Kapitel 4

Adressierung von Variablen

(für MASM-, ASM86- und ASM286-Programmierer)

Die einfachste Methode eine Variable zu adressieren, ist den Offset der Variablen im 8086/80186/80286 anzugeben. Zusammen mit dem Basiswert des zugehörigen Segment-Registers bilden dann die CPUs die physikalische Speicheradresse der Variablen. Diese Methode wird als **Direkt-Offset-Adressierung** bezeichnet.

Beispiel:

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_8
                            2
                            3     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                            4
----                        5     DATEN_PROG      SEGMENT
                            6
0000 0000                   7     VAR_1           DW      0
0002 (2                     8                     DD      2 DUP(?)
     ????????
     )
000A 06                     9     VAR_2           DB      6,7,8
000B 07
000C 08
                           10
----                       11     DATEN_PROG      ENDS
                           12
----                       13     CODE_PROG       SEGMENT
                           14
0000 A10000                15     START:          MOV     AX,VAR_1
0003 80060A0073            16                     ADD     VAR_2,73H
0008 3A0E0B00              17                     CMP     CL,VAR_2+1
                           18
----                       19     CODE_PROG       ENDS
                           20
                           21              END    START
```

Wird das Programm ausgeführt, lädt der erste Befehl `MOV AX, VAR_1` ein Word bei Offset Null von DATEN_PROG in das AX-Register.

Der zweite Befehl `ADD VAR_2, 73H` addiert die Konstante 73H zum Byte bei Offset 10 von DATEN_PROG. Da die Variable VAR_2 vom Typ Byte ist, ist der Befehl `ADD VAR_2, 73H` implizit als Byte-Operation gekennzeichnet.

Das zweite und dritte Byte von VAR_2 kann **nicht** direkt mit einem Variablennamen adressiert werden. Der CMP-Befehl benützt den Ausdruck VAR_2 + 1, um das nachfolgende Byte von VAR_2 zu adressieren. In der Tat hat die mit VAR_2 + 1 spezifizierte Variable das gleiche **Typ-** und **Segment**-Attribut wie VAR_2, nämlich BYTE und DATEN_PROG.

Lediglich der Offset ist gegenüber dem Offset von VAR_2 um 1 erhöht. Soll das dritte Byte nach VAR_2 erreicht werden (Initialisierungswert 8), ist dies mit Hilfe des Ausdrucks VAR_2 + 2 möglich.

Die angegebene Methode der direkten Offset-Adressierung zeigt, daß der Offset einer zu adressierenden Variablen direkt im 8086/80186/80286-Befehl anzugeben ist.

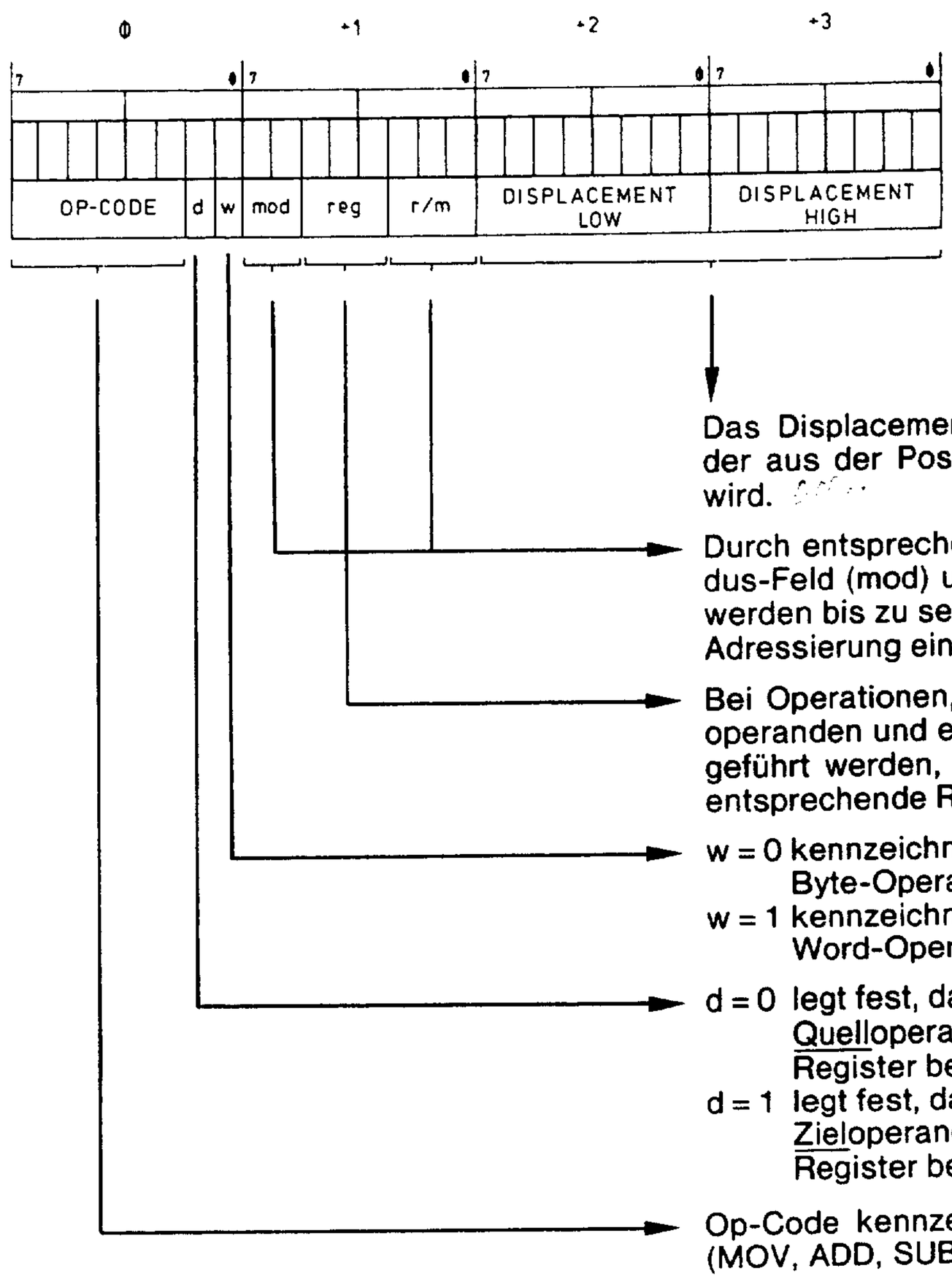
Beispiel:

```
CMP   AL,   VAR_2 + 2
```

Offset einer Variablen

Der Offset einer direkt adressierten Variablen wird im Maschinencode des Befehls als sogenanntes Displacement hinterlegt.

Aufbau eines 8086/80186/80286-Befehls mit Displacement:



Um auf die unterschiedlichen Typen von Speicherdaten zugreifen zu können, verwenden die 8086/80186/80286-CPUs zur Bildung eines 32-Bit-Base:Offset-Zeigers bis zu drei unterschiedliche Offsetkomponenten:

- Ein Displacement
- Ein Base-Register (BX oder BP)
- Ein Index-Register (SI oder DI)

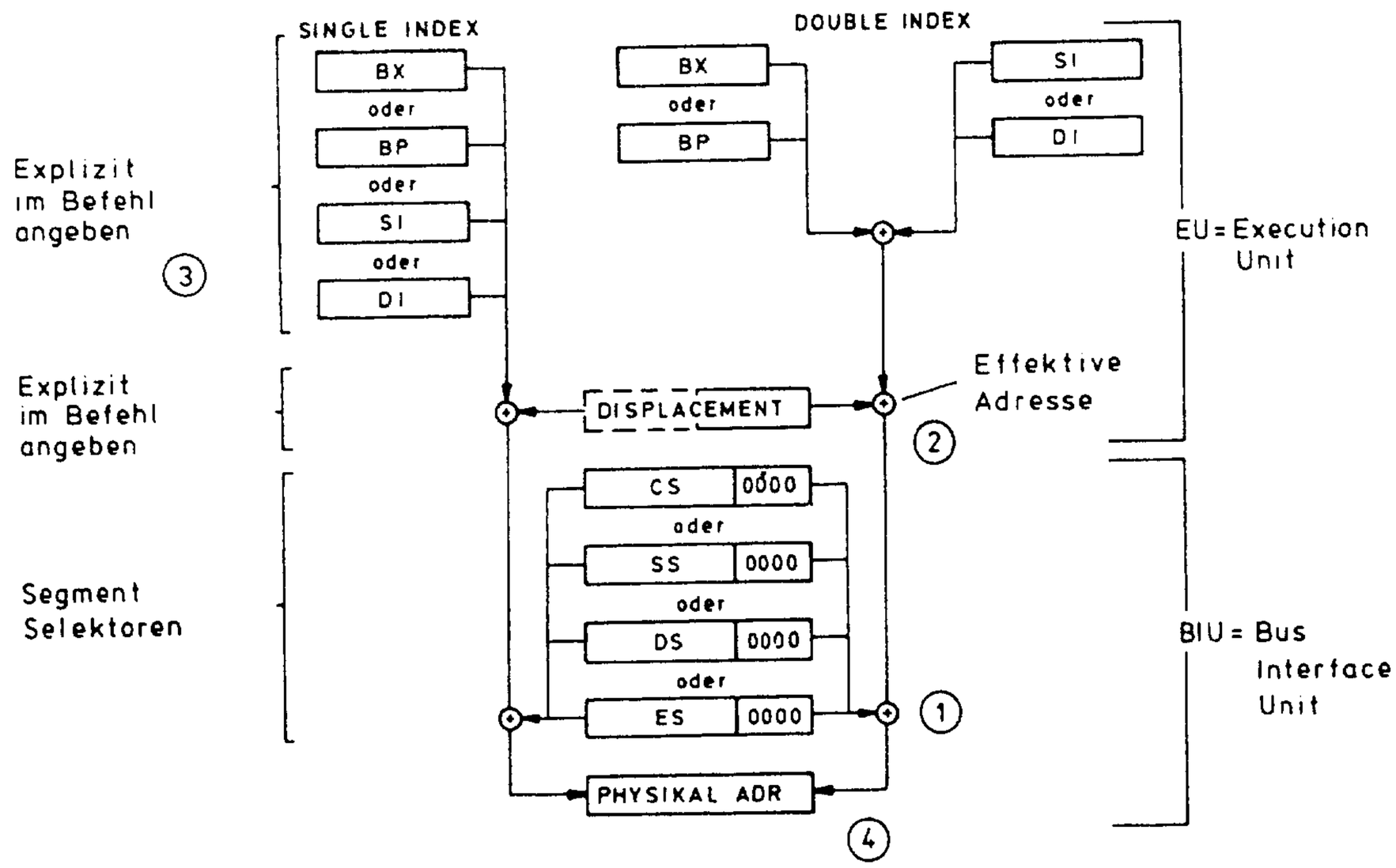
Diese drei Offsetkomponenten können additiv kombiniert werden. Der daraus resultierende Offset einer Variablen wird dann als **effektive Adresse** der Variablen bezeichnet.

Abhängig davon, in welcher Weise die möglichen Offsetkomponenten kombiniert werden, ergeben sich sechs verschiedene Methoden (Adressierungsarten), die effektive Adresse (EA) einer Variablen zu bilden.

Die angegebene Tabelle zeigt, wie die Offsetkomponenten zu kombinieren sind und welche Adressierungsarten sich daraus ergeben.

Adressierungsart	**Offset-Berechnung**
Direkt Offset	EA = DISP
Register Indirekt	EA = (BX) oder (BP) oder (SI) oder (DI)
Based	EA = [(BX) oder (BP)] + DISP
Indexed	EA = [(SI) oder (DI)] + DISP
Based Indexed	EA = [(BX) oder (BP)] + [(SI) oder (DI)]
Based Indexed mit Displacement	EA = [(BX) oder (BP)] + [(SI) oder (DI)] + DISP

Das angegebene Bild illustriert alle Möglichkeiten, die die Mikroprozessoren 8086/80186 und 80286 im Real-Mode haben, um einen 32-Bit-Base:Offset-Zeiger zu bilden.



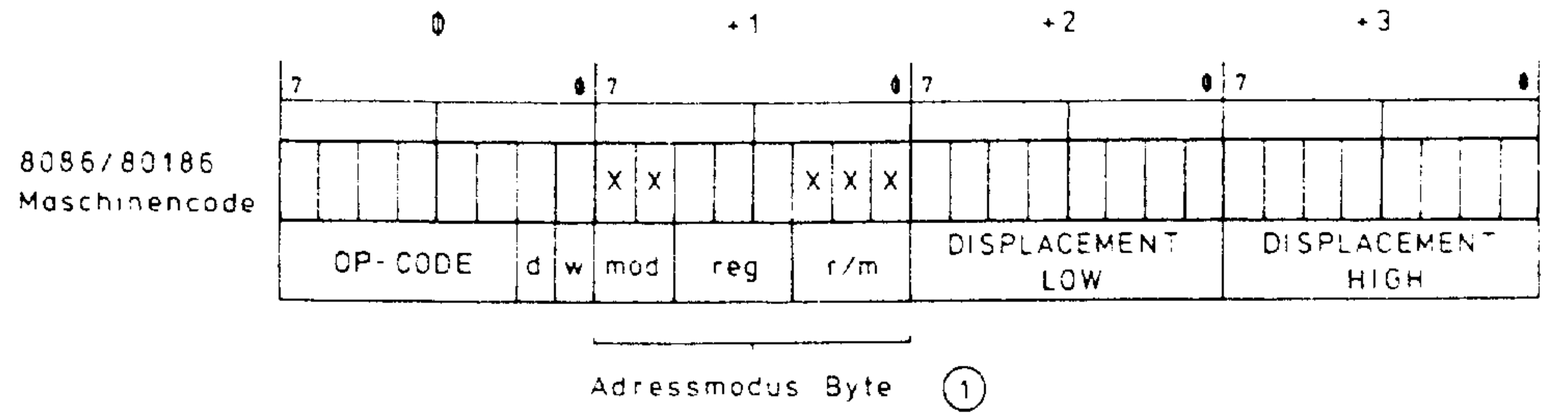
Die Segment-Selektoren CS, SS, DS und ES sind Bestand der 8086/80186/80286-Bus-Interface-Unit (BIU) und entsprechen den Base-Komponenten von 32-Bit-Zeigern, d. h. sie kennzeichnen den Anfang logischer Segmente ①.

Die 8086/80186/80286-Execution-Unit (EU) berechnet die effektive Adresse einer Variablen im selektierten Segment. D.h. die EU stellt die effektive Offset-Komponente eines 32-Bit-Zeigers zur Verfügung ②.

Wie die EU die einzelnen Offset-Komponenten zu kombinieren hat, gibt der Anwenderprogrammierer im 8086/80186/80286-Befehl explizit an ③.

Die BIU addiert schließlich zur physikalischen Basisadresse eines Segments die effektive Offset-Komponente und gibt die so gebildete 20-Bit-Adresse an den Adreßbus aus ④.

Abhängig davon, welche Adressierungsart der Anwenderprogrammierer wählt, belegt der Assembler im Adreßmodus-Byte des übersetzten Befehls die r/m- und mod-Felder mit entsprechenden Bitkombinationen ①.



Die nachfolgende Tabelle zeigt die 8086/80186/80286-Adressierungsarten und die daraus resultierenden Bitkombinationen in den r/m- und mod-Feldern.

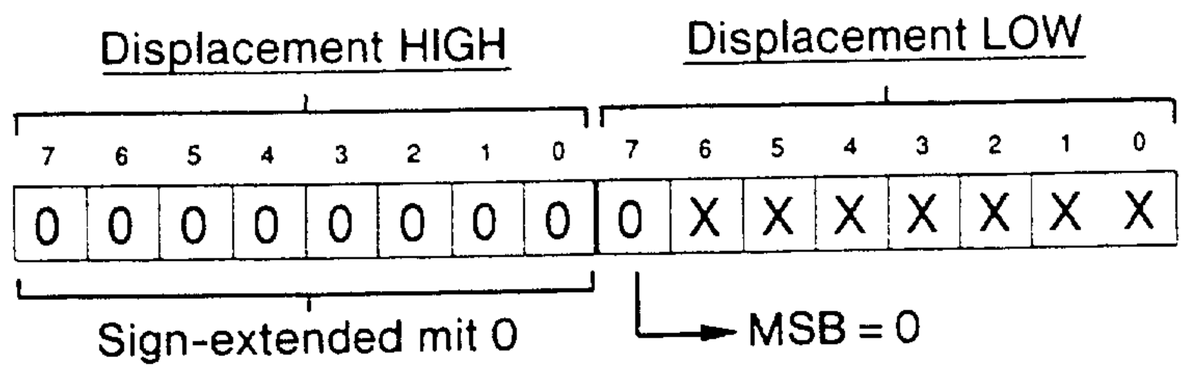
r/m	mod = 00	mod = 01**	mod = 10
000	(BX) + (SI)	(BX) + (SI) + DISP	(BX) + (SI) + DISP
001	(BX) + (DI)	(BX) + (DI) + DISP	(BX) + (DI) + DISP
010	(BP) + (SI)	(BP) + (SI) + DISP	(BP) + (SI) + DISP
011	(BP) + (DI)	(BP) + (DI) + DISP	(BP) + (DI) + DISP
100	(SI)	(SI) + DISP	(SI) + DISP
101	(DI)	(DI) + DISP	(DI) + DISP
110	DISP*	(BP) + DISP	(BP) + DISP
111	(BX)	(BX) + DISP	(BX) + DISP
		Nur Disp. Low	Disp. Low und Disp. High

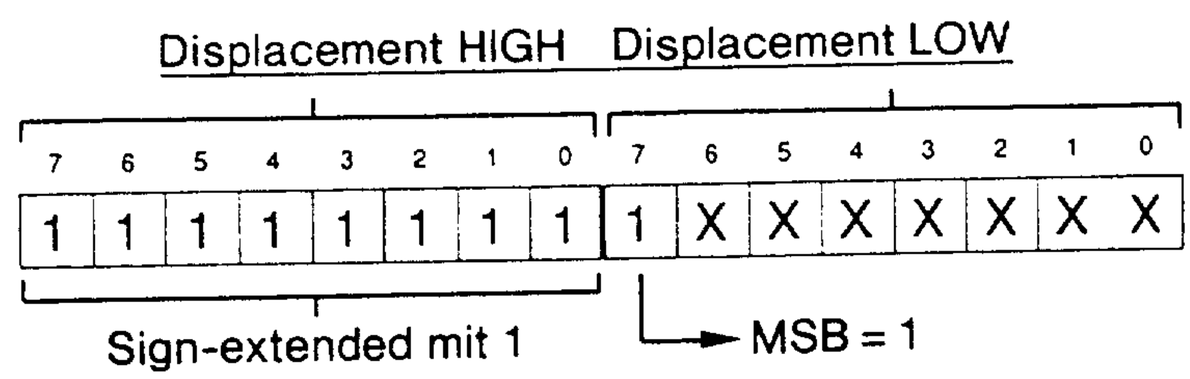
Hinweise:

* Die Bitkombination r/m = 110 und mod = 00 bildet eine Ausnahme. In diesem Fall bilden die beiden auf das Adreßmodusbyte folgenden Displacement-Bytes die effektive 16-Bit-Adresse.

** Wenn im mod-Feld die Bitkombination 01 steht, ist Displacement-HIGH nicht vorhanden. In diesem Fall wird Displacement-LOW abhängig vom Wert des MSB (Bit 7) von der CPU auf 16 Bit vorzeichenbehaftet erweitert (sign-extended).

Beispiele:

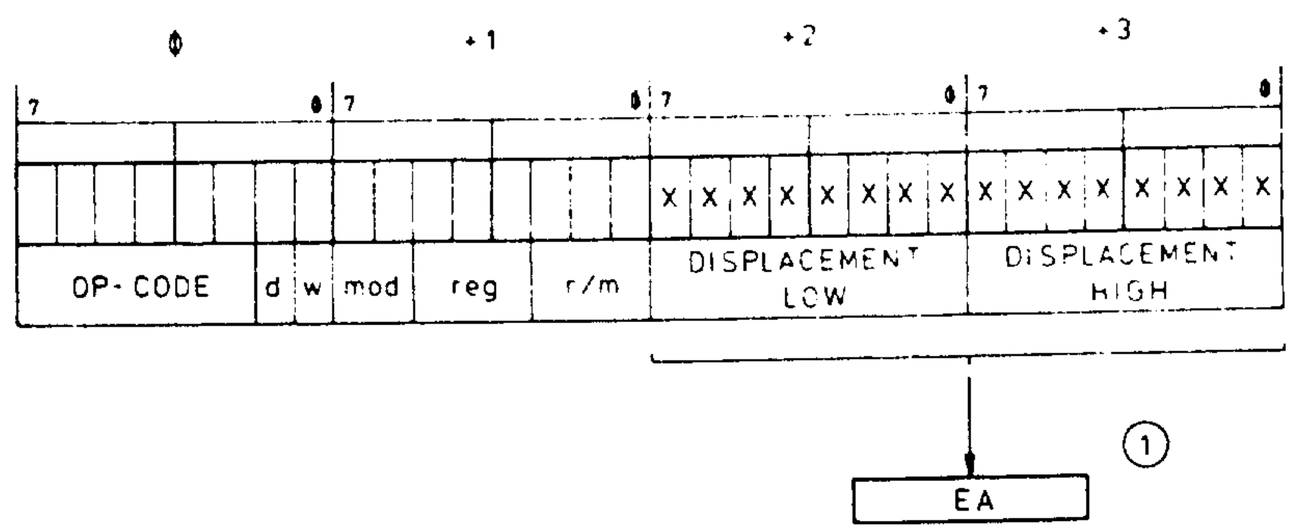




4.1 Anwendung der Adressierungsarten

4.1.1 Direkt Offset

Die Anwendung der Direkt-Offset-Adressierung wurde bereits im Beispiel 8 gezeigt. Es ist die einfachste Adressierungsmethode, bei der zur Bildung der effektiven Adresse **kein** Register benützt wird.



Die effektive Adresse einer Variablen wird direkt dem Displacement-Feld eines 8086/80186/80286-Befehls entnommen ①.

Für den Anwenderprogrammierer bedeutet dies, daß er den Namen der zu adressierenden Variablen im Befehl angeben muß. Die Direkt-Offset-Adressierung wird benützt, um nichtstrukturierte Daten (Skalare) zu adressieren.

Beispiele:

