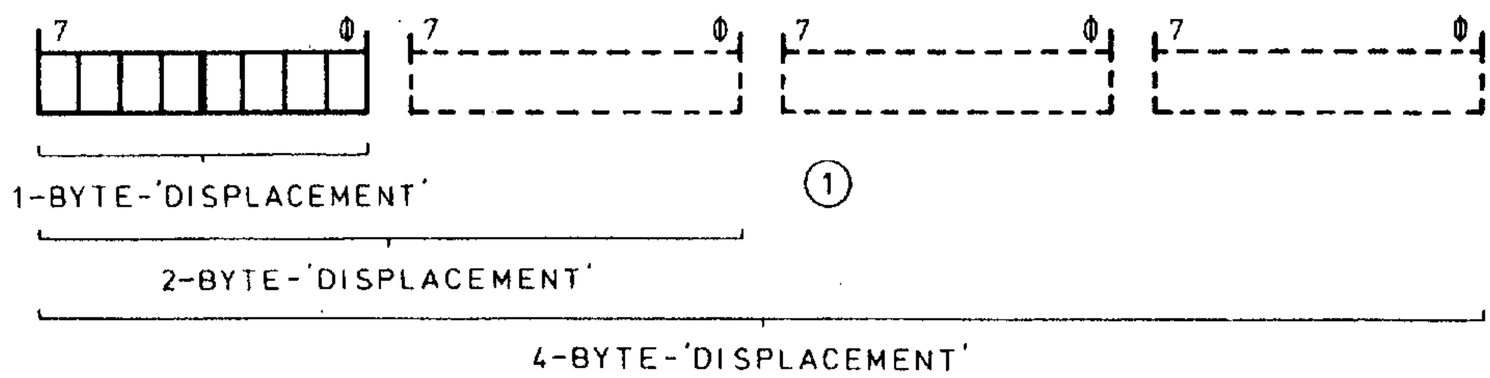
Die Auswirkung dieser Tabelle auf die oben notierten ASM386/486-Befehle zeigt die folgende Auflistung:

①	MOV	Ziel Quelle EAX , ESI reg-Feld r/m-Feld	d = 1
	MOV	EDI, EAX r/m-Feld reg-Feld	d = 0
②	MOV	EAX, EBX r/m-Feld reg-Feld	d = 0
	MOV	EAX, EBX reg-Feld r/m-Feld	d = 1

Es ist ersichtlich, daß zur Identifizierung eines Speicherplatzes immer das r/m-Feld benützt wird ①. Dabei legt d = 1 den Speicherplatz als Quelle und d = 0 den Speicherplatz als Ziel fest. Zur Identifizierung eines Registers kann sowohl das reg- als auch das r/m-Feld benützt werden ②. Dies hat zur Folge, daß für individuelle Register/Register-Operationen jeweils 2 unterschiedliche Befehls-Codes existieren, die jedoch in ihrer Wirkung identisch sind.

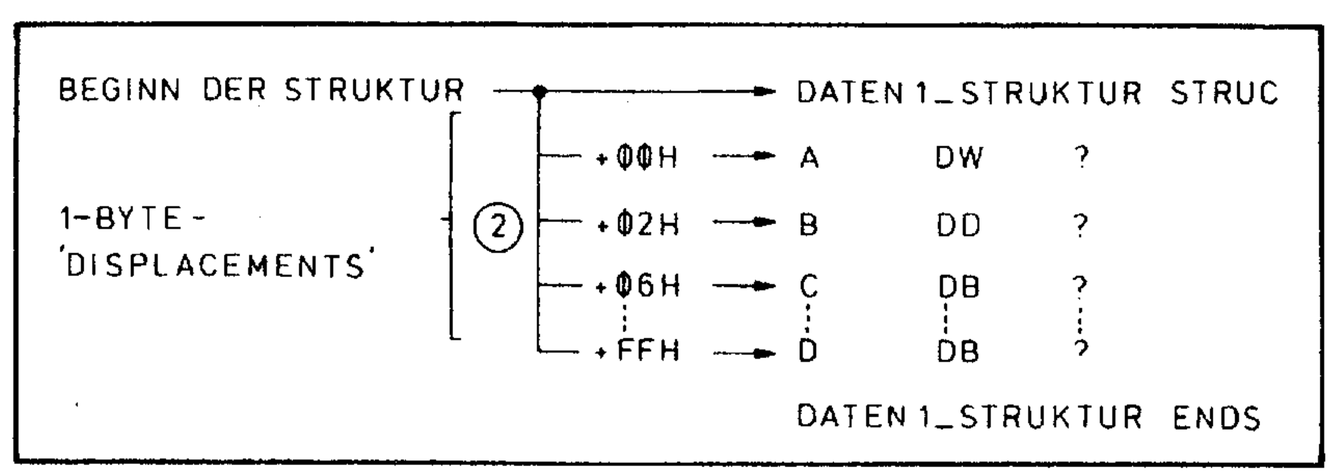
1.11.7 Das „Displacement"

Wie bereits im Kapitel „Adressierungsarten" gezeigt worden ist, kann ein 80486-Befehl ein Displacement enthalten, um die Position eines Speicheroperanden zu erreichen.