```
MOV   AX,   VAR_1
```

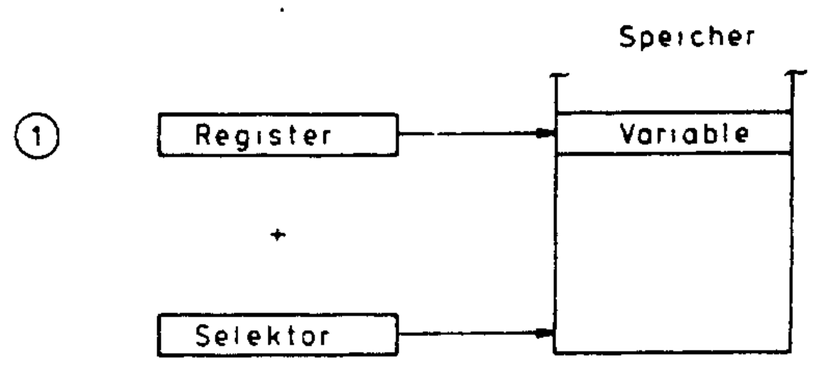
Der Offset von VAR_1 entspricht dem Wert im Displacement-Feld

```
CMP   CL,   VAR_2 + 1
```

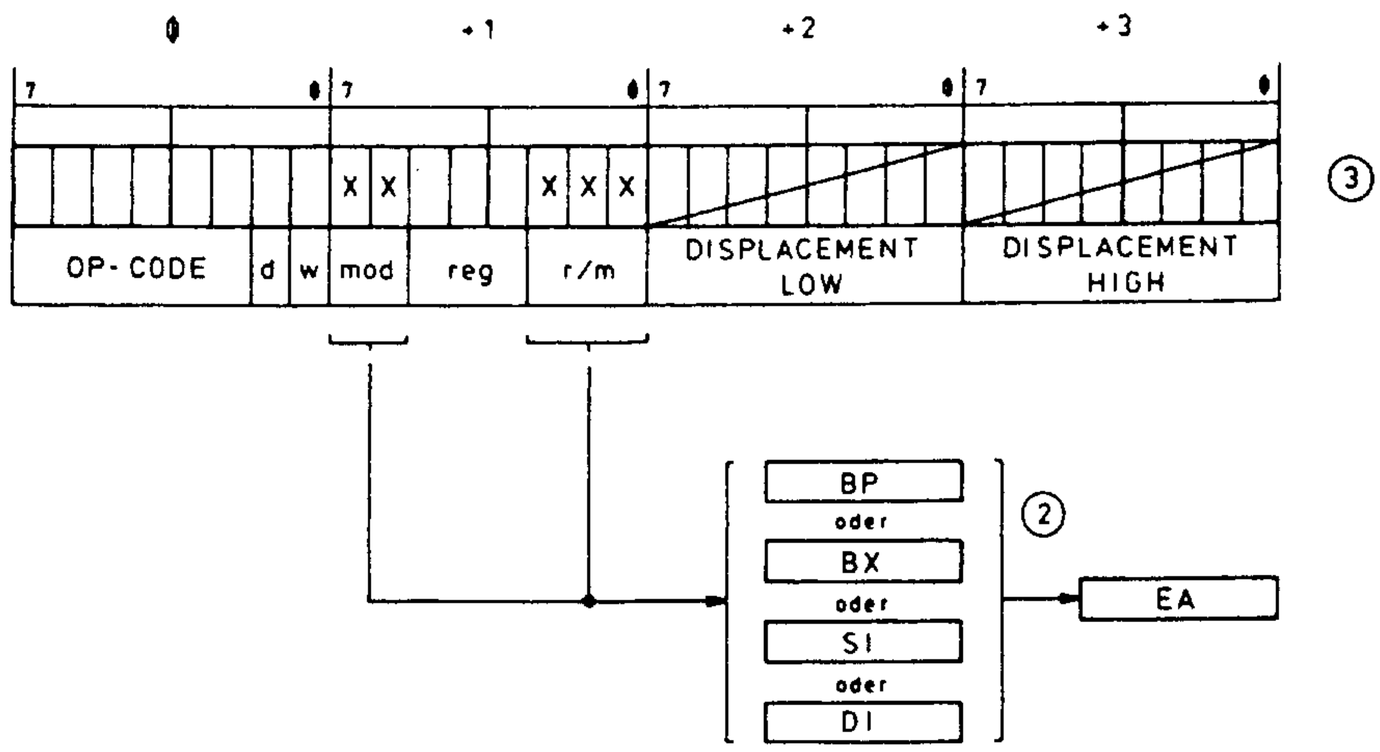
Der Offset von VAR_2 + 1 entspricht dem Wert im Displacement-Feld

4.1.2 Register-Indirekt

Bei dieser Adressierungsart kommt die Offset-Komponente einer Variablen von einem Register (BX, BP, SI oder DI) ①.



Abhängig davon, welches Register der Anwenderprogrammierer als Offset-Komponente wählt, belegt der Assembler das r/m- und mod-Feld im Adreßmodus-Byte mit einer entsprechenden Bitkombination.



Diese Bitkombination kennzeichnet das ausgewählte Register und bestimmt damit die effektive Adresse einer Variablen ②. Das Displacement-Feld ist bei indirekter Registeradressierung nicht belegt ③.

4.1.2 Register Indirekt

Die folgenden zwei Beispiele zeigen, wie in 8086/80186/80286-Befehlen die indirekte Registeradressierung anzugeben ist.

```
MOV CX, [BX]
ADD [DI], DH
```

Die indirekte Adressierung ist dadurch gekennzeichnet, daß die ausgewählten Register in eckigen Klammern angegeben werden.

Der Befehl `MOV CX, [BX]` besagt folgendes: Lade das Word, das durch den Offset im Register BX adressiert wird, in das Register CX.

Im Gegensatz dazu besagt der Befehl `MOV CX, BX` folgendes: Lade den Inhalt vom Register BX in das Register CX.

Wie der Vergleich beider Befehle zeigt, bekommt ein Register in eckigen Klammern die Funktion eines Zeigers.

Der Vorteil der indirekten Registeradressierung liegt darin, daß der Offset, also der Registerinhalt, während des Programmablaufs verändert werden kann. Diese Tatsache kann angewendet werden, um z.B. die einzelnen Elemente eines Datenfeldes zu adressieren.

Hinweis: Die Register **BX**, **SI** und **DI** bilden die effektive Adresse einer Variablen relativ zum **DS**-Selektor. Das Register **BP** bildet die effektive Adresse einer Variablen relativ zum **SS**-Selektor.

Das folgende Programmbeispiel zeigt eine Anwendung der Register-Indirekt-Adressierung.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_9
                            2
                            3     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                            4
----                        5     DATEN_PROG SEGMENT
                            6
0000 00                     7     UNPACKED_DECIMAL        DB      0,1,2,3,4,5,6,7,8,9
0001 01
0002 02
0003 03
0004 04
0005 05
0006 06
0007 07
0008 08
0009 09
                            8
----                        9     DATEN_PROG ENDS
                           10
----                       11     CODE_PROG SEGMENT
                           12
0000 BB0000                13     START:  MOV     BX,OFFSET UNPACKED_DECIMAL      ;[BX] entspricht dem
                           14                                                     ;Offset des ersten Datenbytes
0003 B90A00                15             MOV     CX,10                           ;CX ist der Wiederholungs-
                           16                                                     ;zaehler
0006 B030                  17             MOV     AL,30H                          ;Konstante fuer die ASCII Um-
                           18                                                     ;wandlung
0008 0007                  19     ASCII:  ADD     [BX],AL                         ;Umwandlung einer Dezimalzahl
                           20                                                     ;in ein ASCII-Zeichen
000A 43                    21             INC     BX                              ;BX zeigt zur naechsten Zahl
000B E2FB                  22             LOOP    ASCII                           ;Wiederhole solange, bis CX
                           23                                                     ;Null ist
                           24
----                       25     CODE_PROG ENDS
                           26
                           27             END     START
```

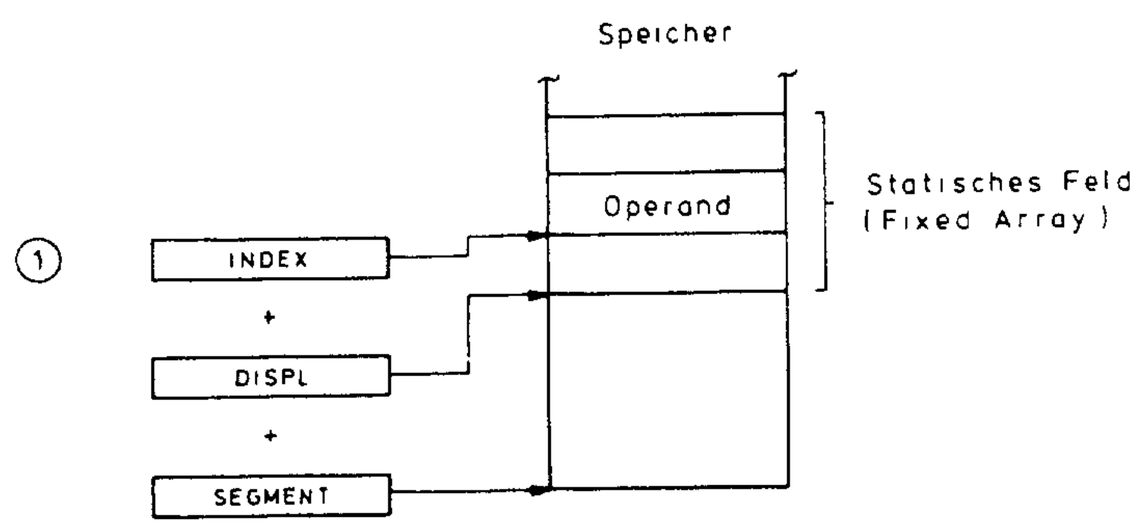
Es wird ein String ungepackter Dezimalzahlen, also Werte im Bereich von 0 bis 9, in einen String von ASCII-Zeichen umgewandelt.

Die Umwandlung erfolgt in einer Wiederholschleife, wobei zu jedem Byte der Wert 30H addiert wird.

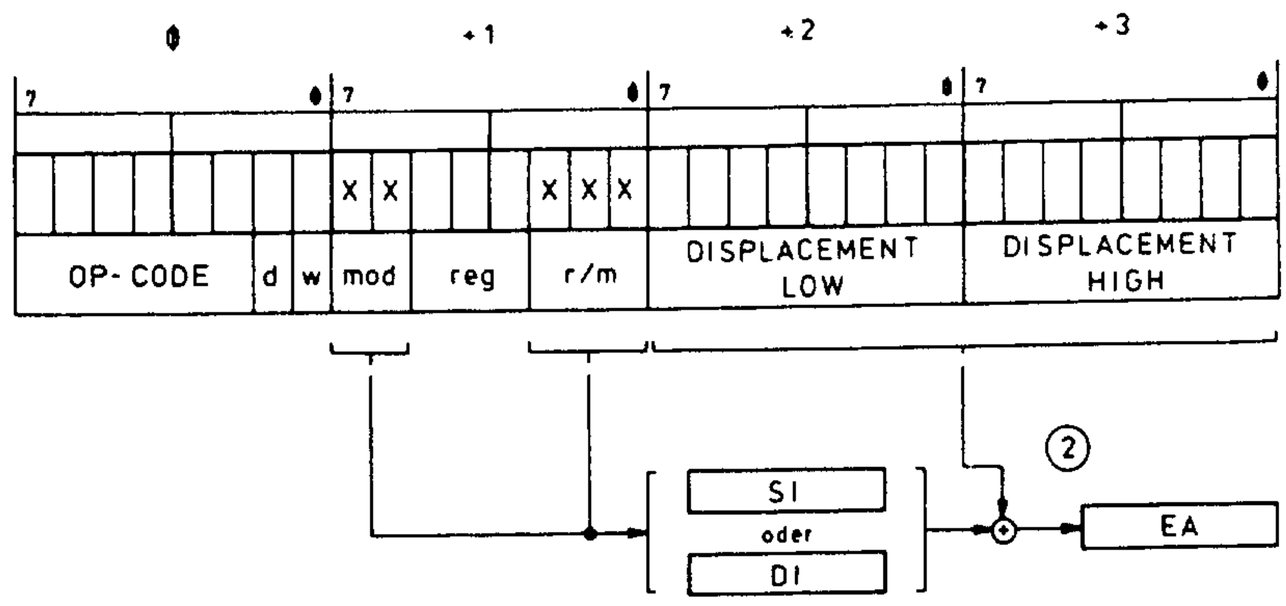
Zu Beginn des Programms wird [BX] mit dem Offset des ersten Datenbytes geladen. Mit jedem Schleifendurchlauf wird BX inkrementiert, so daß [BX] den Offset des nächsten Datenbytes bekommt.

4.1.3 Indexed

Die Indexed-Adressierung benützt den Wert im Register DI oder SI und einen Offset (Displacement) im 8086/80186/80286-Befehl. Die Offset-Komponente einer Variablen ergibt sich aus der Summe des SI- oder DI-Registerwertes und dem im Befehl codierten Displacement ①.



Abhängig davon, welches Register der Anwenderprogrammierer wählt, belegt der Assembler das r/m- und mod-Feld im Adreßmodus-Byte mit einer entsprechenden Bitkombination.



Diese Bitkombination kennzeichnet das ausgewählte Register (SI oder DI). Zusammen mit dem im Befehl codierten Displacement bilden sie die effektive Adresse eines Speicheroperanden ②.

Die Adressierungsart Indexed wird im Allgemeinen dazu verwendet, um auf die einzelnen Elemente in einem statischen Feld zuzugreifen. Jede Feldvariable ist durch einen symbolischen Namen gekennzeichnet. Der Offset dieser Feldvariablen entspricht dem Displacement im 8086/80186/80286-Befehl. D.h. das Displacement zeigt zum Anfang eines Feldes und das Index-Register selektiert das gewünschte Feldelement.

Da alle Feldelemente die gleiche Länge haben, können beliebige Feldelemente durch einfache arithmetische Operationen mit dem Index-Register adressiert werden. Das folgende Beispiel zeigt, wie in 8086/80186/80286-Befehlen die Indexed-Adressierung anzugeben ist.

```
ADD VAR_2 [DI], 92
```

Zunächst gilt auch hier, daß das in eckigen Klammern angegebene Register die Funktion eines Zeigers hat. VAR_2 ist der Name einer Feldvariablen und kennzeichnet den Anfang eines Feldes. Der Wert in DI kann als Index bzw. als Offset relativ zu VAR_2 aufgefaßt werden.

Im angegebenen Beispiel wird die Feldvariable VAR_2 adressiert, während [DI] ein bestimmtes Feldelement selektiert. Der Befehl addiert zu diesem Feldelement die Konstante 92.

Das folgende Programmbeispiel zeigt eine Anwendung der Indexed-Adressierung.

```
LOC  OBJ                LINE    SOURCE
                         1              NAME    BEISPIEL_10
                         2
                         3      ASSUME CS:CODE_PROG,DS:DATEN_PROG
                         4
----                     5      DATEN_PROG SEGMENT
                         6
0000 00                  7      FELD        DB      0,1,2,3,4,5,6,7,8,9     ;Feldvariable mit 10 Elementen
0001 01
0002 02
0003 03
0004 04
0005 05
0006 06
0007 07
0008 08
0009 09
                         8
----                     9      DATEN_PROG ENDS
                        10
----                    11      CODE_PROG SEGMENT
                        12
0000 BF0000             13      START:      MOV     DI,0                    ;[DI] entspricht dem
                        14                                                  ;Index des ersten Feldelements
0003 B90A00             15                  MOV     CX,10                   ;CX ist der Wiederholungs-
                        16                                                  ;zaehler
0006 800530             17      ASCII:      ADD     FELD[DI],30H            ;Umwandlung einer Dezimalzahl
                        18                                                  ;in ein ASCII-Zeichen
0009 47                 19                  INC     DI                      ;Feld Index inkrementieren
000A E2FA               20                  LOOP    ASCII                   ;Wiederholw solnage, bis CX
                        21                                                  ;Null ist
                        22
----                    23      CODE_PROG ENDS
                        24
                        25              END     START
```

Es sind in der Feldvariablen FELD ungepackte Dezimalzahlen, Werte im Bereich von 0 bis 9, gespeichert. Diese Feldelemente werden in ASCII-Zeichen umgewandelt. Die Umwandlung erfolgt in einer Wiederholschleife, wobei zu jedem Element der Wert 30H addiert wird.

Zu Beginn des Programms wird [DI] mit dem Index des ersten Feldelements geladen. Mit jedem Schleifendurchlauf wird DI inkrementiert, so daß [DI] den Index des nächsten Feldelements bekommt.

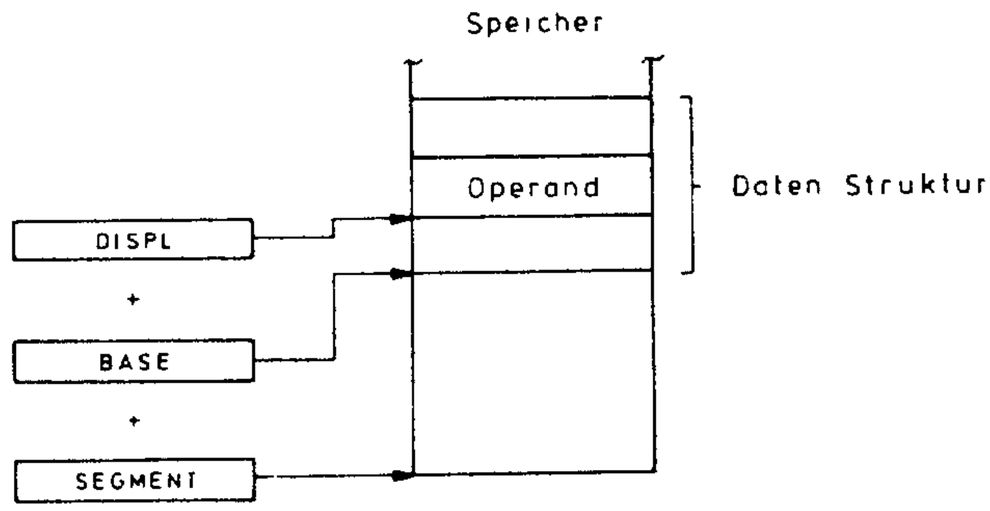
Eine andere Methode die Feldelemente zu adressieren, ist folgende Befehlssequenz:

```
ADD FELD [DI], 30H
ADD FELD [DI+1], 30H
ADD FELD [DI+2], 30H
  .
  .
```

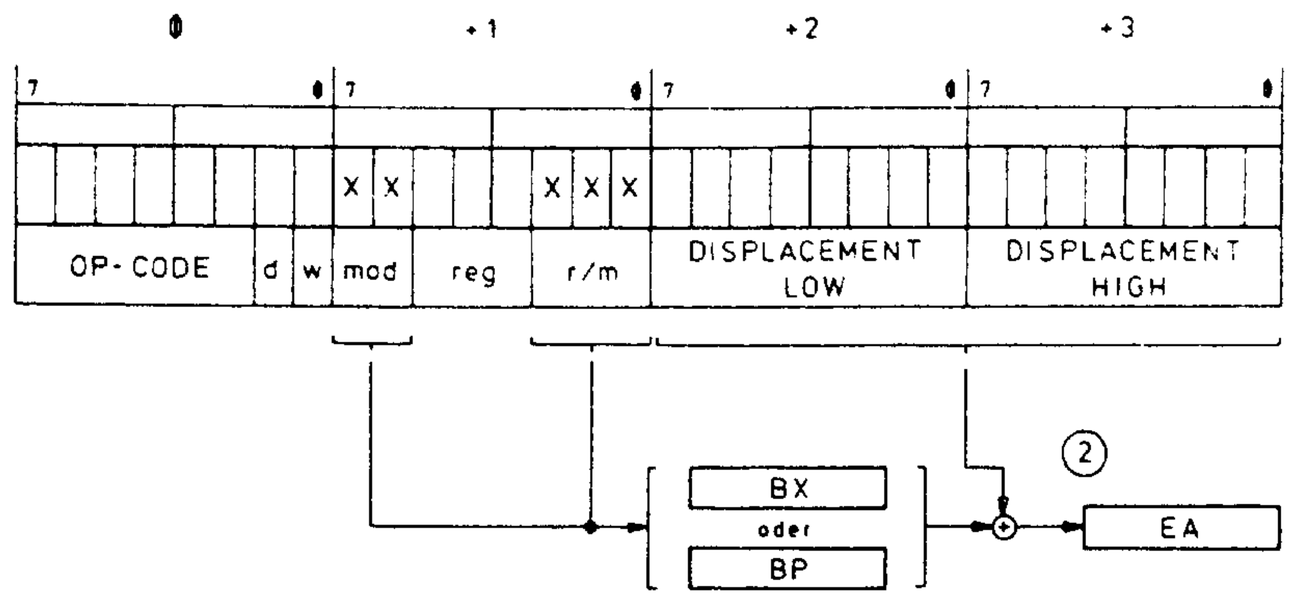
Sie bedeutet allerdings Codeverschwendung und ist daher keine Alternative zum LOOP-Statement.

4.1.4 Based

Die Based-Adressierung benützt den Wert im Register BX oder BP und einen Offset (Displacement) im 8086/80186/80286-Befehl. Die Offset-Komponente einer Variablen ergibt sich aus der Summe des BX- oder BP-Registerwertes und dem im Befehl codierten Displacement ①.



Abhängig davon, welches Register der Anwenderprogrammierer wählt, belegt der Assembler das r/m- und mod-Feld im Adreßmodus-Byte mit einer entsprechenden Bitkombination.



Diese Bitkombination kennzeichnet das ausgewählte Register (BX oder BP). Zusammen mit dem im Befehl codierten Displacement bilden sie die effektive Adresse eines Speicheroperanden ②.

4.1.4.1 Strukturen

In Assembler ist es möglich "strukturierte" Datenblöcke aufzubauen, die aus den fundamentalen Datentypen wie BYTE, WORD, DOUBLE WORD, QUAD WORD und TEN BYTE zusammengesetzt sein können.

Jede Fundamentalvariable innerhalb des strukturierten Datenblocks ist wie üblich mit Hilfe der Datenzuweisungs-Statements DB, DW, DD, DQ oder DT zu definieren. Der so aufgebaute Datenblock wird als "Struktur" bezeichnet, wobei die in ihm definierten Fundamentalvariablen separat durch Based-Adressierung erreichbar sind.

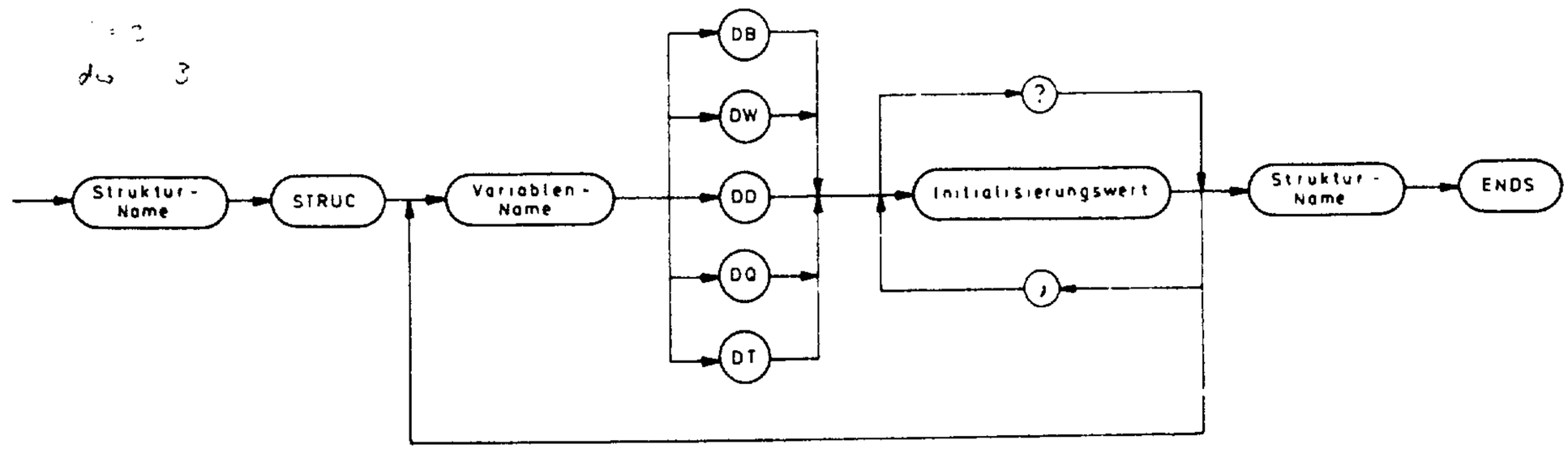
Zur Definition einer Struktur stehen in Assembler die **STRUC**- und **ENDS**-Statements zur Verfügung. Um eine komplexe Zahl darzustellen, kann eine Datenstruktur definiert werden, die aus zwei Fundamentalvariablen zusammengesetzt ist.

```
KOMPLEX STRUC
REAL_TEIL        DQ  1.2
IMAGINAER_TEIL   DD  3.5
KOMPLEX ENDS
```

Im Beispiel sind die Fundamentalvariablen REAL_TEIL und IMAGINAER_TEIL vom Typ QUAD WORD bzw. DOUBLE WORD, da sie mit Hilfe der DQ- und DD-Statements definiert wurden.

Jede vereinbarte Datenstruktur ist durch einen Assembler-Identifizierer gekennzeichnet. Im Beispiel ist dies der symbolische Name KOMPLEX. Anders ausgedrückt: KOMPLEX ist der Name eines **neuen** Datentyps, der aus den fundamentalen Datentypen des Assemblers zusammengesetzt ist.

Syntaxdiagramm zur Vereinbarung einer Datenstruktur:



Struktur-Name ist der metasprachliche Ausdruck für einen Assembler-Identifizierer. Der Anwenderprogrammierer setzt dafür einen symbolischen Namen ein und kennzeichnet damit die Datenstruktur.

Die Statements **STRUC** und **ENDS** sind Schlüsselwörter. Sie müssen angegeben werden, wenn eine Datenstruktur vereinbart werden soll.

Damit die Fundamentalvariablen innerhalb einer Datenstruktur erreicht werden können, sind sie symbolisch zu kennzeichnen. Der Anwenderprogrammierer setzt daher für den metasprachlichen Ausdruck **Variablen-Name** einen entsprechenden Identifizierer ein.

Initialisiert werden die Fundamentalvariablen wie üblich mit Hilfe der Datenzuweisungs-Statements (DB, DW, DD, DQ oder DT).

Ein Variablen-Name hat folgende Attribute:

- **Typ** kennzeichnet den Typ der Initialisierungswerte
- **Offset** gibt die Anzahl der Bytes ab Beginn der Datenstruktur an
- **Segment** **kein** Segment-Attribut, d.h. eine Assembler-Struktur kann außerhalb eines logischen Segments vereinbart werden

4.1.4.2 Strukturzuweisung und Initialisierung

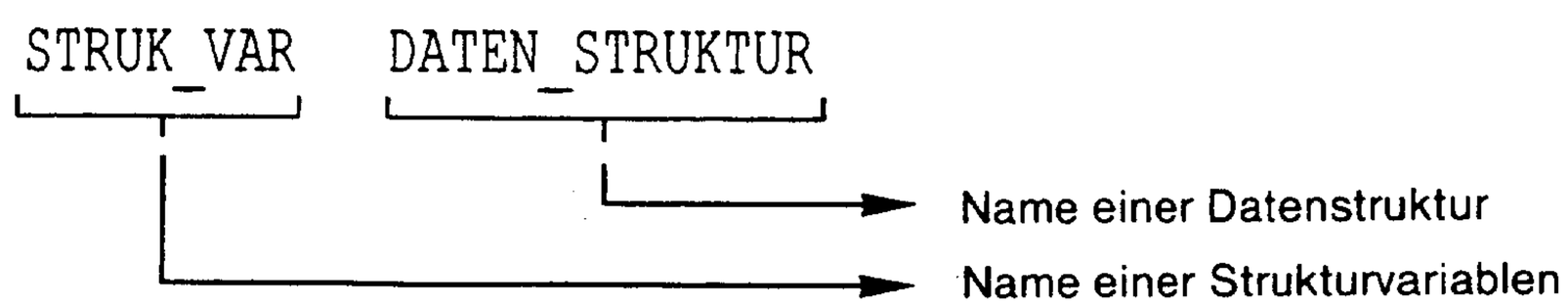
Es sei vorausgesetzt, daß in einem Assembler-Quellprogramm folgende Datenstruktur außerhalb eines logischen Segments vereinbart worden ist.

```
DATEN_STRUKTUR   STRUC
VAR_1  DB        ?
VAR_2  DB        ?
VAR_3  DB        ?
DATEN_STRUKTUR   ENDS
```

Die Struktur enthält drei Fundamentalvariable (VAR_1, VAR_2 und VAR_3) vom gleichen Typ (BYTE). Da die Variablen mit ? initialisiert worden sind, sind ihre Werte unbestimmt. Der vereinbarte Datentyp DATEN_STRUKTUR kann nun benützt werden, um einer neu zu definierenden Variablen Speicherplatz und Initialisierungswerte zuzuweisen.

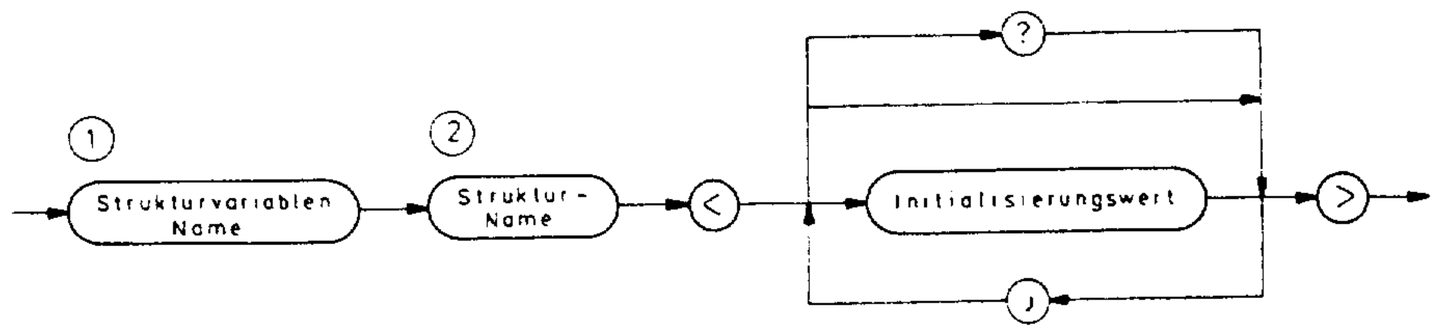
```
DATEN_PROG   SEGMENT
STRUK_VAR    DATEN_STRUKTUR <0, 3, 5>
DATEN_PROG   ENDS
```

Im logischen Segment DATEN_PROG existiert die Variable STRUK_VAR vom Typ DATEN_STRUKTUR. Da in DATEN_STRUKTUR drei Fundamentalvariable vom Typ BYTE definiert sind, kann die Variable STRUK_VAR nun diese drei Bytevariablen speichern. Definiert wird diese Strukturzuweisung durch:



In der so definierten Strukturvariablen STRUK_VAR existieren nun drei Bytes, die durch die Anweisung <0,3,5> initialisiert werden. Gemäß der angegebenen Reihenfolge der Initialisierungswerte bekommt in STRUK_VAR die Fundamentalvariable VAR_1 den Wert 0, VAR_2 den Wert 3 und VAR_3 den Wert 5.

Syntaxdiagramm der Strukturzuweisung und Initialisierung:



Basierend auf einer vereinbarten Datenstruktur, zeigt das angegebene Syntaxdiagramm Statements, mit deren Hilfe einer neu zu definierenden Strukturvariablen Speicherplatz und Initialisierungswerte zugewiesen werden können.

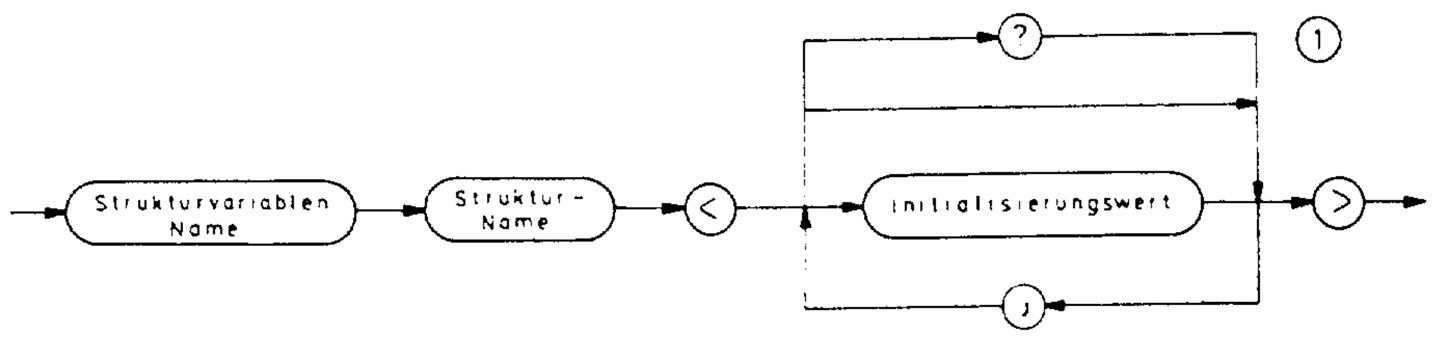
Strukturvariablen-Name ist der metasprachliche Ausdruck eines Namens, den der Anwenderprogrammierer für eine Strukturvariable vergibt ①.

Diese Variable hat folgende Attribute:

- Das **Segment-Attribut** gibt das logische Segment an, in dem die Strukturvariable definiert ist.
- Der **Offset** der Strukturvariablen entspricht der Anzahl der Bytes ab dem Beginn des Segments, in dem die Variable definiert ist.
- Der **Typ** der Strukturvariablen entspricht der Anzahl der Bytes, die ihr, basierend auf der vereinbarten Datenstruktur, zugewiesen werden.

Struktur-Name ist der metasprachliche Ausdruck eines Namens, den der Anwenderprogrammierer vorher als Identifizierer für eine Datenstruktur vergeben hat ②.

Syntaxdiagramm der Strukturzuweisung und Initialisierung:



Das Feld zwischen den Operatoren <. . .>, die bei einer Strukturzuweisung immer anzugeben sind, zeigt alle Möglichkeiten, den Fundamentalvariablen in einer Strukturvariablen aktuelle Werte zuzuweisen ①.

Dies bedeutet, daß die Werte, die den Fundamentalvariablen bei der Strukturvereinbarung zugewiesen wurden, durch die Strukturzuweisung überschrieben werden können (Variablen-Override).

Dies ist möglich, wenn folgende Voraussetzungen erfüllt sind:

- Der **Typ** von "Initialisierungswert" muß identisch sein mit dem Typ der in Datenstruktur vereinbarten Fundamentalvariablen.
- Ist für eine Fundamentalvariable in der Datenstruktur eine Liste von Werten spezifiziert, kann diese Liste **nicht** überschrieben werden. D.h. es kann nur ein einzelner Wert, der einer Fundamentalvariablen zugewiesen wurde, überschrieben werden.
- Ist in der Datenstruktur mit `DB` ein String vereinbart, kann dieser nur durch einen anderen String überschrieben werden. Ist er kürzer als der Originalstring, bleiben die restlichen Zeichen des Originalstring erhalten.

Ein Originalstring kann durch einen längeren Override-String **nicht** überschrieben werden.

Beispiel 1: Überschreibbare Fundamentalvariablen

```
DATEN_STRUKTUR    STRUC
EIN_STRING        DB 'HELLO'
UNBESTIMMT        DW ?
REAL_ZAHL         DD 3.14159
DATEN_STRUKTUR    ENDS
```

Wird eine Strukturvariable vom Typ DATEN_STRUKTUR definiert, können durch die Strukturzuweisung die angegebenen Werte der vereinbarten Fundamentalvariablen z.B. mit `VAR_1 DATEN_STRUKTUR <'HALLO', 125, 6.2815>` überschrieben werden.