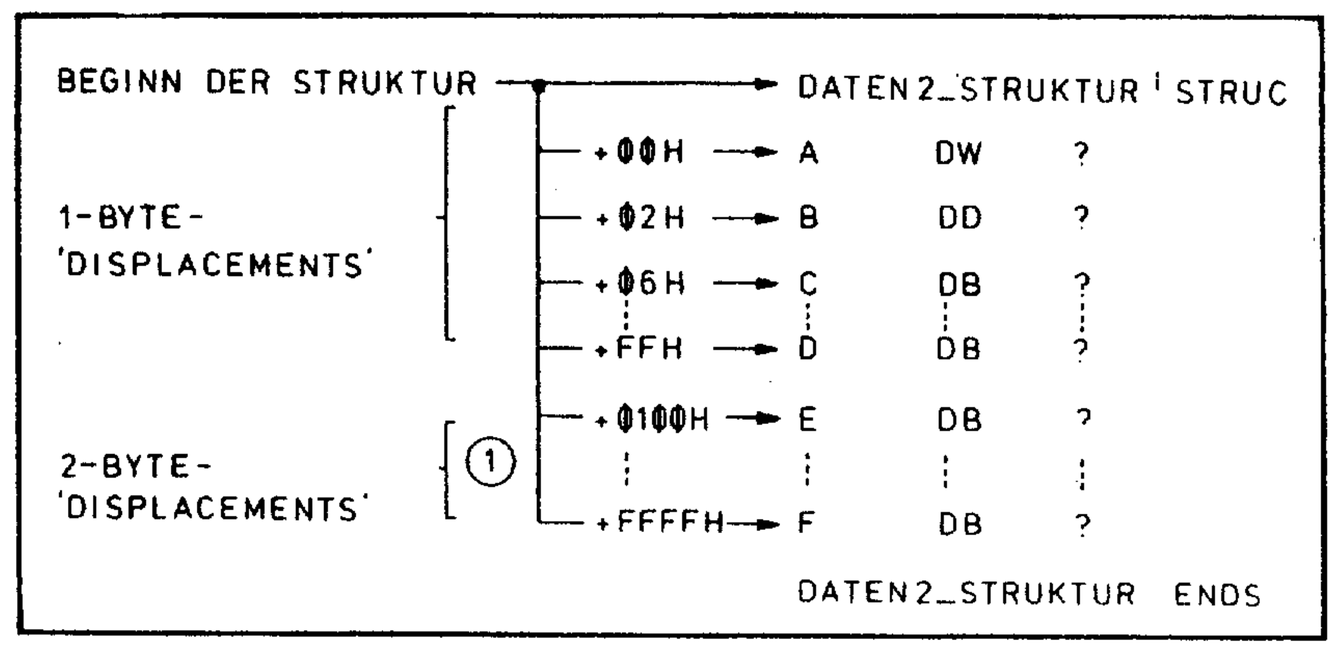
Ist ein Displacement vorhanden, kann es aus 1, 2 oder 4 Bytes bestehen ①.

Ein ***1-Byte-Displacement*** wird immer dann benützt, wenn Fundamental-Variablen innerhalb einer ASM386/486-Struktur-Variablen zu adressieren sind. Dabei ist vorausgesetzt, daß die Struktur-Variable maximal 256 Bytes enthält. das Displacement ist identisch mit dem Byte-Abstand einer Fundamental-Variablen relativ zum Beginn der Struktur. Dies illustriert die folgende Struktur-Vereinbarung:



Sie zeigt das „Rahmenwerk" der ASM386/486-Struktur DATEN1_STRUKTUR, in der die Fundamental-Variablen A (Typ Word), B (Typ Doubleword), C (Typ Byte) und D (Typ Byte) enthalten sind. Es ist ersichtlich, daß die DATEN1_STUKTUR maximal 256 Bytes (0 ... FFH) enthält. Die Fundamental-Variablen können daher im Byte-Abstand +0, +2, +6 bis +FFH durch ein 1-Byte-Displacement relativ zum Beginn der DATEN1_STRUKTUR erreicht werden ②.