Beispiel 2: Nichtüberschreibbare Fundamentalvariablen

```
DATEN_STRUKTUR_1 STRUC
EINE_LISTE DB 1, 2, 3
KOPIEN  DW  10 DUP (?)
DATEN_STRUKTUR_1 ENDS
```

Wird eine Strukturvariable vom Typ DATEN_STRUKTUR_1 definiert, können durch die Strukturzuweisung die angegebenen Werte der vereinbarten Fundamentalvariablen **nicht** überschrieben werden. Sollte trotzdem versucht werden, die Fundamentalvariablen zu überschreiben, wird der Assembler eine entsprechende Fehlermeldung ausgeben. Existiert eine Datenstruktur mit überschreibbaren Fundamentalvariablen, können gezielt bestimmte Variablen überschrieben werden.

Beispiel:

```
LOC  OBJ                  LINE     SOURCE

                             1             NAME    BEISPIEL_11
                             2
----                         3     DATEN_STRUKTUR  STRUC
                             4
0000                         5     VAR_1   DB      0
0001                         6     VAR_2   DB      3
0002                         7     VAR_3   DB      3
                             8
----                         9     DATEN_STRUKTUR ENDS
                            10
                            11
                            12     ASSUME DS:DATEN_PROG
                            13
                            14
----                        15     DATEN_PROG SEGMENT
                            16
0000 00                     17     STRUK_VAR   DATEN_STRUKTUR <,2,>
0001 02
0002 03
                            18
----                        19     DATEN_PROG ENDS
                            20
                            21             END
```

Im angegebenen Programmfragment bleiben die Werte der Fundamentalvariablen VAR_1 und VAR_3 erhalten, während der Wert von VAR_2 überschrieben wird.

Die Auswahl einer oder mehrerer Fundamentalvariablen, deren Werte überschrieben werden sollen, geschieht in folgender Weise:

- Die Reihenfolge der Werte .... im Zuweisungs-Statement <....> entspricht der Reihenfolge der vereinbarten Variablen in der Struktur.
- Soll nun eine bestimmte Variable überschrieben werden, wird der neue Wert an der entsprechenden Position im Zuweisungs-Statement angegeben.
- Variablen die nicht überschrieben werden sollen, werden im Zuweisungs-Statement mit (,) (Empty Override) markiert.

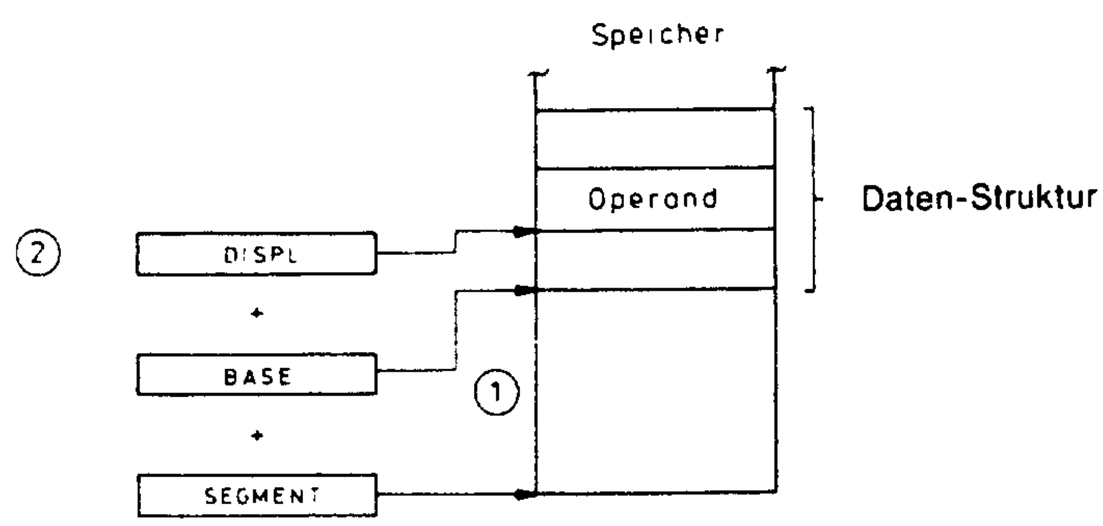
Es soll eine Datenstruktur existieren, deren Fundamentalvariablen nicht überschreibbar sind. Wird die Datenstruktur einer Strukturvariablen zugewiesen, sind im Zuweisungs-Statement nur die < >-Operatoren anzugeben. Dadurch wird eine Überschreibung vermieden.

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_12
                               2
----                           3     DATEN_STRUKTUR  STRUC
                               4
0000                           5     VAR_1   DB      0,1,2,3,4
0005                           6     VAR_2   DB      5,6,7,8,9
                               7
----                           8     DATEN_STRUKTUR  ENDS
                               9
                              10
                              11     ASSUME DS:DATEN_PROG
                              12
                              13
----                          14     DATEN_PROG SEGMENT
                              15
0000 00                       16     STRUK_VAR_1     DATEN_STRUKTUR <>
0001 01
0002 02
0003 03
0004 04
0005 05
0006 06                  ①
0007 07
0008 08
0009 09
000A 00                       17     STRUK_VAR_2     DATEN_STRUKTUR <>
000B 01
000C 02
000D 03
000E 04
000F 05
0010 06
0011 07
0012 08
0013 09
                              18
----                          19     DATEN_PROG ENDS
                              20
                              21             END
```

In diesem Programmfragment sind die Strukturvariablen STRUK_VAR_1 und STRUK_VAR_2 vom Typ DATEN_STRUKTUR vereinbart worden. Die Liste der Werte in VAR_1 und VAR_2 wird wegen der Codierung von <> nicht überschrieben. D.h. die Strukturvariablen STRUK_VAR_1 und STRUK_VAR_2 werden jeweils mit beiden Listen initialisiert ①.

4.1.4.3 Adressierung von Strukturen

Um die Fundamentalvariablen innerhalb einer Strukturvariablen zu erreichen, kann der Anwenderprogrammierer die Based-Adressierung benützen.



Dabei zeigt SS:BP zum Anfang einer Strukturvariablen im Stack-Segment bzw. DS:BX zeigt zum Anfang einer Strukturvariablen im Daten-Segment ①. In beiden Fällen enthalten die Base-Register BX oder BP den Offset einer Strukturvariablen.

Gekennzeichnet wird eine Daten-Struktur durch den (.)-Operator. Beispielsweise gibt der Ausdruck [BX]. an, daß [BX] auf eine Strukturvariable zeigt. Die gewünschte Fundamentalvariable liegt dann in einem bestimmten Abstand relativ zum Anfang der Datenstruktur.

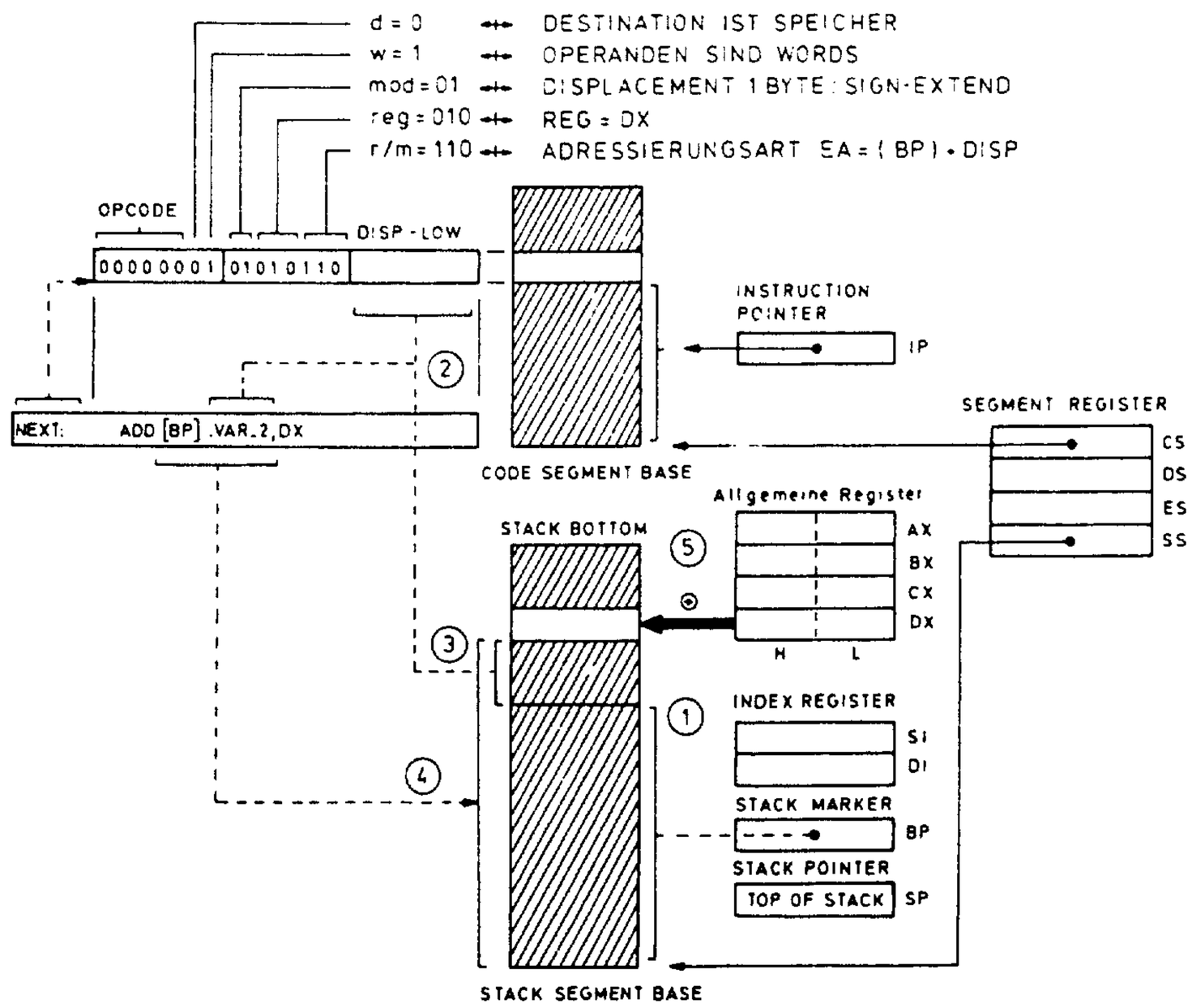
Beispiel:

```
ADD [BX]. VAR_1, AL
```

Bei diesem Befehl zeigt [BX] zum Anfang einer Strukturvariablen. VAR_1 ist eine Fundamentalvariable, die in einem bestimmten Abstand (in Bytes) relativ zum Beginn der Strukturvariablen liegt. Der Assembler hinterlegt diesen relativen Abstand als Offset im Displacement-Feld des Befehls ②.

Das angegebene Bild zeigt die Analyse des Befehlsbeispiels

```
NEXT: ADD [BP]. VAR_2, DX
```



Wegen [BP]. markiert [BP] den Anfang einer Strukturvariablen im Stack-Segment ①. Der Offset von VAR_2 ist als LOW-Displacement im Maschinencode hinterlegt ② und stellt den relativen Abstand zum Anfang der Strukturvariablen dar ③.

Die Summe von BP und VAR_2 ergibt im Stack-Segment die effektive Adresse der Fundamentalvariablen VAR_2 ④. Zum Wert der Variablen VAR_2 addiert der Befehl den Inhalt des allgemeinen Registers DX ⑤.

Im folgenden Programmbeispiel ist die Strukturvariable STRUK_VAR aus den Fundamentalvariablen VAR_1, VAR_2 und VAR_3 zusammengesetzt. In der Strukturzuweisung werden diese Variablen mit den Dezimalzahlen 0, 3 und 5 initialisiert.

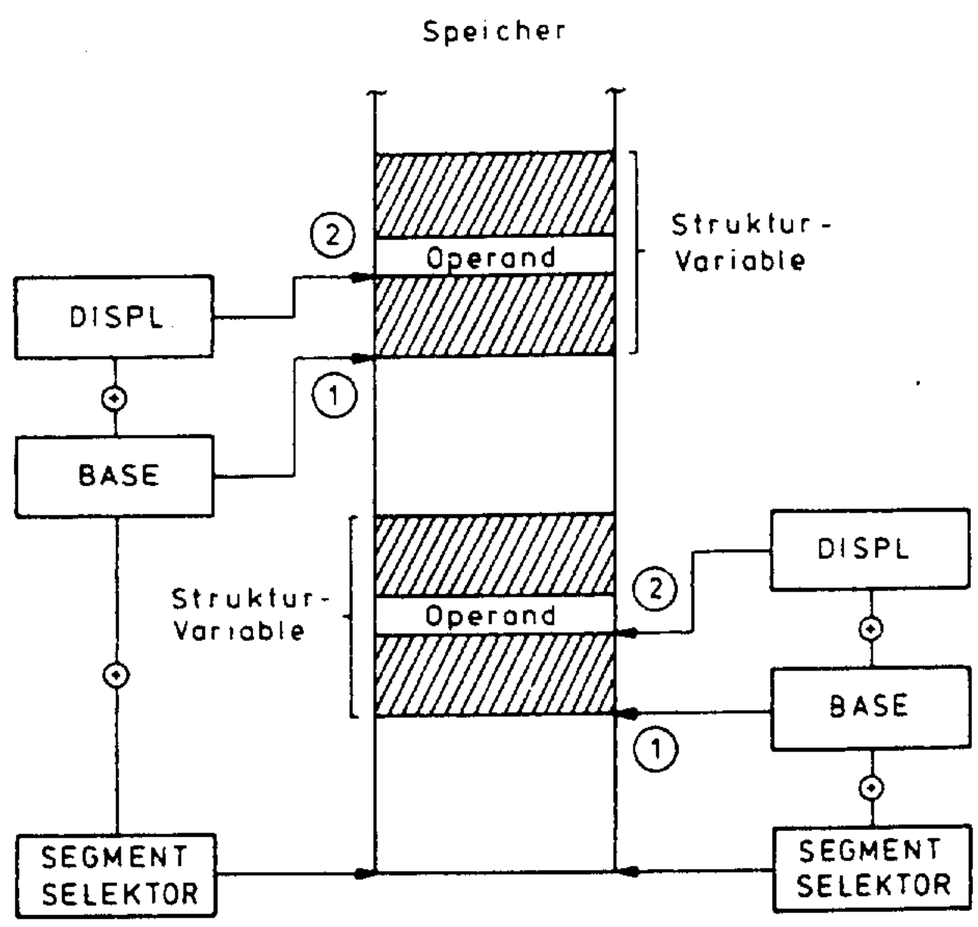
```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_13
                            2
----                        3     DATEN_STRUKTUR  STRUC
                            4
0000                        5     VAR_1   DB      ?
0001                        6     VAR_2   DB      ?
0002                        7     VAR_3   DB      ?
                            8
----                        9     DATEN_STRUKTUR  ENDS
                           10
                           11
                           12     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                           13
                           14
----                       15     DATEN_PROG  SEGMENT
                           16
0000 00                    17     STRUK_VAR   DATEN_STRUKTUR <0,3,5>      ;In STRUK_VAR werden VAR_1 mit 0,
0001 03
0002 05
                           18                                            ;VAR_2 mit 3 und VAR_3 mit 5
                           19                                            ;initialisiert
                           20
----                       21     DATEN_PROG  ENDS
                           22
                           23
----                       24     CODE_PROG   SEGMENT
                           25
0000 BB0000                26     START:  MOV     BX,OFFSET STRUK_VAR    ;[BX] zeigt zum Anfang der Struktur-
                           27                                            ;variablen STRUK_VAR
0003 B030                  28             MOV     AL,30H                 ;Konstante fuer die ASCII-Umwandlung
0005 0007                  29             ADD     [BX].VAR_1,AL          ;Umwandlung der in VAR_1,VAR_2 und
0007 004701                30             ADD     [BX].VAR_2,AL          ;VAR_3 gespeicherten Dezimalzahlen
000A 004702                31             ADD     [BX].VAR_3,AL          ;in ASCII-Zeichen
                           32
----                       33     CODE_PROG   ENDS
                           34
                           35             END     START
```

Der Befehl `MOV BX, OFFSET STRUK_VAR` lädt den Offset der Strukturvariablen STRUK_VAR in das Register BX. Damit zeigt [BX] zum Anfang von STRUK_VAR. Die einzelnen Fundamentalvariablen werden durch ihre effektiven Adressen erreicht:

```
[BX]. VAR_1
[BX]. VAR_2
[BX]. VAR_3
```

Zu ihren dezimalen Werten wird die Konstante 30H in einer Folge von Additionsbefehlen addiert. Das bedeutet, daß die Werte von VAR_1, VAR_2 und VAR_3 in ihre entsprechenden ASCII-Zeichen umgewandelt werden.

Die Adressierungsart Based wird im Allgemeinen dazu verwendet, um auf die gleichen Fundamentalvariablen in verschiedenen Strukturvariablen zuzugreifen.



Dabei zeigen die Base-Register BX oder BP auf den Anfang der Strukturvariablen ①.

Das Displacement selektiert anschließend die gewünschte Fundamentalvariable ②.

Im Programmbeispiel 14 sind zwei Strukturvariablen STRUK_VAR_1 und STRUK_VAR_2 vereinbart.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME BEISPIEL_14
                            2
----                        3     DATEN_STRUKTUR  STRUC
                            4
0000                        5     VAR_1   DB      0,1,2
0003                        6     VAR_2   DB      3,4,5
                            7
----                        8     DATEN_STRUKTUR  ENDS
                            9
                           10
                           11     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                           12
                           13
----                       14     DATEN_PROG SEGMENT
                           15
0000 00                    16     STRUK_VAR_1     DATEN_STRUKTUR <>
0001 01
0002 02
0003 03
0004 04
0005 05
0006 00                    17     STRUK_VAR_2     DATEN_STRUKTUR <>
```

```
LOC  OBJ                    LINE     SOURCE

0007 01
0008 02
0009 03
000A 04
000B 05
                              18
----                          19     DATEN_PROG ENDS
                              20
                              21
----                          22     CODE_PROG SEGMENT
                              23
0000 BB0000                   24     START:  MOV     BX,OFFSET STRUK_VAR_1   ;[BX] zeigt zum Anfang der Struktur-
                              25                                             ;variablen STRUK_VAR_1
0003 B030                     26             MOV     AL,30H                  ;Konstante fuer die ASCII-Umwandlung
0005 004705                   27     ASCI_1: ADD     [BX].VAR_2+2,AL         ;In STRUK_VAR_1 wird das zweite Byte
                              28                                             ;nach VAR_2 in ein ASCII-Zeichen
                              29                                             ;umgewandelt
0008 BB0600                   30             MOV     BX,OFFSET STRUK_VAR_2   ;[BX] zeigt zum Anfang der Struktur-
                              31                                             ;variablen STRUK_VAR_2
000B 004701                   32     ASCI_2: ADD     [BX].VAR_1+1,AL         ;In STRUK_VAR_2 wird das erste Byte
                              33                                             ;nach VAR_1 in ein ASCII-Zeichen
                              34                                             ;umgewandelt
                              35
----                          36     CODE_PROG ENDS
                              37
                              38             END     START
```

Der Befehl `MOV BX, OFFSET STRUK_VAR_1` legt den Zeiger [BX] zum Anfang von STRUK_VAR_1. Der Befehl `MOV BX, OFFSET STRUK_VAR_2` legt den Zeiger [BX] zum Anfang von STRUK_VAR_2.