Sollte eine ASM386/486-Struktur mehr als 256 Bytes enthalten, sind mindestens 2-Byte-Displacements erforderlich, um die Fundamental-Variablen zu adressieren. Dies verdeutlicht die folgende Struktur-Vereinbarung:

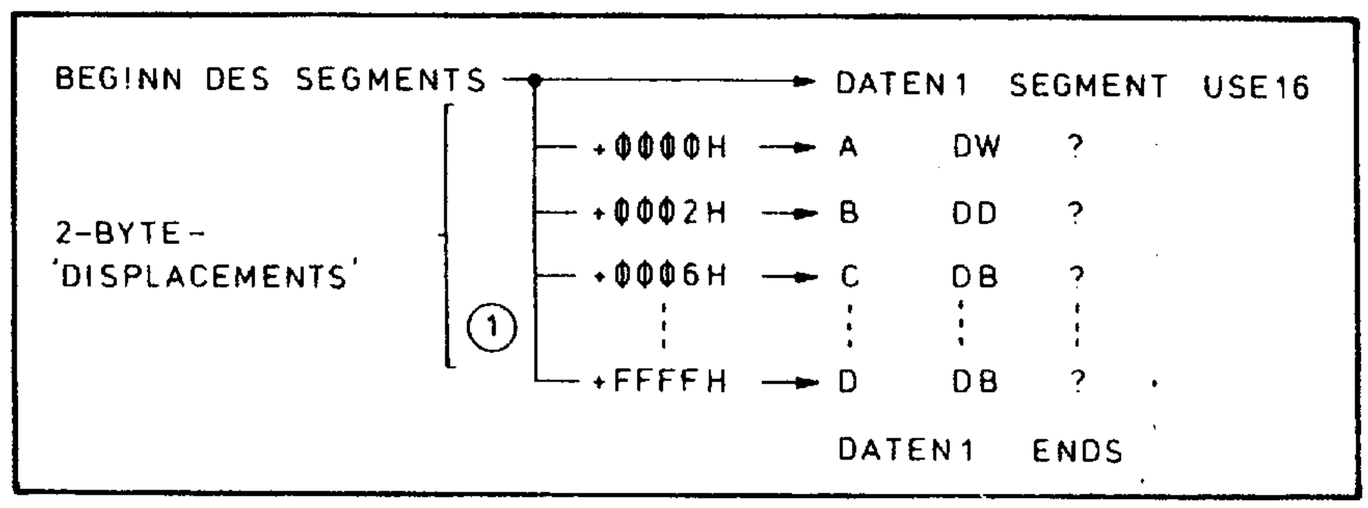


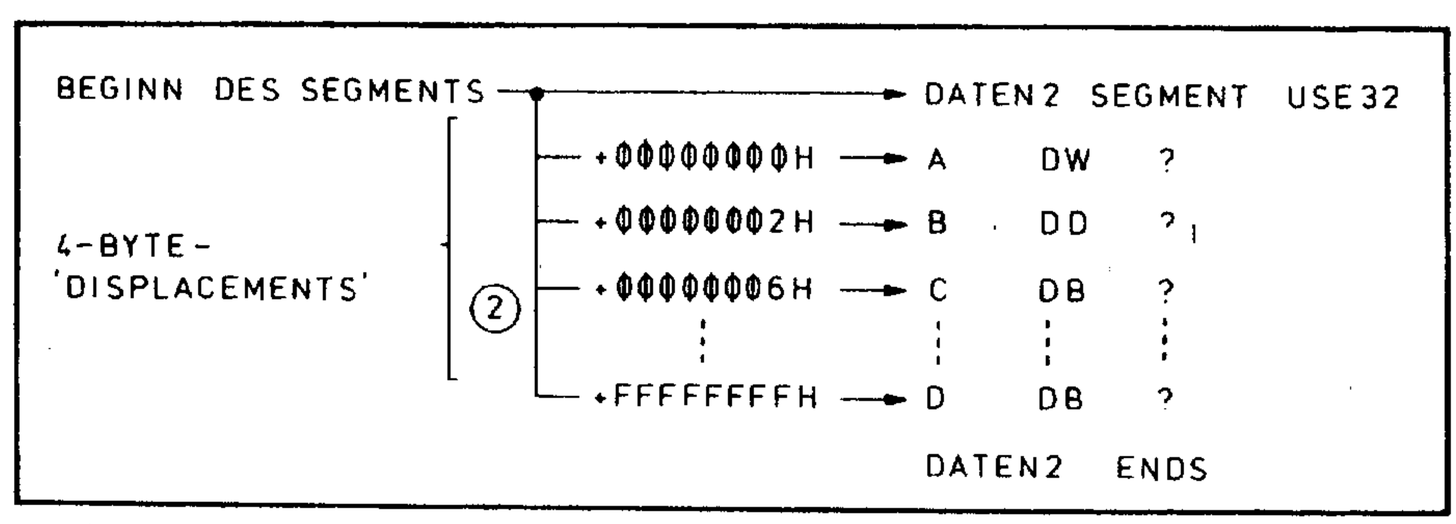
Es ist zu sehen, daß ab dem Byte-Abstand 100H 2-Byte-Displacements benützt werden. Unter der Voraussetztung, daß sich die Struktur-Variable in einem USE16-Segment aufhält, sind bis zu 64 KBytes (0 ... FFFFH) innerhalb der Struktur-Variablen adressierbar ①.

Sollen „normale" Variablen innerhalb eines Segments adressiert werden, so halten sie sich entweder in einem USE16- oder in einem USE32-Segment auf.

Dabei werden Variablen in einem USE16-Segment immer durch ***2-Byte-Displacements*** und in einem USE32-Segment durch ***4-Byte-Displacements*** erreicht. Hier sind die Displacements identisch mit den Byte-Abständen der Variablen relativ zum Beginn der Segmente.

Dies verdeutlicht die folgenden Segment-Vereinbarungen:

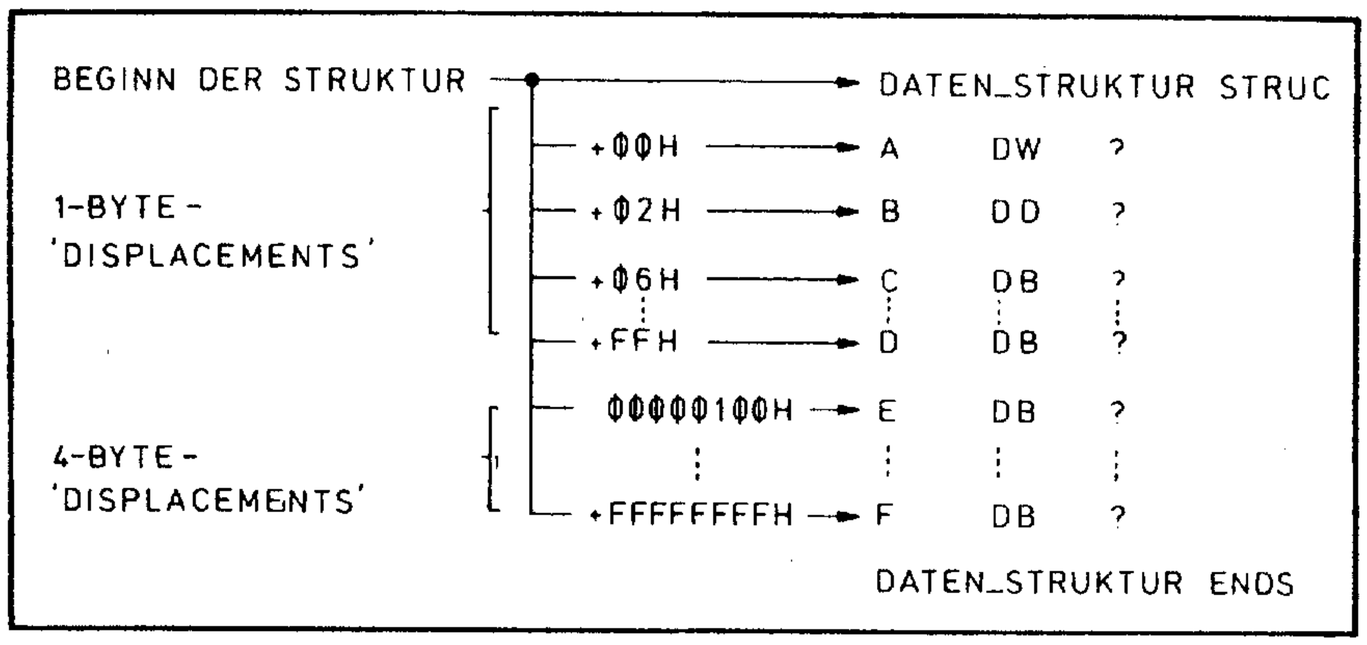




Die Variablen A (Typ Word), B (Typ Doubleword), C (Typ Byte) und D (Typ Byte) sind im USE16-Segment DATEN1 durch 2-Byte-Displacements und im USE32-Segment DATEN2 durch 4-Byte-Displacements erreichbar.

Dies bedeutet, daß der maximale Displacement-Wert im USE16-Segment FFFFH ist ①, während im USE32-Segment das Displacement Werte bis FFFFFFFFH erreichen kann ②. Letzteres ist aber nur im „Protected Mode" möglich, wenn der System-Programmierer dem Anwender einen flachen Adreßraum zu Verfügung stellt.