Die beiden Fundamentalvariablen VAR_1 und VAR_2 sind als Feldvariable mit je drei Elementen vereinbart. Dabei werden bestimmte Elemente in den Feldvariablen durch ihre effektiven Adressen erreicht.

$$[BX].\ \underbrace{VAR_2 + 2}_{\text{Zweites Element nach VAR_2}}$$

bzw.

$$[BX].\ \underbrace{VAR_1 + 1}_{\text{Erstes Element nach VAR_1}}$$

4.1.4.4 Direkt Offset

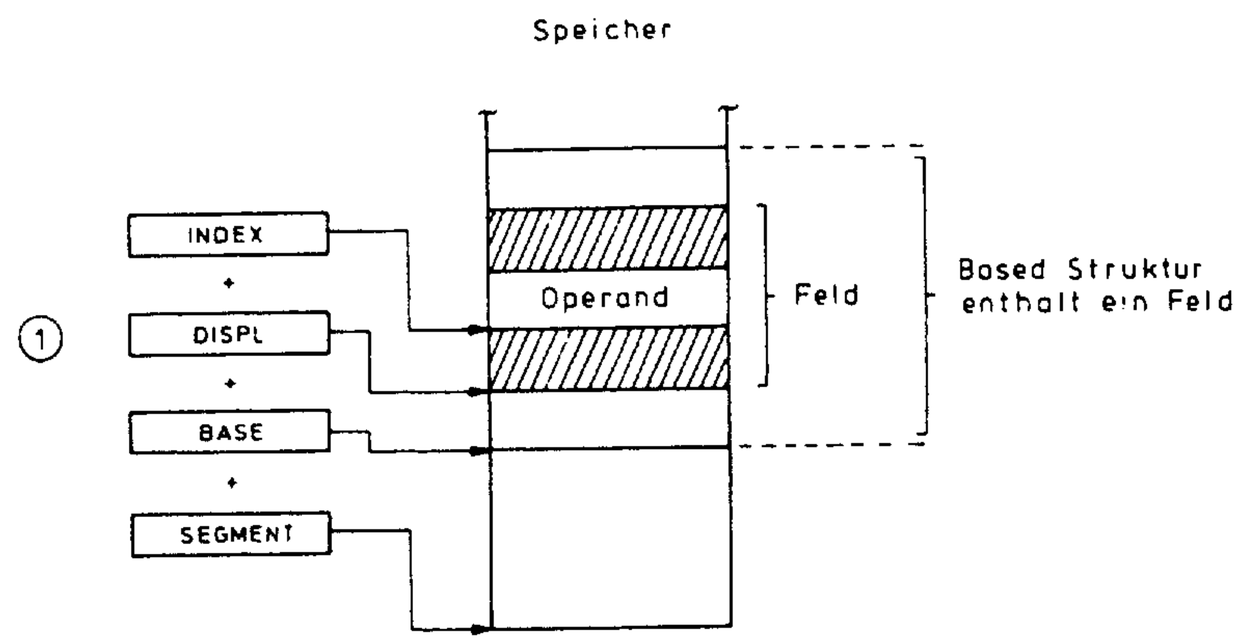
Um auf die gleichen Fundamentalvariablen in verschiedenen Strukturvariablen zuzugreifen, ist auch die Direkt-Offset-Adressierung verwendbar. Im folgenden Beispiel sind die Strukturvariablen STRUK_VAR_1 und STRUK_VAR_2 vereinbart. Sie sind aus den Fundamentalvariablen VAR_1 und VAR_2 zusammengesetzt.

```
LOC  OBJ                  LINE     SOURCE

                           1              NAME    BEISPIEL_DIREKT_OFFSET
                           2
----                       3     DATEN_STRUKTUR  STRUC
                           4
0000                       5     VAR_1   DB      0,1,2
0003                       6     VAR_2   DB      3,4,5
                           7
----                       8     DATEN_STRUKTUR  ENDS
                           9
                          10
                          11     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                          12
                          13
----                      14     DATEN_PROG SEGMENT
                          15
0000 00                   16     STRUK_VAR_1     DATEN_STRUKTUR <>
0001 01
0002 02
0003 03
0004 04
0005 05
0006 00                   17     STRUK_VAR_2     DATEN_STRUKTUR <>
0007 01
0008 02
0009 03
000A 04
000B 05
                          18
----                      19     DATEN_PROG ENDS
                          20
                          21
----                      22     CODE_PROG SEGMENT
                          23
0000 B030                 24     START:  MOV     AL,30H                  ;Konstante fuer die ASCII-Umwandlung
0002 00060500             25     ASCI_1: ADD     STRUK_VAR_1.VAR_2+2,AL  ;In STRUK_VAR_1 wird das zweite Byte
                          26                                             ;nach VAR_2 in ein ASCII-zeichen
                          27                                             ;umgewandelt
0006 00060700             28     ASCI_2: ADD     STRUK_VAR_2.VAR_1+1,AL  ;In STRUK_VAR_2 wird das erste Byte
                          29                                             ;nach VAR_1 in ein ASCII-Zeichen
                          30                                             ;umgewandelt
                          31
----                      32     CODE_PROG ENDS
                          33
                          34             END     START
```

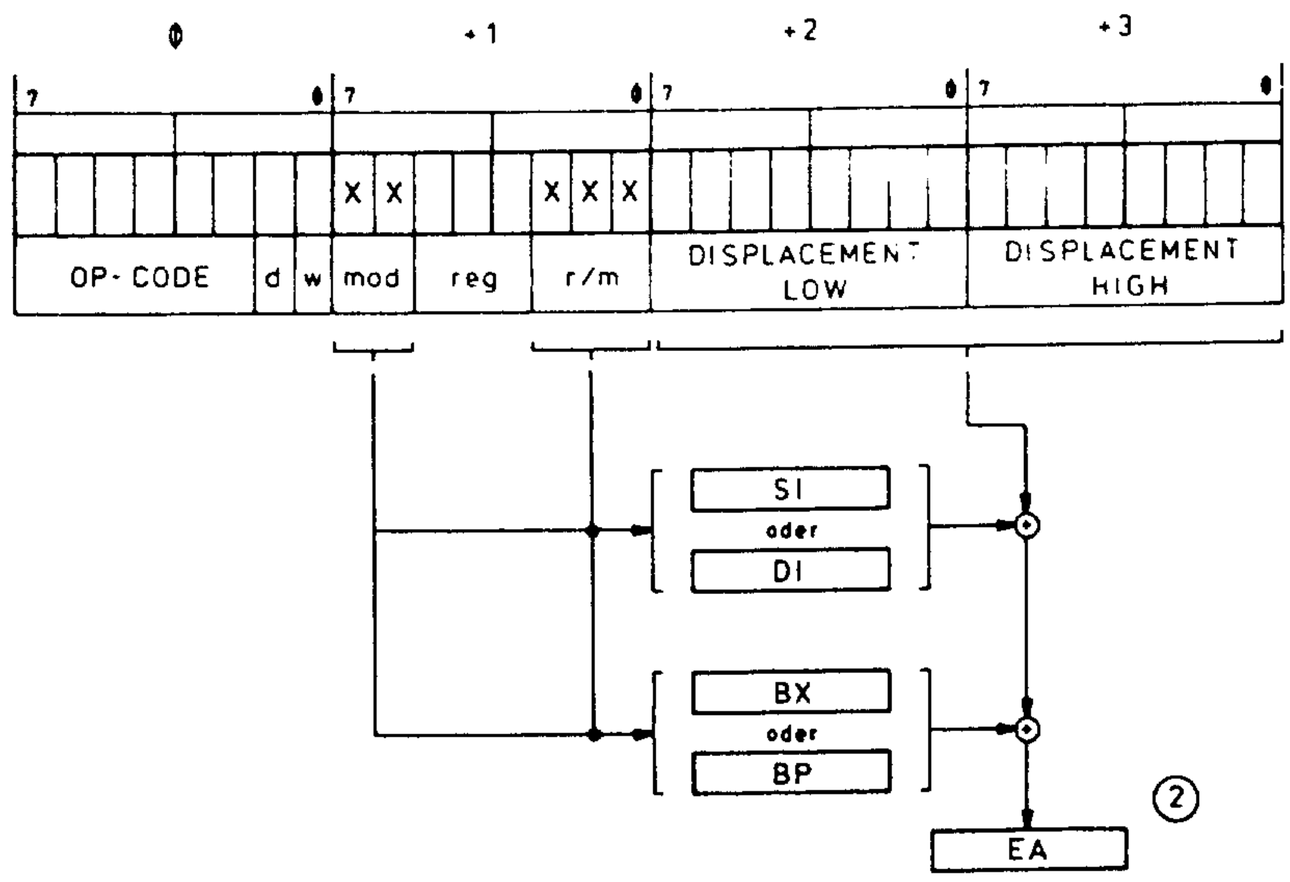
Bei den Befehlen `ADD STRUK_VAR_1. VAR_2+2, AL` und `ADD STRUK_VAR_2. VAR_1+1, AL` selektieren STRUK_VAR_1. und STRUK_VAR_2. direkt die Strukturvariablen. Die Fundamentalvariablen VAR_2 und VAR_1, die als Feldvariable vereinbart sind, geben die Abstände relativ zum Anfang der selektierten Strukturvariablen an, während die Ausdrücke +2 bzw. +1 die Abstände relativ zum Anfang der Feldvariablen festlegen.

4.1.5 Based-Indexed mit Displacement

Die Adressierungsart Based-Indexed mit Displacement generiert die effektive Adresse einer Variablen, indem sie die Summe aus einem Base-Register (BX oder BP), einem Index-Register (SI oder DI) und einem Displacement bildet ①.

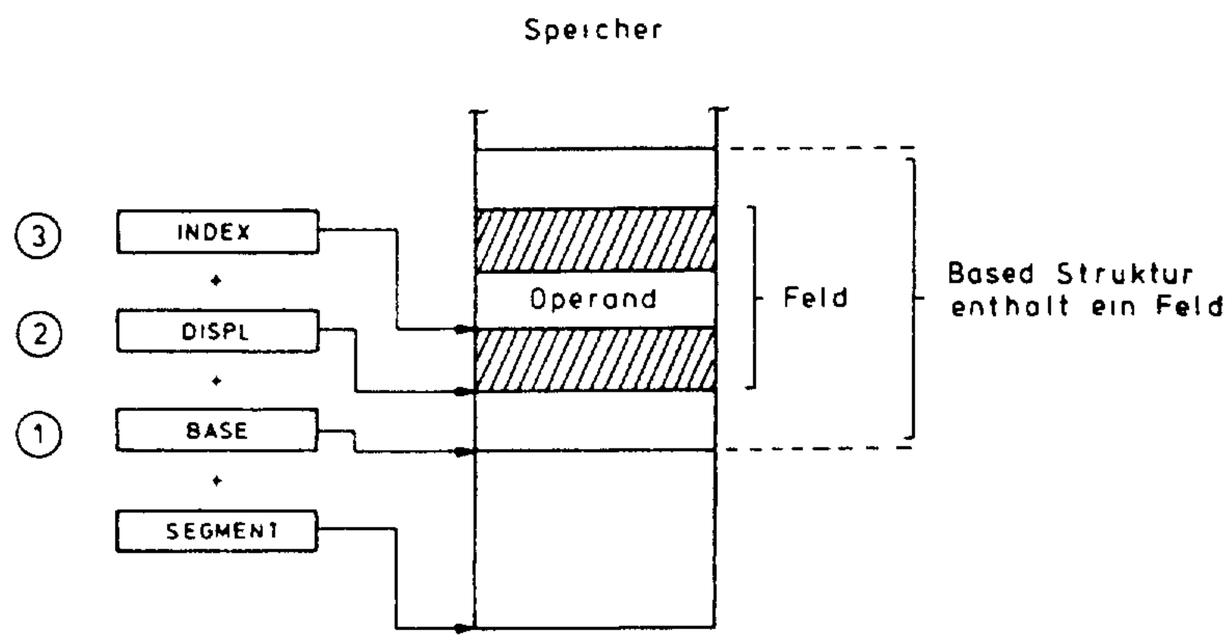


Abhängig davon, welches Registerpaar der Anwenderprogrammierer wählt, belegt der Assembler das r/m- und mod-Feld im Adreßmodus-Byte mit einer entsprechenden Bitkombination.



Diese Bitkombination kennzeichnet das ausgewählte Registerpaar (BX oder BP + SI oder DI). Zusammen mit dem im Befehl codierten Displacement bilden sie die effektive Adresse eines Speicheroperanden ②.

Die Adressierungsart Based-Indexed mit Displacement wird im Allgemeinen dazu verwendet, um Feldelemente in einer Datenstruktur zu adressieren. Dabei zeigt das Base-Register BX oder BP auf den Anfang der Datenstruktur ①. Das Displacement gibt den Abstand zwischen Startadresse der Datenstruktur und Anfangsadresse des Feldes an ②. Das Index-Register selektiert schließlich das gewünschte Feldelement ③.



Befehlsbeispiele:

```
ADD [BP-14] [SI], BL
CMP VAR_1 [BX] [SI], 9378H
```

Bei beiden Befehlen werden zur Bildung der effektiven Adresse zwei Register (Base- und Index-Register) und ein im Befehl codiertes Displacement benützt. Kann oder soll der Abstand zwischen der Basis einer Datenstruktur und eines Feldes explizit spezifiziert werden, ist es möglich, daß das Displacement als Zahl im Befehl angegeben wird. Zeigt SS:BP zur Basis einer Datenstruktur, kann sich [BP-14] als "Offset-14" innerhalb der Datenstruktur gedacht werden.

Hinweis: Die beiden Ausdrücke `[BX] [SI] [6]` und `[BX + SI + 6]` sind identisch.

Soll eine Strukturvariable adressiert werden, ist sie durch den (.)-Operator zu kennzeichnen.

Beispiel:

```
ADD [BX]. FELD [DI], 30H
```

Bei diesem Befehl zeigt [BX] zum Anfang einer Strukturvariablen. FELD ist der Name einer Feldvariablen. Der Wert von FELD ist der relative Abstand zum Anfang der Strukturvariablen und entspricht dem Displacement des Befehls. Der Wert in DI entspricht dem Index, bzw. dem Offset relativ zu FELD.

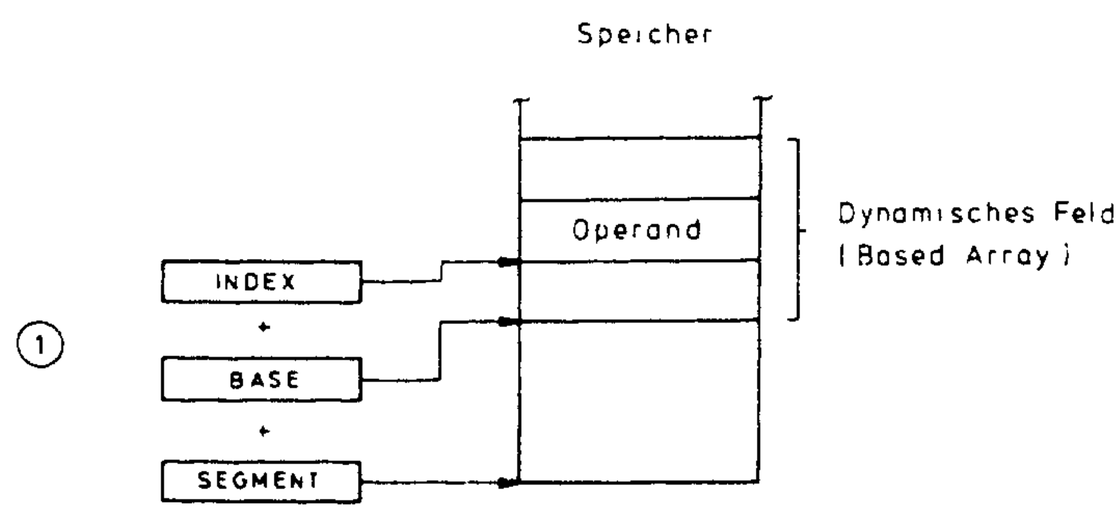
```
LOC  OBJ                LINE     SOURCE

                          1              NAME    BEISPIEL_15
                          2
----                      3     DATEN_STRUKTUR  STRUC
                          4
0000                      5     VAR_1   DW      ?
0002                      6     FELD    DB      0,1,2,3,4,5,6,7,8,9
000C                      7     VAR_2   DW      ?
                          8
----                      9     DATEN_STRUKTUR  ENDS
                         10
                         11
                         12     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                         13
                         14
----                     15     DATEN_PROG SEGMENT
                         16
0000 ????                17     STRUK_VAR   DATEN_STRUKTUR <>
0002 00
0003 01
0004 02
0005 03
0006 04
0007 05
0008 06
0009 07
000A 08
000B 09
000C ????
                         18
----                     19     DATEN_PROG ENDS
                         20
----                     21     CODE_PROG SEGMENT
                         22
0000 BB0000              23     START:  MOV     BX,OFFSET STRUK_VAR     ;[BX] zeigt zum Anfang der Struktur-
                         24                                             ;variablen STRUK_VAR
0003 BF0000              25             MOV     DI,0                    ;[DI] entspricht dem Index des ersten
                         26                                             ;Feldelements in FELD
0006 B90A00              27             MOV     CX,10                   ;CX ist der Wiederholungszaehler
0009 80410230            28     ASCII:  ADD     [BX].FELD[DI],30H       ;Umwandlung einer Dezimalzahl in ein
                         29                                             ;ASCII-Zeichen
000D 47                  30             INC     DI                      ;Feld Index inkrementieren
000E E2F9                31             LOOP    ASCII                   ;Wiederhole solange, bis alle Feld-
                          32                                              ;elemente in ASCII Zeichen umgewan-
delt
                         33                                             ;sind
----                     34     CODE_PROG ENDS
                         35
                         36             END     START
```

In diesem Programmbeispiel existiert die Strukturvariable STRUK_VAR vom Typ DATEN_STRUKTUR. Wobei DATEN_STRUKTUR aus drei Fundamentalvariablen VAR_1, FELD und VAR_2 zusammengesetzt sind. VAR_1 und VAR_2 sind WORD-Variable, während FELD eine Feldvariable mit zehn Elementen ist.