Sollte in einem USE32-Segment eine Struktur-Variable existieren, so werden die Fundamental-Variablen bis zum Byte-Abstand FFH durch ein 1-Byte-Displacement adressiert, während für Byte-Abstände, die größer als FFH sind, 4-Byte-Displacements benützt werden. Dies verdeutlicht die folgende Struktur-Vereinbarung:



1.11.8 Die hexadezimalen Codierungen der 16-Bit-Adressierungsarten mit ModRM

Die folgende Tabelle zeigt alle hexadezimalen Repräsentationen im ModRM-Byte und die daraus resultierenden 16-Bit-Adressierungsformen des 80486:

8-Bit-Register 16-Bit-Register 32-Bit-Register			AL AX EAX	CL CX ECX	DL DX EDX	BL BX EBX	AH SP ESP	CH BP EBP	DH SI ESI	BH DI EDI
reg/opcode			000	001	010	011	100	101	110	111
Effektive Adressen	mod	r/m	ModRM-Byte (hexadezimal)							
BX] [SI]	00	000	00	08	10	18	20	28	30	38
[BX] [DI]		001	01	09	11	19	21	29	31	39
[BP] [SI]		010	02	0A	12	1A	22	2A	32	3A
[BP] [DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX] [SI]*disp8*	01	000	40	48	50	58	60	68	70	78
[BX] [DI]*disp8*		001	41	49	51	59	61	69	71	79
[BP] [SI]*disp8*		010	42	4A	52	5A	62	6A	72	7A
[BP] [DI*disp8*		011	43	4B	53	5B	63	6B	73	7B
[SI]*disp8*		100	44	4C	54	5C	64	6C	74	7C
[DI]*disp8*		101	45	4D	55	5D	65	6D	75	7D
[BP]*disp8*		110	46	4E	56	5E	66	6E	76	7E
[BX]*disp8*		111	47	4F	57	5F	67	6F	77	7F
[BX] [SI*disp16*	10	000	80	88	90	98	A0	A8	B0	B8
[BX] [DI]*disp16*		001	81	89	91	99	A1	A9	B1	B9
[BP] [SI]*disp16*		010	82	8A	92	9A	A2	AA	B2	BA
[BP] [DI]*disp16*		011	83	8B	93	9B	A3	AB	B3	BB
[SI]*disp16*		100	84	8C	94	9C	A4	AC	B4	BC
[DI]*disp16*		101	85	8D	95	9D	A5	AD	B5	BD
[BP]*disp16*		110	86	8E	96	9E	A6	AE	B6	BE
[BX]*disp16*		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CK		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

HINWEIS:
disp8 ist ein 8 Bit-Displacement, ***disp16*** ist ein 16 Bit-Displacement.

1.11.9 Die hexadezimalen Codierungen der 32-Bit-Adressierungsarten mit ModRM

Die folgende Tabelle zeigt alle hexadezimalen Repräsentationen im ModRM-Byte und die daraus resultierenden 32 Bit-Adressierungsformen des 80486:

8-Bit-Register 16-Bit-Register 32-Bit-Register			AL AX EAX	CL CX ECX	DL DX EDX	BL BX EBX	AH SP ESP	CH BP EBP	DH SI ESI	BH DI EDI
reg/opcode			000	001	010	011	100	101	110	111
Effektive Adressen	mod	r/m	ModRM-Byte (hexadezimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
SIB ist präsent		100	04	0C	14	1C	24	2C	34	3C
*disp*32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]*disp8*	01	000	40	48	50	58	60	68	70	78
[ECX*disp8*		001	41	49	51	59	61	69	71	79
[EDX]*disp8*		010	42	4A	52	5A	62	6A	72	7A
[EBX]*disp8*		011	43	4B	53	5B	63	6B	73	7B
SIB ist präsent		100	44	4C	54	5C	64	6C	74	7C
[EBP]*disp8*		101	45	4D	55	5D	65	6D	75	7D
[ESI]*disp8*		110	46	4E	56	5E	66	6E	76	7E
[EDI]*disp8*		111	47	4F	57	5F	67	6F	77	7F
[EAX]*disp32*	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]*disp32*		001	81	89	91	99	A1	A9	B1	B9
[EDX*disp32*		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]*disp32*		011	83	8B	93	9B	A3	AB	B3	BB
SIB ist präsent		100	84	8C	94	9C	A4	AC	B4	BC
[EBP] *disp32*		101	85	8D	95	9D	A5	AD	B5	BD
[ESI] *disp32*		110	86	8E	96	9E	A6	AE	B6	BE
[EDI] *disp32*		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

HINWEIS:
disp8 ist ein 8 Bit-Displacement, ***disp32*** ist ein 32 Bit-Displacement.

Es ist zu sehen, daß durch das ModRM-Byte alleine nur ein Teil der vom 80486 unterstützten Adressierungsarten dargestellt werden kann. Zu dieser Teilmenge gehören

- alle Register-Adressierungen

und alle Adressierungsformen, bei denen der Prozessor zur effektiven Adreß-Bildung folgende Register benutzt:

- ein einzelnes Index-Register oder
- ein einzelnes Index-Register zusammen mit einem Displacement.

1.11.10 Das SIB-Byte

Zur Unterstützung der restlichen Adressierungsarten, bei denen der Prozessor

- ein skaliertes Index-Register (Index-Register × Skalierfaktor) zusammen mit einem Basis-Register oder
- ein skaliertes Index-Register zusammen mit einem „Displacement" oder
- ein skaliertes Index-Register zusammen mit einem Basis-Register und einem „Displacement"

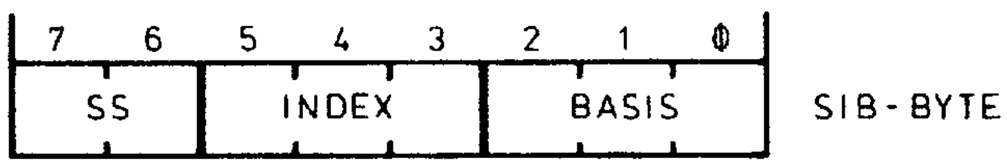
benützt, liefert ASM386/486 das sogenannte SIB-Byte (*S*cale *I*ndex *B*ase), das dem ModRM-Byte unmittelbar folgt.

Ist das SIB-Byte erforderlich, wird dies bereits durch die Codes

MOD	R/M
00	100
01	100
10	100

im ModRM-Byte angezeigt. In der Tabelle „Die hexadezimalen Codierungen der 32 Bit-Adressierungsarten mit ModRM" ist dies durch ***SIB ist präsent*** gekennzeichnet.

Immer dann, wenn in einem Programm die 32-Bit-Adressierungsformen „Based Indexed" oder „Scaled Indexed" benützt werden, liefert ASM386/486 das SIB-Byte. Das folgende Bild zeigt das Format dieses Bytes:



- Das ***ss-Feld***, das die obersten 2 Bits besetzt, spezifiziert den Skalierfaktor in Übereinstimmung mit der angegebenen Tabelle:

SS	SKALIER-FAKTOR
00	X1
01	X2
10	X4
11	X8

- Das ***Index-Feld***, das die nächsten 3 Bits nach dem ss-Feld besetzt, spezifiziert das Index-Register in Übereinstimmung mit der folgenden Tabelle:

INDEX	SKALIERTES INDEX-REGISTER
000	EAX
001	ECX
010	EDX
011	EBX
100	KEIN INDEX-REGISTER**
101	EBP
110	ESI
111	EDI

****HINWEIS:**
Wenn das Index-Feld den Wert 100 enthält, wird angezeigt, daß ***kein*** Index-Register vorhanden ist. In diesem Fall ***muß*** ss = 0 sein. Ist der Index 100 und ss = 01 oder 10 oder 11, dann ist die effektive Adresse undefiniert.

Das ***Basis-Feld***, das die letzten 3 Bits des SIB-Bytes besetzt, spezifiziert das Basis-Register.

Ist das SIB-Byte präsent, weil im ModRM-Byte mod = 00, 01 oder 10 und r/m = 100 ist, ergeben sich die zusätzlichen 32-Bit-Adressierungsformen „Based Indexed" bzw. „Scaled Indexed" in Übereinstimmung mit der folgenden Tabelle:

Mod/RM-Byte		SIB-Byte	
Mod-Feld	r/m-Feld	Basis-Feld	Effektive Adresse
00	100	000	[EAX + (Skaliertes Index-Register)]
00	100	001	[ECX + (Skaliertes Index-Register)]
00	100	010	[EDX + (Skaliertes Index-Register)]
00	100	011	[EBX + (Skaliertes Index-Register)]
00	100	100	[ESP + (Skaliertes Index-Register)]
00	100	101	[disp32 + (Skaliertes Index-Register)]**
00	100	110	[ESI + (Skaliertes Index-Register)]
00	100	111	[EDI + (Skaliertes Index-Register)]
01	100	000	[EAX + (Skaliertes Index-Register) + disp8]
01	100	001	[ECX + (Skaliertes Index-Register) + disp8]
01	100	010	[EDX + (Skaliertes Index-Register) + disp8]
01	100	011	[EBX + (Skaliertes Index-Register) + disp8]
01	100	100	[ESP + (Skaliertes Index-Register) + disp8]
01	100	101	[EBP + (Skaliertes Index-Register) + disp8]
01	100	110	[ESI + (Skaliertes Index-Register) + disp8]
01	100	111	[EDI + (Skaliertes Index-Register) + disp8]
10	100	000	[EAX + (Skaliertes Index-Register) + disp32]
10	100	001	[ECX + (Skaliertes Index-Register) + disp32]
10	100	010	[EDX + (Skaliertes Index-Register) + disp32]
10	100	011	[EBX + (Skaliertes Index-Register) + disp32]
10	100	100	[ESP + (Skaliertes Index-Register) + disp32]
10	100	101	[EBP + (Skaliertes Index-Register) + disp32]
10	100	110	[ESI + (Skaliertes Index-Register) + disp32]
10	100	111	[EDI + (Skaliertes Index-Register) + disp32]