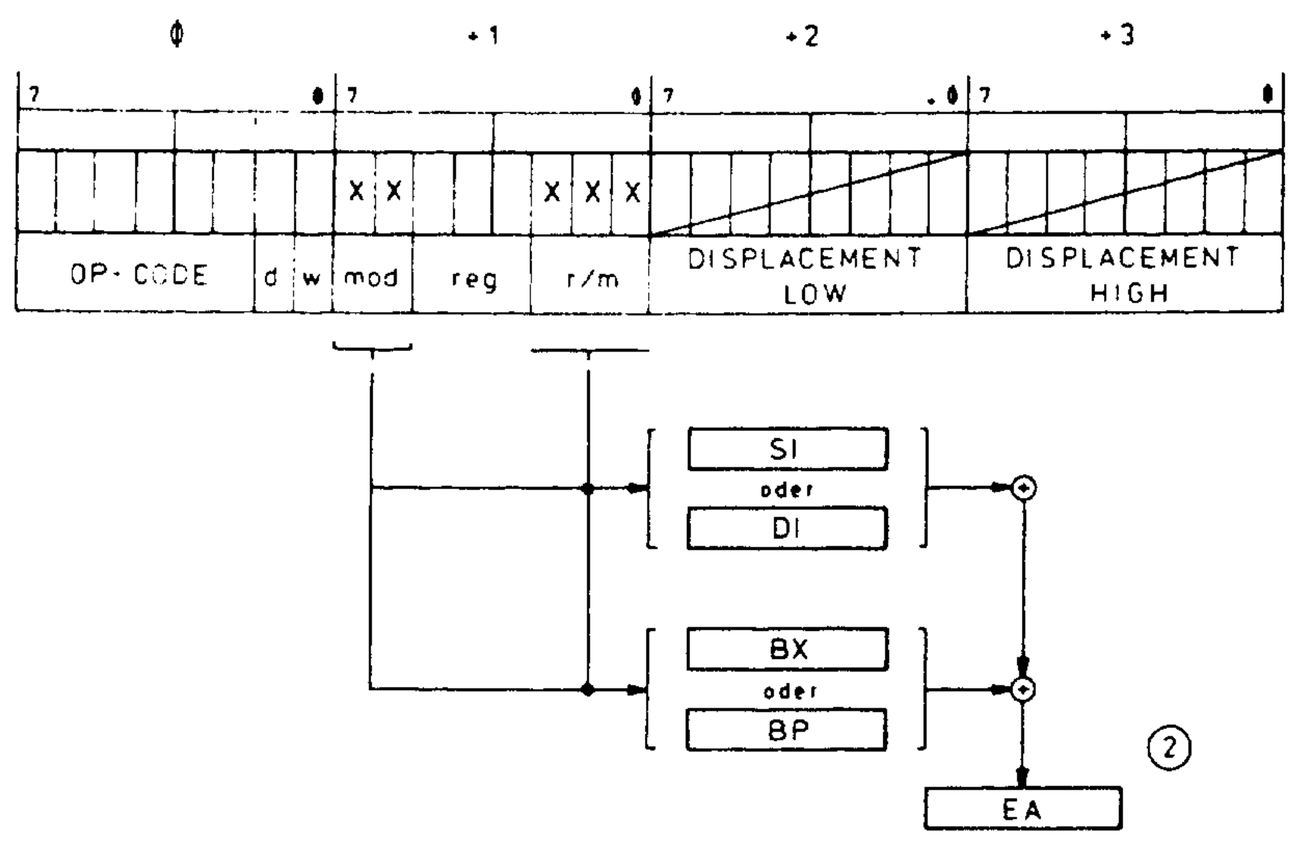
Die Feldelemente sind mit ungepackten Dezimalzahlen initialisiert. Unter Anwendung der Adressierungsart Based-Indexed mit Displacement werden diese Dezimalzahlen in ASCII-Zeichen umgewandelt.

4.1.6 Based Indexed

Bei der Based-Indexed-Adressierung wird die effektive Adresse einer Variablen generiert, indem die Summe aus einem Base-Register (BX oder BP) und einem Index-Register (SI oder DI) gebildet wird ①.



Abhängig davon, welches Registerpaar der Anwenderprogrammierer wählt, belegt der Assembler das r/m- und mod-Feld im Adreßmodus-Byte mit einer entsprechenden Bitkombination.



Diese Bitkombination kennzeichnet das ausgewählte Registerpaar (BX oder BP + SI oder DI). Bei der Based-Indexed-Adressierung wird im Befehl kein Displacement codiert, d.h. die effektive Adresse einer Variablen entspricht der Summe des gewählten Registerpaares ②.

Die Adressierungsart Based-Indexed wird im Allgemeinen dazu verwendet, um auf die einzelnen Elemente in einem dynamischen Feld zuzugreifen. Die Anfangsadresse und die Länge eines dynamischen Feldes sind variabel, d.h. ihre Werte können sich während der Programmausführung ändern. Dies bedeutet, daß ein dynamisches Feld nicht durch einen Namen, sondern durch die Base-Register BX oder BP selektiert wird.

BX oder BP legen die Position bzw. die Anfangsadresse eines dynamischen Feldes fest (Based Array). Die Index-Register DI oder SI werden benützt, um innerhalb des dynamischen Feldes die gewünschten Feldelemente zu adressieren.

Befehlsbeispiel:

```
MOV [BX] [DI], AL
```

Bei diesem Beispiel zeigt DS:[BX] zum Anfang eines dynamischen Feldes, d.h. der Inhalt von BX ist veränderlich und wird während des Programmablaufs festgelegt. [DI] selektiert das gewünschte Feldelement. Der Befehl lädt in dieses Feldelement den Inhalt des Registers AL.

Das angegebene Programmbeispiel zeigt die Anwendung der Based-Indexed-Adressierung. Im statischen Feld UNPACKED_DEZIMAL sind ungepackte Dezimalzahlen gespeichert. Das Programm packt jeweils zwei Dezimalzahlen in ein Byte und speichert sie in einem dynamischen Feld.

```
LOC  OBJ                  LINE     SOURCE

                             1             NAME    BEISPIEL_16
                             2
                             3     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                             4
----                         5     DATEN_PROG SEGMENT
                             6
0000 (12                     7     PACKED_DEZIMAL      DB      12 DUP(?)
     ??
     )
000C 09                      8     UNPACKED_DEZIMAL    DB      9,8,7,6,5,4,3,2
000D 08
000E 07
000F 06
0010 05
0011 04
0012 03
0013 02
                             9
----                        10     DATEN_PROG ENDS
                            11
----                        12     CODE_PROG SEGMENT
```

```
LOC  OBJ                LINE    SOURCE

                         13
0000 BB0400              14     START:  MOV     BX,OFFSET PACKED_DEZIMAL+4  ;[BX] zeigt zum Anfang eines
                         15                                                 ;dynamischen Feldes
0003 BE0000              16             MOV     SI,0                        ;[SI] entspricht dem Index
                         17                                                 ;des ersten Elements im sta-
                         18                                                 ;tischen Feld UNPACKED_DEZIMAL
0006 8BFE                19             MOV     DI,SI                       ;[DI] entspricht dem Index
                         20                                                 ;des ersten Elements im dynami-
                         21                                                 ;schen Feld PACKED_DEZIMAL
0008 B90400              22             MOV     CX,4                        ;CX ist der Wiederholungszaehler
000B 8A440C              23     GEPACK: MOV     AL,UNPACKED_DEZIMAL[SI]     ;Ungepackte Dezimalzahl (nieder-
                         24                                                 ;wertiges Byte) in AL laden
000E 46                  25             INC     SI
000F 8A640C              26             MOV     AH,UNPACKED_DEZIMAL[SI]     ;Ungepackte Dezimalzahl (hoeher-
                         27                                                 ;wertiges Byte) in AH laden
0012 D0E4                28             SHL     AH,1                        ;Hoeherwertiges Byte mit 16
0014 D0E4                29             SHL     AH,1                        ;multiplizieren
0016 D0E4                30             SHL     AH,1
0018 D0E4                31             SHL     AH,1
001A 02C4                32             ADD     AL,AH                       ;Packe zwei Dezimalzahlen in
                         33                                                 ;ein Byte
001C 8801                34             MOV     [BX][DI],AL                 ;Speichere gepackte Dezimalzahl
001E 46                  35             INC     SI                          ;Bilde Index zur naechsten
                         36                                                 ;ungepackten Dezimalzahl
001F 47                  37             INC     DI                          ;Bilde Index zur naechsten
                         38                                                 ;gepackten Dezimalzahl
0020 E2E9                39             LOOP    GEPACK                      ;Wiederhole bis der Zaehler
                         40                                                 ;CX=0 ist
                         41
----                     42     CODE_PROG ENDS
                         43
                         44             END START
```

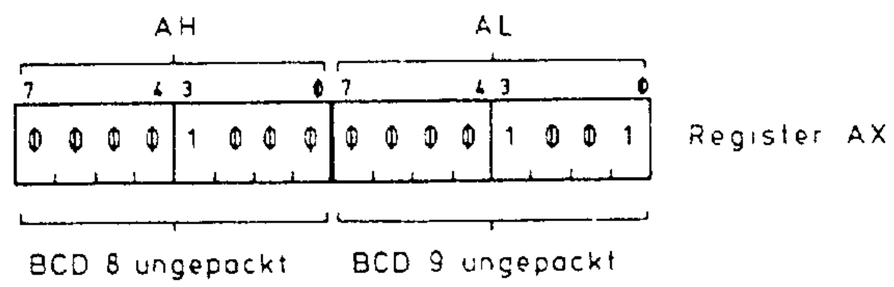
Im DATEN_PROG-Segment ist das statische Feld UNPACKED_DEZIMAL definiert und mit acht ungepackten Dezimalzahlen initialisiert. Die Feldvariable UNPACKED_DEZIMAL hat den Typ BYTE. Für die dynamische Feldvariable PACKED_DEZIMAL sind zwölf Bytes reserviert. Dieser Speicherbereich wird benützt, um darin ab beliebigen Anfangspositionen Felder variabler Länge zu speichern.

Der Befehl `MOV BX, OFFSET PACKED_DEZIMAL+4` legt die Anfangsposition des dynamischen Feldes fest. D.h. DS: [BX] zeigt zum ersten Feldelement. Es liegt im Abstand von vier Bytes innerhalb des reservierten Speicherbereichs von PACKED_DEZIMAL entfernt.

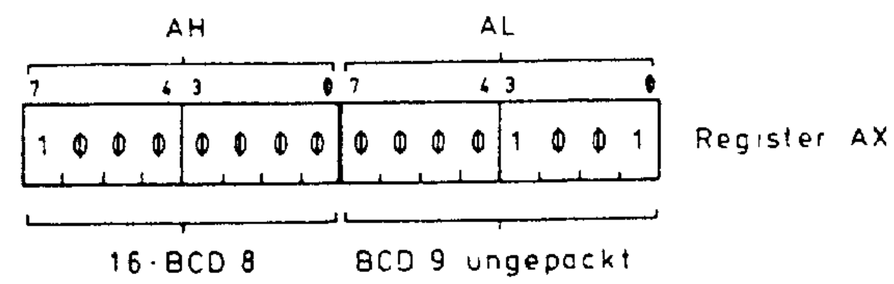
Der Befehl `MOV SI, 0` initialisiert das SI-Register mit dem ersten Index des statischen Quellfeldes UNPACKED_DEZIMAL. Der Befehl `MOV DI, SI` initialisiert das DI-Register mit dem ersten Index des dynamischen Zielfeldes PACKED_DEZIMAL.

Die beiden Befehle `MOV AL, UNPACKED_DEZIMAL [SI]` und `MOV AH, UNPACKED_DEZIMAL [SI]` laden zwei ungepackte Dezimalzahlen in das AX-Register.

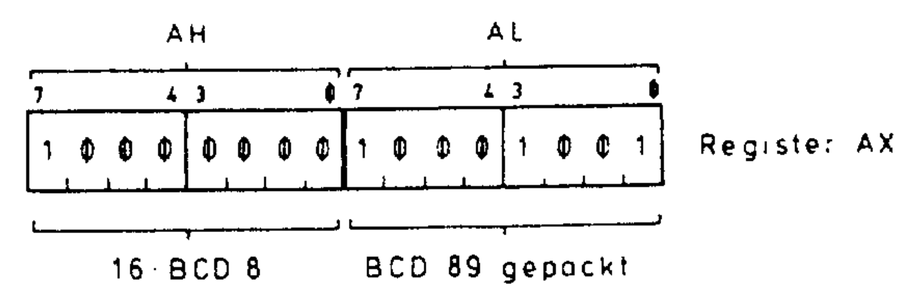
Bei beiden Befehlen wird die Indexed-Adressierung verwendet.



Die Befehle `SHL AH, 1` ... `SHL AH, 1` schieben die LOW-Tetrade von AH in die High-Tetrade von AH und füllen die LOW-Tetrade mit Nullen auf. Das AX-Register hat dann folgenden Datenbestand:



Der Befehl `ADD AL, AH` bewirkt, daß jeweils zwei der ursprünglich ungepackten Dezimalzahlen in ein Byte (AL Register) gepackt werden. Das AX-Register hat danach folgenden Datenbestand:



Der Befehl `MOV [BX] [DI], AL` speichert die gepackte BCD-Zahl im dynamischen Feld. Dabei zeigt [BX] zum Anfang des dynamischen Feldes, während [DI] die Feldelemente adressiert.

Die Länge des dynamischen Zielfeldes wird durch den Wert im Wiederholungszähler CX festgelegt. Im Programmbeispiel ist CX mit dem Wert 4 initialisiert. Das LOOP-Statement bewirkt, daß die folgende Befehlssequenz solange wiederholt wird, bis CX den Wert 0 erreicht hat.

```
GEPACK:   -
          -
          -
          -
          INC DI
          LOOP GEPACK
```

Dies bedeutet, daß wegen `INC DI` das Register DI den oberen Indexwert 4 bekommt. Damit liegt die Länge des dynamischen Zielfeldes mit vier Feldelementen fest.

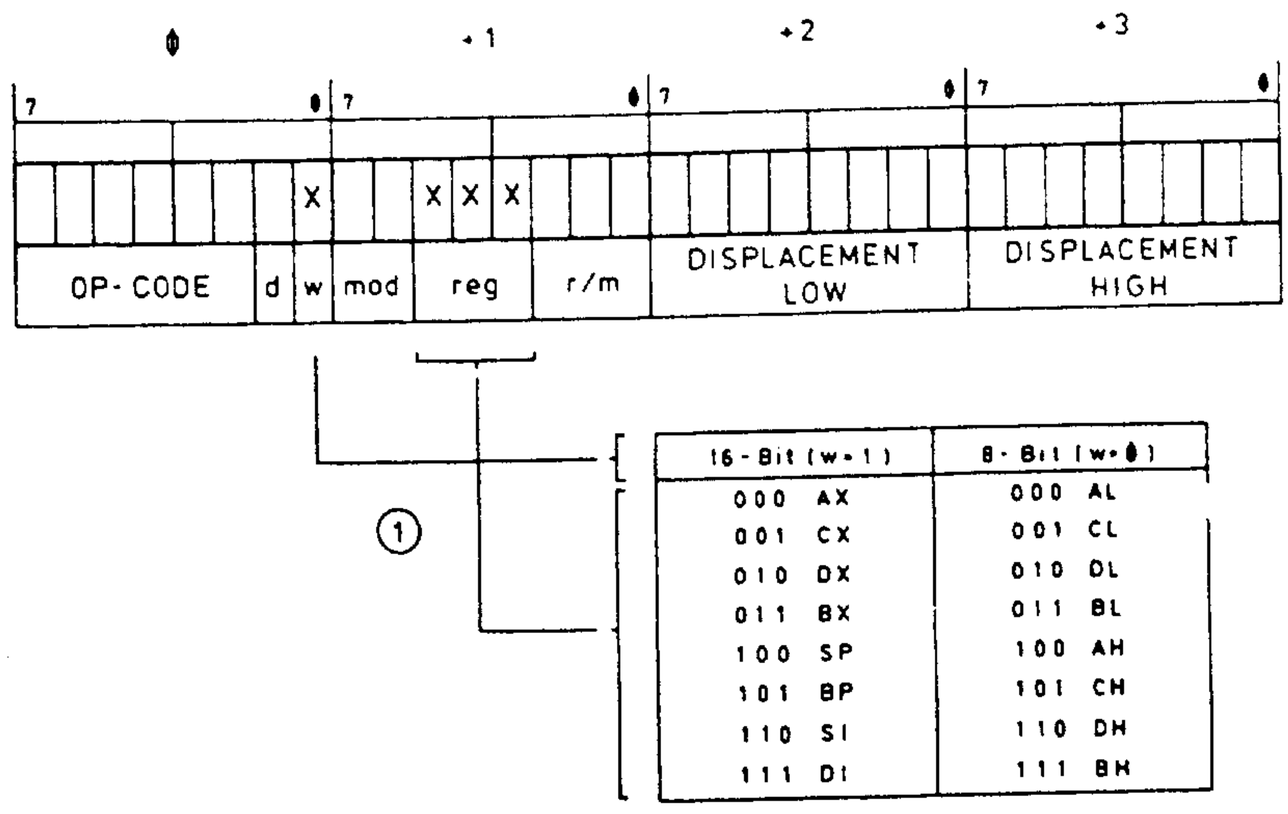
4.2 Zusammenfassung der Adressierungsarten

Die folgende Tabelle zeigt in Kurzform die 8086/80186/80286-Adressierungsarten.

Adressierungsart	Offset	Befehlsbeispiele
Direkt Offset	EA = DISP	MOV AX, VAR_1 ADD STRUK_VAR.VAR_1, AL
Register Indirekt	EA = (BX) oder (BP) oder (SI) oder (DI)	MOV AX, [BX] ADD [DI], AL
Based	EA = [(BX) oder (BP)] + DISP	ADD [BP]. VAR_1, AL MOV AX, VAR_2 [BX]
Indexed	EA = [(SI) oder (DI)] + DISP	ADD FELD [SI], AL
Based Indexed	EA = [(BX) oder (BP)] + [(SI) oder (DI)]	MOV [BX] [DI], AL
Based Indexed mit Displacement	EA = [(BX) oder (BP)] + [(SI) oder (DI)] + DISP	ADD [BP-6] [SI], CL SUB VAR [BX] [SI], AL

4.3 Register-Operanden

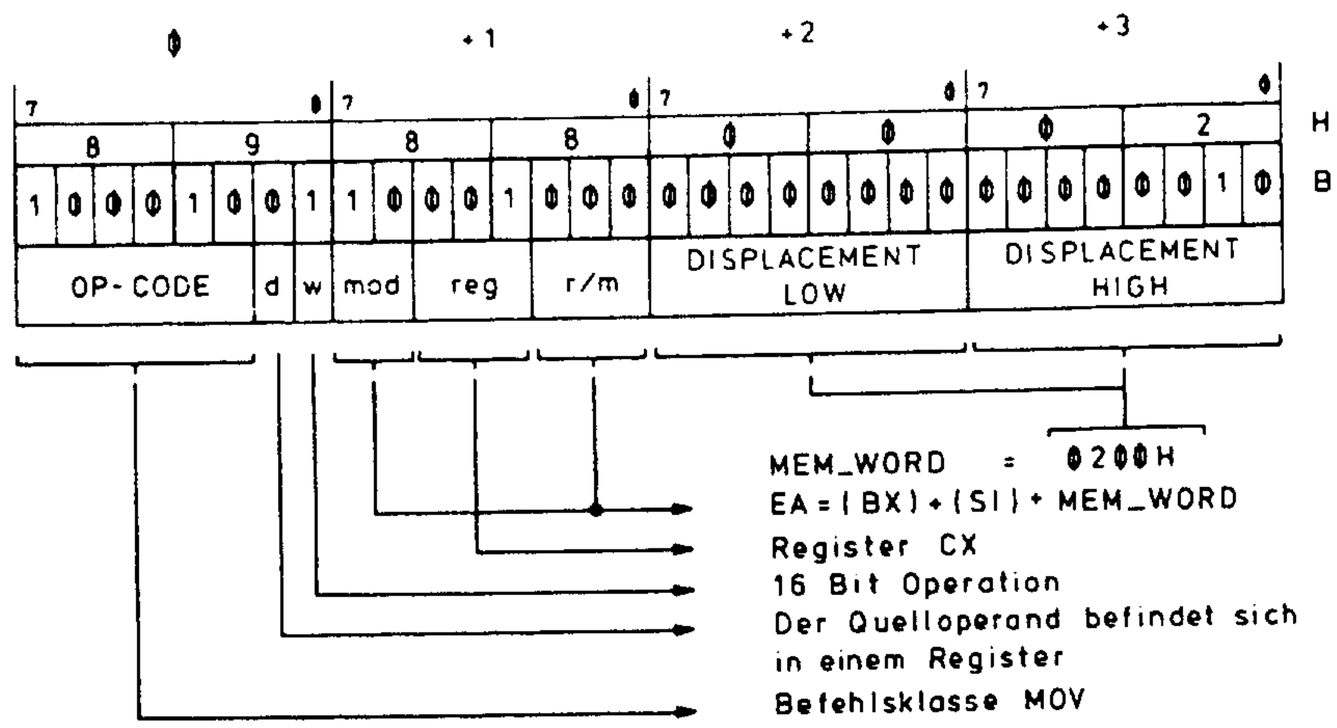
Register-Operanden können im Maschinencode von 8086/80186/80286-Befehlen durch unterschiedliche Felder gekennzeichnet sein.