****HINWEIS:**
Wenn mod = 00 ist und das Basis-Feld 101 enthält, wird angezeigt, daß ***kein*** Basis-Register benützt wird. Die so spezifizierte Adressierungsart ist dann z. B. durch folgende Notierung repräsentiert:

MOV EAX,FELD [ESIx2]

disp 32 skaliertes Index-Register

1.11.11 Die hexadezimalen Codierungen der 32-Bit-Adressierungsarten mit SIB

Die folgende Tabelle zeigt alle hexadezimalen Repräsentationen im ***SIB***-Byte und die daraus resultierenden 32-Bit-Adressierungsformen des 80486:

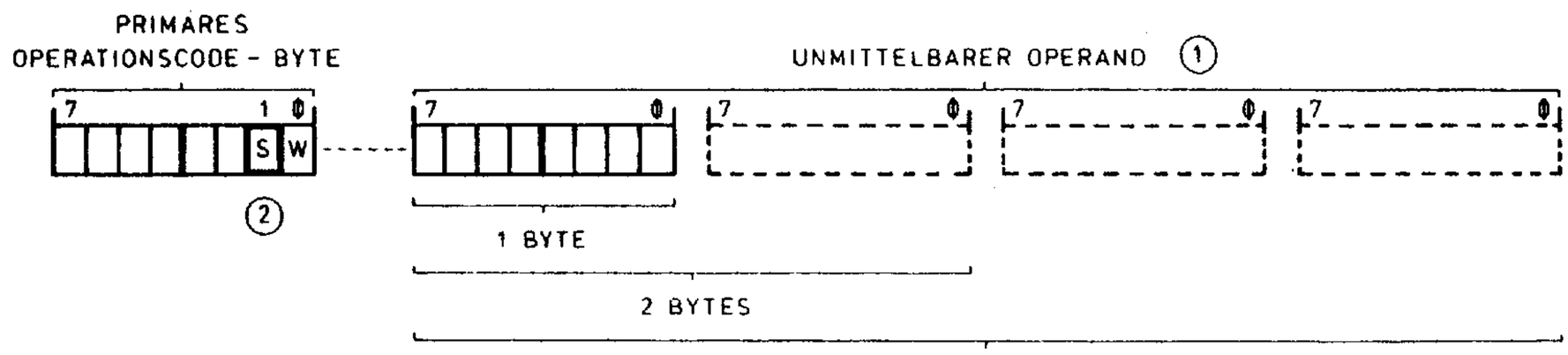
Basis-Register			[EAX] 000	[ECX] 001	[EDX] 010	[EBX] 011	[ESP] 100	[EBP] 101	[ESI] 110	[EDI] 111
Skaliertes Index-Register	SS-	Index	SIB-Byte (hexadezimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
kein Index-Reg.		100	04	0C	14	1C	24	2C	34	3C
[EBP]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX*2]	01	000	40	48	50	58	60	68	70	78
[ECX*2]		001	41	49	51	59	61	69	71	79
[EDX*2]		010	42	4A	52	5A	62	6A	72	7A
[EBX*2]		011	43	4B	53	5B	63	6B	73	7B
kein Index-Reg.		100	44	4C	54	5C	64	6C	74	7C
[EBP*2]		101	45	4D	55	5D	65	6D	75	7D
[ESI*2]		110	46	4E	56	5E	66	6E	76	7E
[EDI*2]		111	47	4F	57	5F	67	6F	77	7F
[EAX*4]	10	000	80	88	90	98	A0	A8	B0	B8
[ECX*4]		001	81	89	91	99	A1	A9	B1	B9
[EDX*4]		010	82	8A	92	9A	A2	AA	B2	BA
[EBX*4]		011	83	8B	93	9B	A3	AB	B3	BB
kein Index-Reg.		100	84	8C	94	9C	A4	AC	B4	BC
[EBP*4]		101	85	8D	95	9D	A5	AD	B5	BD
[ESI*4]		110	86	8E	96	9E	A6	AE	B6	BE
[EDI*4]		111	87	8F	97	9F	A7	AF	B7	BF
[EAX*8]	11	000	C0	C8	D0	D8	E0	E8	F0	F8
[ECX*8]		001	C1	C9	D1	D9	E1	E9	F1	F9
[EDX*8]		010	C2	CA	D2	DA	E2	EA	F2	FA
[EBX*8]		011	C3	CB	D3	DB	E3	EB	F3	FB
kein Index-Reg.		100	C4	CC	D4	DC	E4	EC	F4	FC
[EBP*8]		101	C5	CD	D5	DD	E5	ED	F5	FD
[ESI*8]		110	C6	CE	D6	DE	E6	EE	F6	FE
[EDI*8]		111	C7	CF	D7	DF	E7	EF	F7	FF