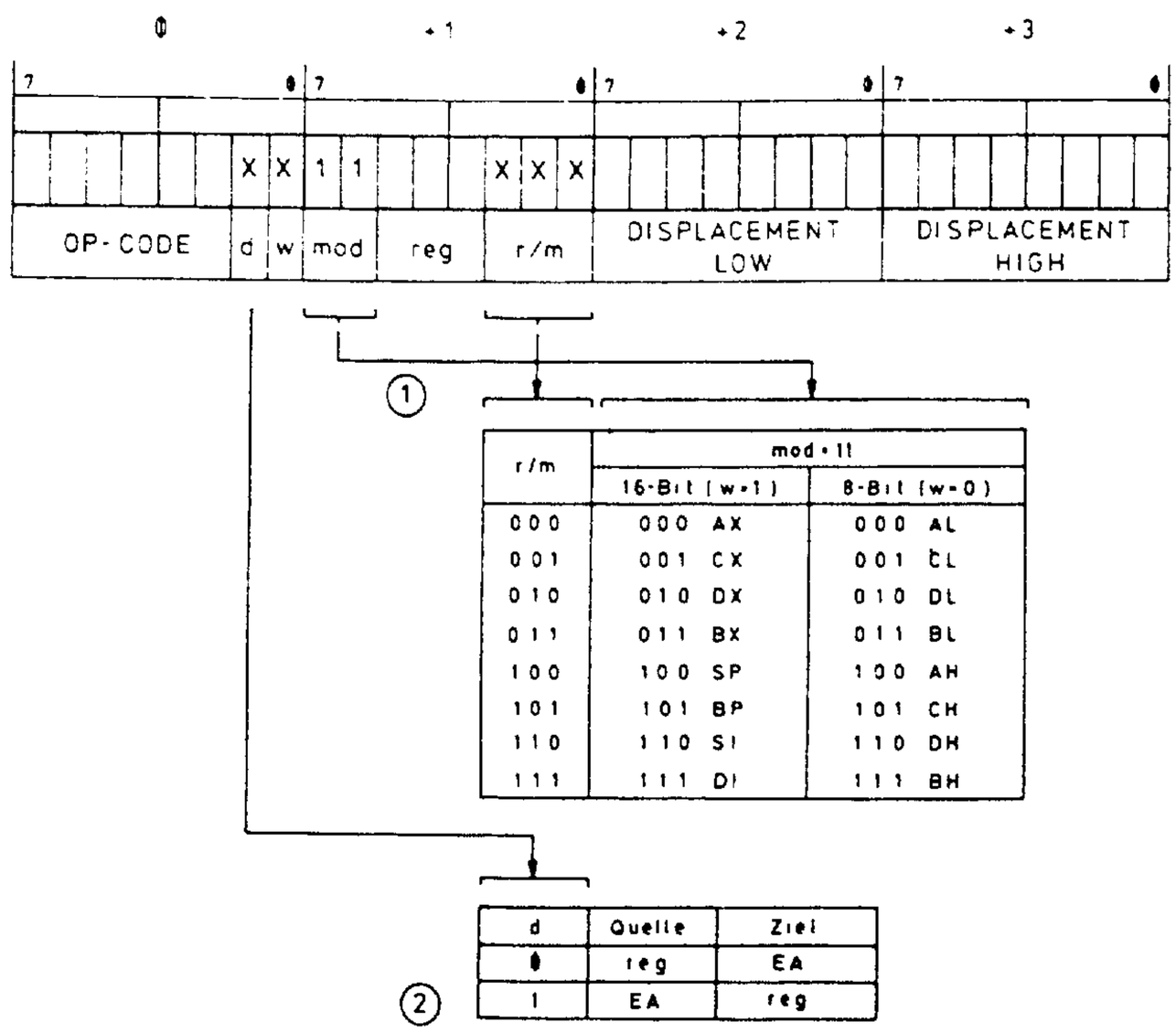
Ein mögliches Feld ist das **reg-Feld**, das zusammen mit dem w-Bit ein selektiertes Register repräsentiert ①.

Befehlsbeispiel:

```
MOV MEM_WORD [BX] [SI], CX = 89  88  00  02H
```



Als Weiteres besteht die Möglichkeit, ein selektiertes Register im **r/m-Feld** zusammen mit dem mod-Feld und den w- und d-Bits zu repräsentieren.



Nur für den Fall, daß das mod-Feld mit der Bitkombination 11 belegt ist, repräsentiert das r/m-Feld ein selektiertes Register ①. Beide Felder stellen normalerweise die effektive Adresse (EA) des Speicheroperanden dar.

Bei mod 11 repräsentiert das r/m-Feld ein Register, das als Zeiger auf sich selbst aufgefaßt werden kann. D.h. das selektierte Register wirkt als effektive Adresse.

Angewendet wird diese Register-Repräsentation bei Register/Register-Operationen.

Beispiel:

```
SUB   AX,  BX
```

Das Quellregister wirkt z. B. als effektive Adresse und wird im r/m-Feld codiert

Das Zielregister wird im reg-Feld codiert

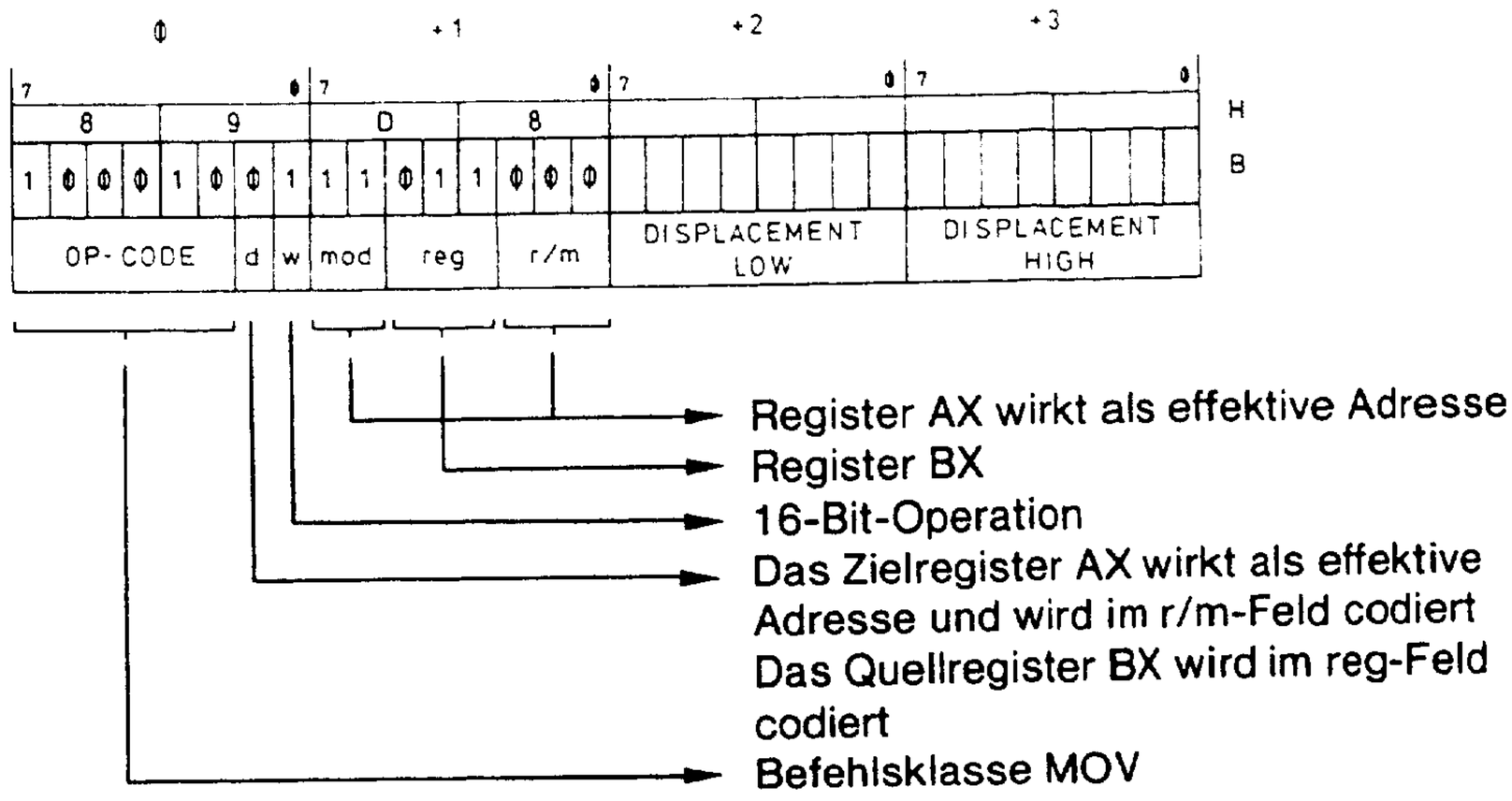
Das d-Bit wird in diesem Fall 1 sein ②.

Befehlsbeispiele für Register/Register-Operationen:

```
MOV   AX,  BX  =  89 D8H
```

Quellregister

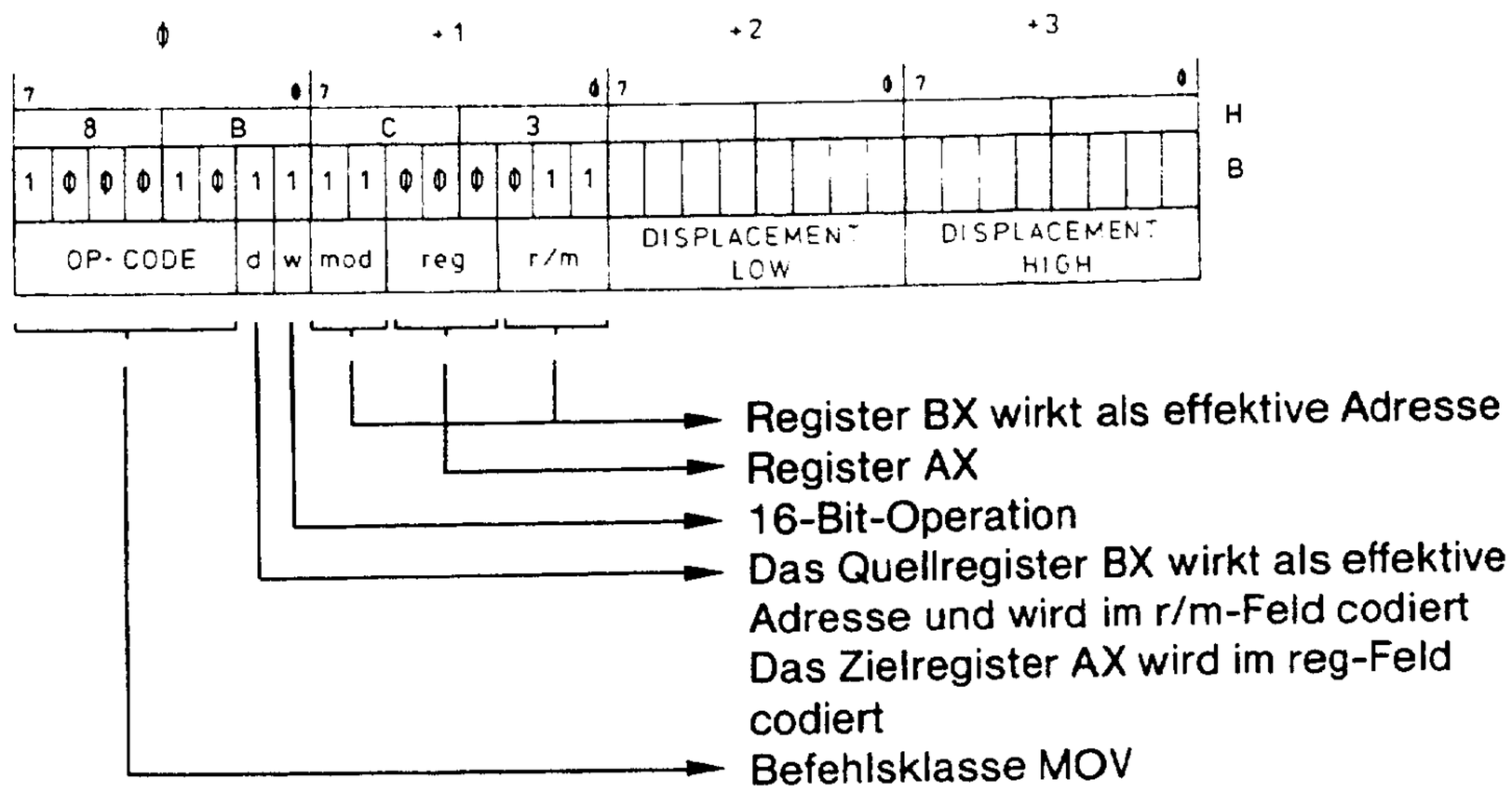
Zielregister = EA




```
MOV   AX,  BX  =  8B C3H
```

Quellregister = EA

Zielregister



Die Befehlsbeispiele zeigen, daß für gleiche Befehlsmnemoniks unterschiedliche Maschinencodes existieren, die in ihrer Wirkung identisch sind.

4.4 Immediate Operanden

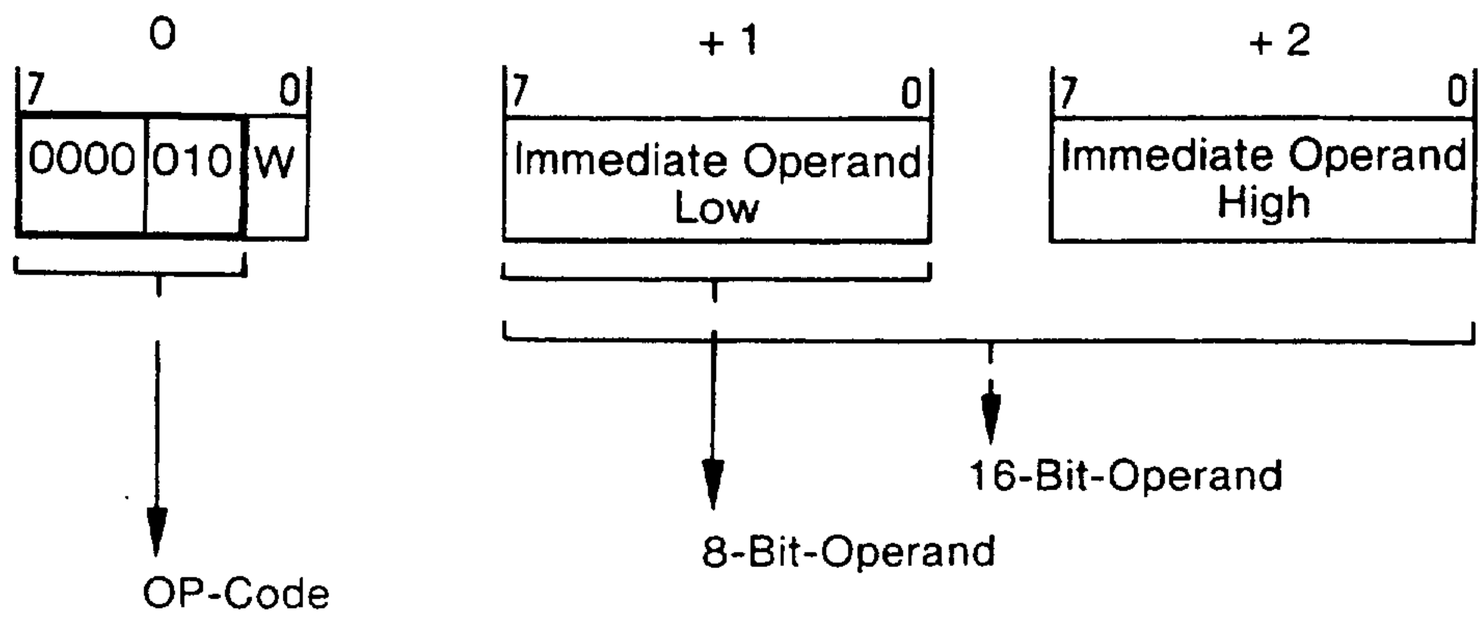
Ein immediate Operand ist ein konstanter Wert. Sie werden als Quelloperanden in 8086/80186/8028-Befehlen benützt. Im folgenden Beispiel ist AL das Zielregister und 5 der immediate Quelloperand: `SBB AL, 5`

In der Assemblersprache werden Speicher- und Registeroperationen von immediate Operationen im Opcode-Feld der Maschinencodes unterschieden.

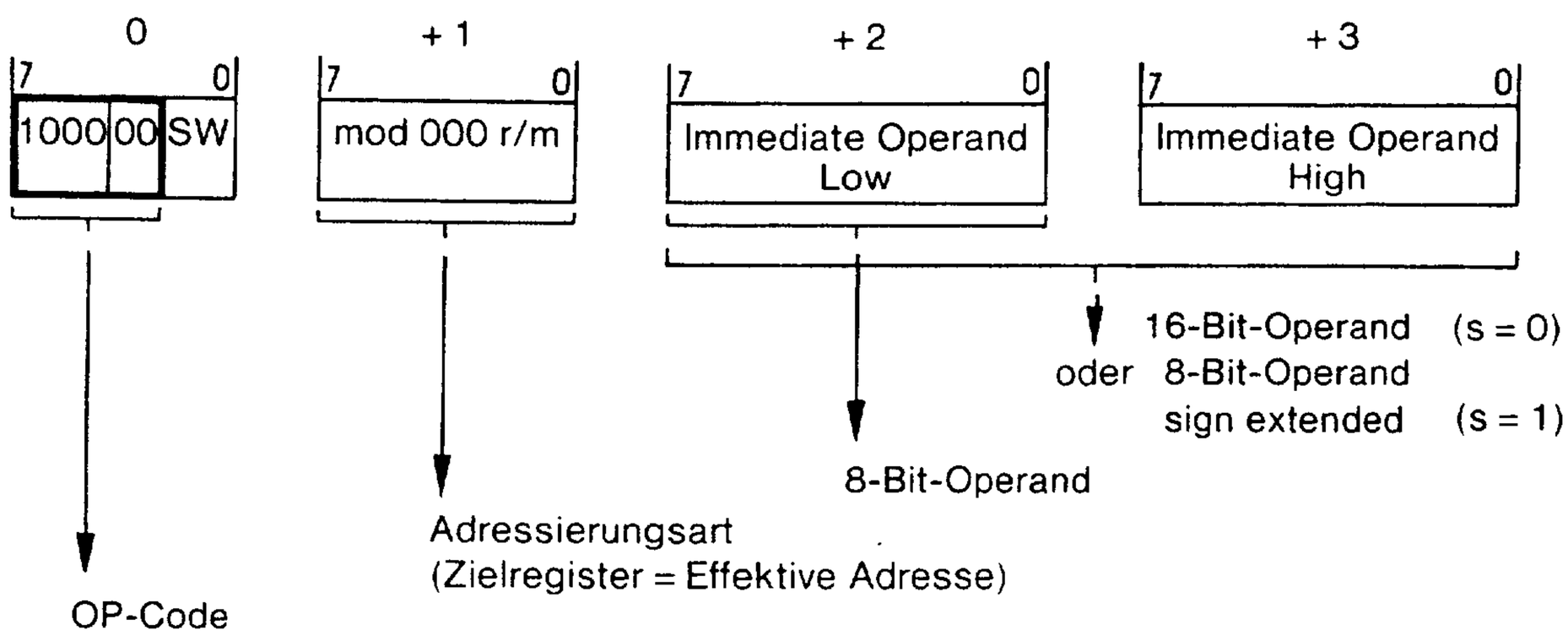
Beispiel: ADD (Addition)

In der Klasse der 8086/80186/80286-Additionsbefehle existieren zwei Gruppen von immediate Additionen.

1. Addiere einen immediate Operanden zum Akkumulator: `ADD AX, 0F0FH`



2. Addiere einen immediate Operanden zu einem Speicheroperanden oder einem Registeroperanden: `ADD DX, 0F0FH`



Die am Beispiel des ADD-Befehls gezeigten Gruppen von immediate Operationen existieren sinngemäß auch für folgende Befehlsklassen:

Klasse	Codier-Beispiele
ADC-Befehle	ADC AX, 256 ADC GAMMA, 30H ADC BX, 5
AND-Befehle	AND AL, 7FH AND BETA, 01H AND CX, 0F0H

CMP-Befehle	CMP AL, 0DH
	CMP SI, 0200H
	CMP [BX]. RADAR, 34H

MOV-Befehle	MOV AX, 9E23H
	MOV CL, 2
	MOV [BX] [SI], 2CH

OR-Befehle	OR AL, 6CH OR CX, 01H OR [BX]. VAR, 0CFH
SBB-Befehle	SBB AX, 2 SBB CL, 1 SBB ZAHL [SI], 10

SUB-Befehle	SUB AL, 10
	SUB SI, 5280
	SUB [BP]. VAR, 1000

TEST-Befehle	TEST AL, 20H TEST BX, 0CC4H TEST NUMMER, 01H
XOR-Befehle	XOR AX, 7C89H XOR SI, 00C2H XOR NUMMER, 0D2H

Nachfolgend sind noch einige Befehlsbeispiele mit immediate Operanden angegeben.

```
LOC  OBJ                LINE     SOURCE

                          1              NAME    BEISPIEL_17
                          2
                          3     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                          4
----                      5     DATEN_PROG SEGMENT
                          6
0000 00                   7     VAR     DB      0,1,2,3
0001 01
0002 02
0003 03
                          8
----                      9     DATEN_PROG ENDS
                         10
----                     11     CODE_PROG SEGMENT
                         12
0000 B005                13     START:  MOV     AL,5                ;AL=5
0002 3DFFFF              14             CMP     AX,0FFFFH           ;Vergleiche AX mit 0FFFFH
0005 800130              15             ADD     VAR[BX][DI],30H     ;Addiere 30H zu [BX]+[DI]+VAR
0008 3C0F   ①            16             CMP     AL,15 OR 5          ;Ein Ausdruck Beispiel - vergleiche
                         17                                         ;AL mit 15
000A 83C204              18             ADD     DX,(30+10)/10       ;Ein Ausdruck Beispiel - addiere
            ②            19                                         ;4 zu DX
                         20
----                     21     CODE_PROG ENDS
                         22
                         23             END     START
```

Die Befehle `CMP AL, 15 OR 5` und `ADD DX, (30 + 10)/10` zeigen, daß in Assembler immediate Operanden auch als Ausdrücke codiert werden können. Unter Ausdruck ist in diesem Fall z.B. die logische oder arithmetische Verknüpfung von Operanden oder wiederum von Ausdrücken zu verstehen. Als Ergebnis der Verknüpfung wird ein immediate Operand geliefert.

Für den Befehl `CMP AL, 15 OR 5` bedeutet dies, daß der Assembler während der Sprachübersetzung die Verknüpfung 15 ODER 5 durchführt und als Ergebnis den immediate Operanden 15 (0FH) liefert ①. Dabei ist **OR** ein **logischer Operator** des Assemblers.

Beim Befehl `ADD DX, (30 + 10)/10` ermittelt der Assembler den Wert des Ausdrucks (30 + 10)/10 und liefert dafür den immediate Operanden 4 ②. Bei diesem Befehlsbeispiel sind (+) (Addition) und (/) (Division) **arithmetische Operatoren** des Assemblers.

Sollen Ausdrücke verknüpft werden, sind sie in runde Klammern (...) einzuschließen. Dies bedeutet, daß vor der Verknüpfung zuerst die Werte der Klammerausdrücke ermittelt werden.

4.4.1 Übersicht über Assembler-Operatoren

Arithmetische Operationen		Syntax
+	(Addition)	Operand + Operand
–	(Subtraktion)	Operand – Operand
*	(Multiplikation)	Operand * Operand
/	(Division)	Operand / Operand
MOD	(Modulo Division)	Operand MOD Operand
SHL	(Shift links)	Operand SHL Zahl
SHR	(Shift rechts)	Operand SHR Zahl
HIGH	(High-Teil)	HIGH-Operand
LOW	(Low-Teil)	LOW-Operand

Hinweis: Operanden sind ausgewertete numerische Ausdrücke oder absolute Zahlen.

Das folgende Programmlisting zeigt Anwendungen von **arithmetischen** Assembler-Operatoren.

```
LOC  OBJ                LINE    SOURCE

                          1             NAME    BEISPIEL_18
                          2
                          3     ASSUME CS:CODE_PROG
                          4
----                      5     CODE_PROG SEGMENT
                          6
0000 3C08                 7     START:  CMP     AL,2*4          ;Vergleiche AL mit 8
0002 B91200               8             MOV     CX,123H/16      ;CX = 12H
0005 80C403               9             ADD     AH,10-7         ;AH = AH + 3
0008 80E90F              10             SUB     CL,12+3         ;CL = CL - 15
000B B003                11             MOV     AL,11 MOD 4     ;AL = 3 (Restwert der Division 11 : 4)
000D 80FD02              12             CMP     CH,8 SHR 2      ;Schiebe 8 um zwei Bitstellen nach rechts
                         13                                     ;und vergleiche das Ergebnis (=2) mit CH
0010 B310                14             MOV     BL,2 SHL 3      ;Schiebe 2 um 3 Bitstellen nach links
                         15                                     ;BL = 16D = 10H
0012 B404                16             MOV     AH,HIGH 1234D   ;Wandle 1234D in hexadezimal um und
                         17                                     ;speichere High-Teil in AH
0014 B0D2                18             MOV     AL,LOW  1234D   ;Wandle 1234D in hexadezimal um und
                         19                                     ;speichere Low-Teil in AL
                         20
----                     21     CODE_PROG ENDS
                         22
                         23             END     START
```

Übersicht über Assembler-Operatoren: Fortsetzung

Logische Operatoren		Syntax		
AND	(Und-Verknüpfung)	Operand	AND	Operand
OR	(Oder-Verknüpfung)	Operand	OR	Operand
XOR	(Exklusiv Oder-Verknüpfung)	Operand	XOR	Operand
NOT	(Negation)		NOT	Operand

Hinweis: Operanden sind absolute Zahlen.

Das folgende Programm-Listing zeigt Anwendungen von **logischen** Assembler-Operatoren.

```
LOC  OBJ                LINE    SOURCE

                          1             NAME BEISPIEL_19
                          2
                          3     ASSUME CS:CODE_PROG
                          4
----                      5     CODE_PROG SEGMENT
                          6
0000 3CDB                 7     START:  CMP     AL,11011001B OR 10011011B       ;Vergleiche AL mit
                          8                                                     ;11011011B = DBH
0002 80C466               9             ADD     AH,10111011B XOR 11011101B      ;AH = AH + 01100110B (66H)
0005 80E981              10             SUB     CL,10110011B AND 11001101B      ;CL = CL - 10000001B (81H)
0008 B350                11             MOV     BL,NOT 10101111B                ;BL = 01010000B (5)
                         12
----                     13     CODE_PROG ENDS
                         14
                         15             END     START
```

4.4.2 Equates

In der 8086/80186/80286-Assemblersprache gibt es die Möglichkeit für Ausdrücke symbolische Namen zu vergeben. Ein Ausdruck kann eine Konstante sein, die in einem Assembler-Quellprogramm z.B. als immediate Operand mehrmals verwendet wird ①.

Beispiel:

```
CODE_PROG SEGMENT
   START: -
          -
          TEST AL, 20H
          -
          -
          TEST AL, 20H      ①
          -
          -
          TEST AL, 20H
          -
          -
CODE_PROG ENDS
          END START
```

Muß der immediate Operand geändert werden, bedeutet dies, daß der Anwenderprogrammierer die notwendige Änderung mehrmals durchführen muß. Dies ist zeitaufwendig und birgt die Gefahr in sich, daß notwendige Änderungen übersehen werden.

Mit Hilfe der EQU-Direktive (Equate) ist es möglich für den immediate Operanden ein Symbol zu vereinbaren. Sollte nun eine Änderung notwendig sein, ist lediglich die Wertzuweisung in der EQU-Direktive zu ändern. Statt also den immediate Operanden 20H zu benützen, ist es vorteilhafter, ein dafür vereinbartes Symbol zu verwenden ②.

Beispiel:

```
CODE_PROG SEGMENT
   START: -
          -
          TEST AL, MASKE
          -
          -
          TEST AL, MASKE      ②
          -
          -
          TEST AL, MASKE
          -
          -
CODE_PROG ENDS
          END START
```

Das folgende Statement zeigt die Sytax der EQU-Direktive.

Syntaxdiagramm der EQU-Direktive:

Symbolischer Name ist der metasprachliche Ausdruck für einen Namen (Identifizierer), den der Anwenderprogrammierer wählt.

EQU ist ein Assembler-Wortsymbol und muß angegeben werden.

Im Feld **Ausdruck** kann eine Konstante, eine Adresse (Label oder Variable), ein Register oder eine Befehlsmnemonik spezifiziert werden.

Beispiel:

```
LOC  OBJ                LINE     SOURCE

                          1              NAME    BEISPIEL_20
                          2
 0020      ②            3     MASKE EQU 20H
                          4
                          5     ASSUME CS:CODE_PROG
                          6
----                      7     CODE_PROG SEGMENT
                          8
0000                      9     START:  ;------
                         10             ;------
0000 A820                11             TEST    AL,MASKE            ;Undiere AL mit 20H
                         12             ;------
                         13             ;------
                         14             ;------
0002 A820  ①             15             TEST    AL,MASKE            ;Undiere AL mit 20H
                         16             ;------
                         17             ;------
0004 A820                18             TEST    AL,MASKE            ;Undiere AL mit 20H
                         19             ;------
                         20
----                     21     CODE_PROG ENDS
                         22
                         23             END     START
```

Im OBJ-Feld des Programmlistings ist ersichtlich, daß der Assembler für das Symbol MASKE den konstanten Wert 20H als immediate Operanden einsetzt ①.

Vereinbart wurde diese Wertzuweisung mit Hilfe der EQU-Direktive `MASKE EQU 20H` ②. Sollte für MASKE ein neuer Wert erforderlich sein, ist die Änderung nur einmal im Zuweisungs-Statement (EQU-Direktive) notwendig.

4.4.2.1 Vorwärts-Referenzen

In Assembler ist es möglich in einem Statement ein Symbol zu benützen, das erst später mit Hilfe eines anderen Statements definiert wird. Im folgenden Programmlisting wird eine Anwendung der Vorwärtsreferenz gezeigt.

```
LOC  OBJ                  LINE    SOURCE

                            1             NAME    BEISPIEL_21
                            2
 0000                       3     ZAHL EQU BYTE_VAR
                            4
                            5     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                            6
----                        7     DATEN_PROG SEGMENT
                            8
0000 ??   ①                 9     BYTE_VAR        DB      ?
                           10
----                       11     DATEN_PROG ENDS
                           12
----                       13     CODE_PROG SEGMENT
                           14
0000 C606000055   ②        15     START:  MOV     ZAHL,55H
                           16
----                       17     CODE_PROG ENDS
                           18
                           19             END     START
```

Mit Hilfe der EQU-Direktive `ZAHL EQU BYTE_VAR` wird das neue Symbol ZAHL definiert, das als **anderer Name** für BYTE_VAR benützt werden kann.

Da BYTE_VAR an dieser Stelle noch undefiniert ist, betrachtet der Assembler zunächst ZAHL als ein Symbol ohne Bedeutung. BYTE_VAR wird erst später im DATEN_PROG-Segment als eine Variable vom Typ BYTE deklariert ①.

An dieser Stelle "erinnert" sich der Assembler an das noch undefinierte Symbol ZAHL. Er wird also "zurückgehen" und ZAHL als einen weiteren Namen für die Bytevariable BYTE_VAR definieren.

Der Befehl `MOV ZAHL, 55H` zeigt, daß nun statt BYTE_VAR der andere Name ZAHL verwendet werden kann ②.

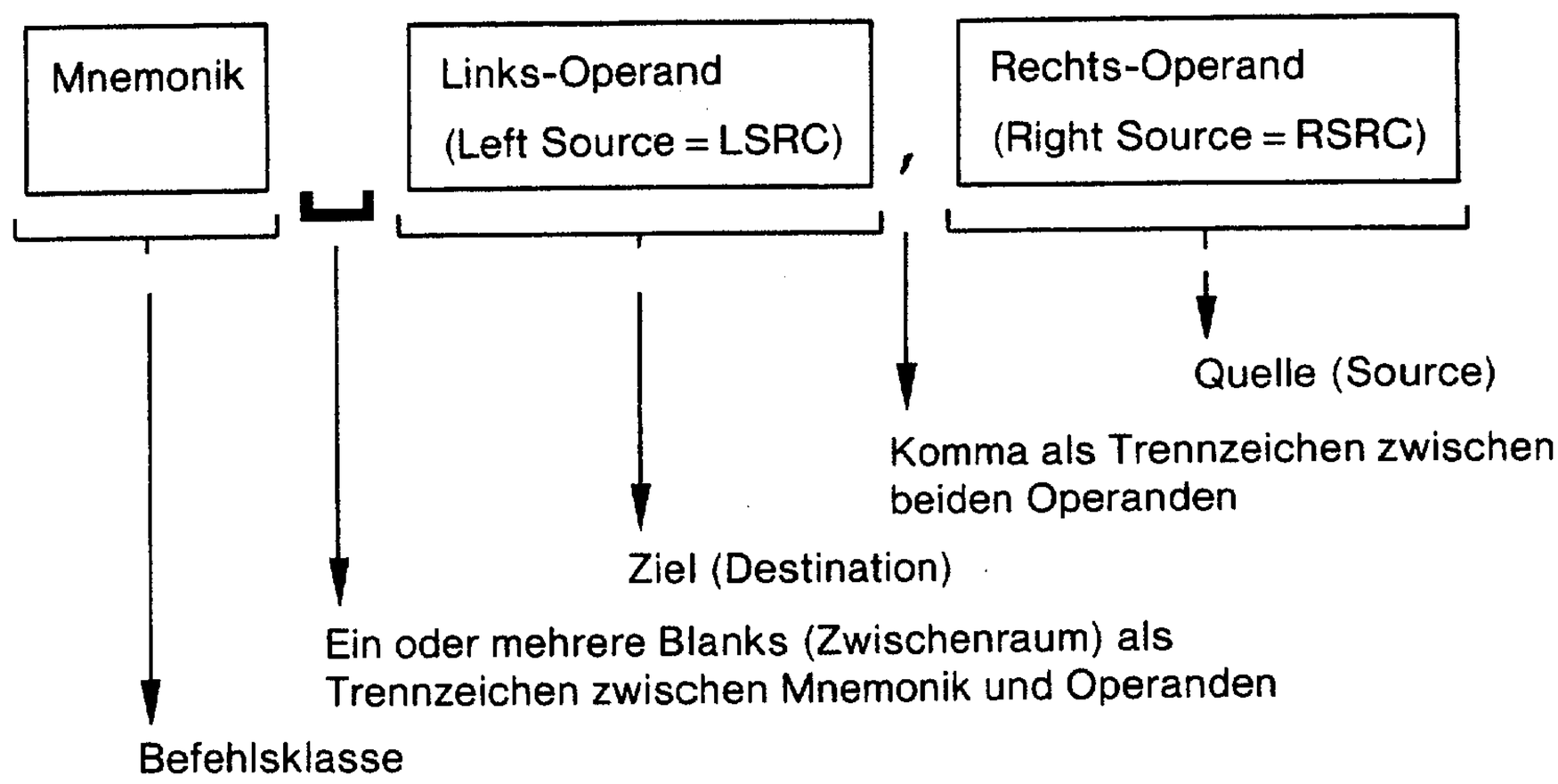
Das folgende Programmlisting zeigt weitere Beispiele für die Anwendung der EQU-Direktive.

```
LOC  OBJ                LINE    SOURCE

                         1              NAME BEISPIEL_22
                         2
  0000                   3     EIN_LAB     EQU START       ;EIN_LAB ist ein anderer Name fuer den Label
                         4                                 ;START, der erst spaeter im CODE_PROG Segment
                         5                                 ;definiert wird (Vorwaerts Referenz)
                         6
  REG                    7     ZAEHLER     EQU CX          ;ZAEHLER ist ein anderer Name fuer  Register CX
  REG                    8     POINTER     EQU BX          ;POINTER ist ein anderer Name fuer  Register BX
  0003[]                 9     VAR         EQU [BX+3]      ;VAR ist der Name des Registerausdrucks [BX+3]
                        10
  C MACRO               11     TRANSFER    EQU MOV         ;TRANSFER ist ein anderer Name fuer den
                        12                                 ;Befehlsnamen MOV
  C MACRO               13     INKREMENT   EQU INC         ;INKREMENT ist ein anderer Nmae fuer den
                        14                                 ;Befehlsname INC
                        15
  REAL                  16     PI          EQU 3.14159     ;PI ist der Name fuer die Floating-Point-
                        17                                 ;Zahl 3.14159
  0009                  18     TOTAL       EQU 9           ;TOTAL ist der Name fuer die Integer-Zahl 9
                        19
  0005                  20     A1          EQU 2+3         ;A1 ist der Name des ausgewerteten Ausdrucks
                        21                                 ;2+3
  0004                  22     A2          EQU A1 AND 4    ;A2 ist der Name des ausgewerteten Ausdrucks
                        23                                 ;A1 AND 4
                        24
                        25     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                        26
----                    27     DATEN_PROG SEGMENT
                        28
0000 D00F4940           29     VAR_1   DD      PI          ;Einfache Genauigkeit: 3.14159 wird in 32 Bits
                        30                                 ;dargestellt
0004 6E861BF0F92109     31     VAR_2   DQ      PI          ;Doppelte Genauigkeit: 3.14159 wird in 64 Bits
     40
                        32                                 ;dargestellt
000C 00                 33     FELD    DB      0,1,2,?
000D 01
000E 02
000F ??
                        34
----                    35     DATEN_PROG ENDS
                        36
----                    37     CODE_PROG SEGMENT
                        38
0000 B80000             39     START:  MOV     AX,0
                        40     ;
                        41     ;
0003 EBFB               42             JMP     EIN_LAB             ;Als Sprungziel wird der andere Name
                        43                                         ;fuer START,naemlich EIN_LAB benuetzt
0005 B90A00             44             MOV     ZAEHLER,10          ;CX = 10
0008 BB0C00             45             MOV     POINTER,OFFSET FELD ;BX = OFFSET von FELD
000B C6470306           46             MOV     BYTE PTR VAR,6      ;[BX+3] zeigt zum dritten Element in
                        47                                         ;FELD. Es wird mit 6 initialisiert
                        48                                         ;Siehe Hinweis
000F 8BC3               49             TRANSFER    AX,BX           ;AX = BX
0011 40                 50             INKREMENT   AX              ;AX = AX + 1
0012 83EA04             51             SUB     DX,A2               ;DX = DX - (2 + 3) AND 4
                        52
----                    53     CODE_PROG ENDS
                        54
                        55             END     EIN_LAB             ;Das Programm beginnt bei START
```