**

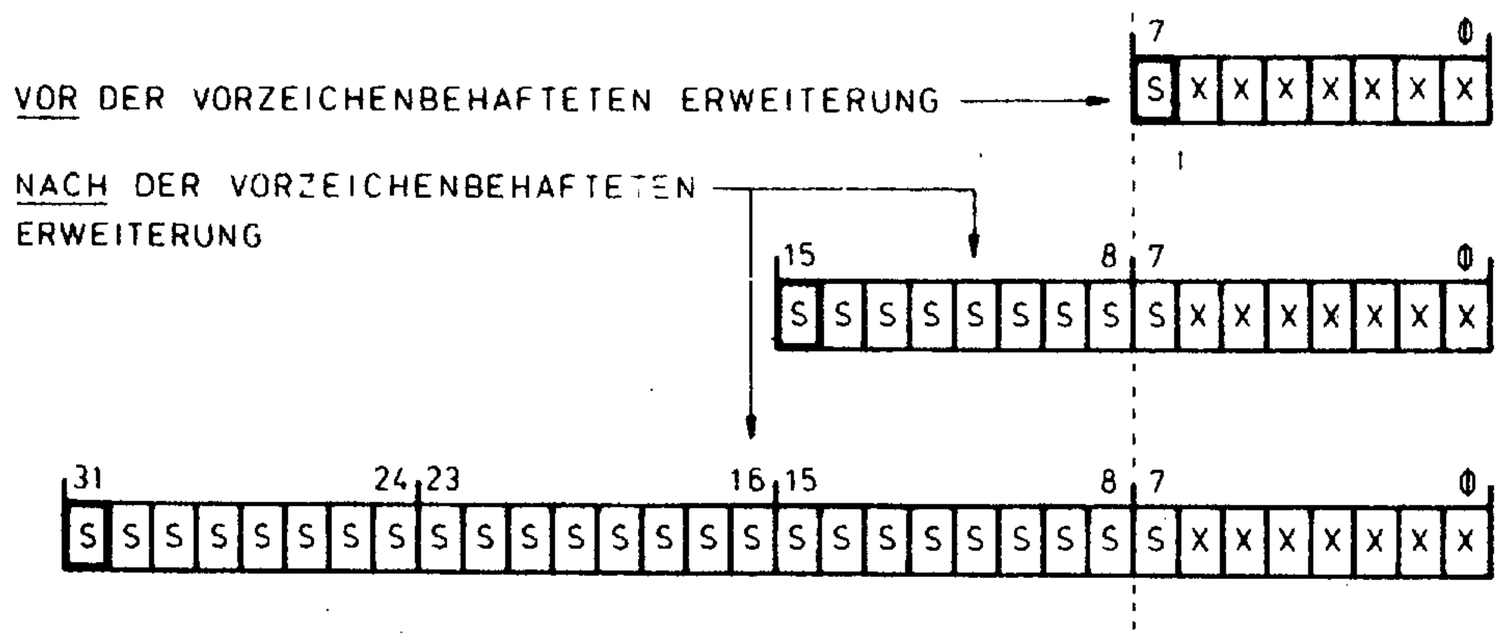
****HINWEIS:**
Wenn mod = 00 ist, wird ***kein*** Basis-Register benützt. Dies hat zur Folge, daß nun die in der Spalte angegebenen hexadezimalen Werte die Adressierungsform [***disp32*** + (skaliertes Index-Register)]
kennzeichnen.

1.11.12 Unmittelbarer Operand

Wenn der Befehl einen unmittelbaren Operanden benützt, dann folgt dieser Operand irgendeinem Displacement-Byte. Der unmittelbare Operand hält sich also immer im letzten Feld des Befehls auf.



Ist der unmittelbare Operand spezifiziert, kann er aus 1, 2 oder 4 Bytes bestehen ①. Im Zusammenhang mit unmittelbaren Operanden wird bei einigen Befehlen das ***s-Feld*** im primären Operations-Code-Byte benützt ②. Dieses Feld ist nur dann wirksam, wenn ein 8-Bit unmittelbarer Operand in einem 16-Bit- oder 32-Bit-Zielformat aufbewahrt werden soll. Welche Wirkung s hat, zeigt die folgende Tabelle:

s	Wirkung auf 8-Bit unmittelbare Operanden	Wirkung auf 16-Bit oder 32-Bit unmittelbare Operanden
0	keine	keine
1	Vorzeichenbehaftete Erweiterung des 8-Bit-Operanden auf das 16-Bit- oder 32-Bit-Zielformat	keine

Bei s = 1 interpretiert der Prozessor einen 8-Bit unmittelbaren Operanden als vorzeichenbehaftete Integer-Zahl und füllt automatisch die zusätzlichen Bits im größeren 16-Bit- oder 32-Bit-Zielformat mit dem Vorzeichen-Bit im kleineren 8-Bit-Quellformat auf („Sign Extension"). Dies verdeutlicht das folgende Bild: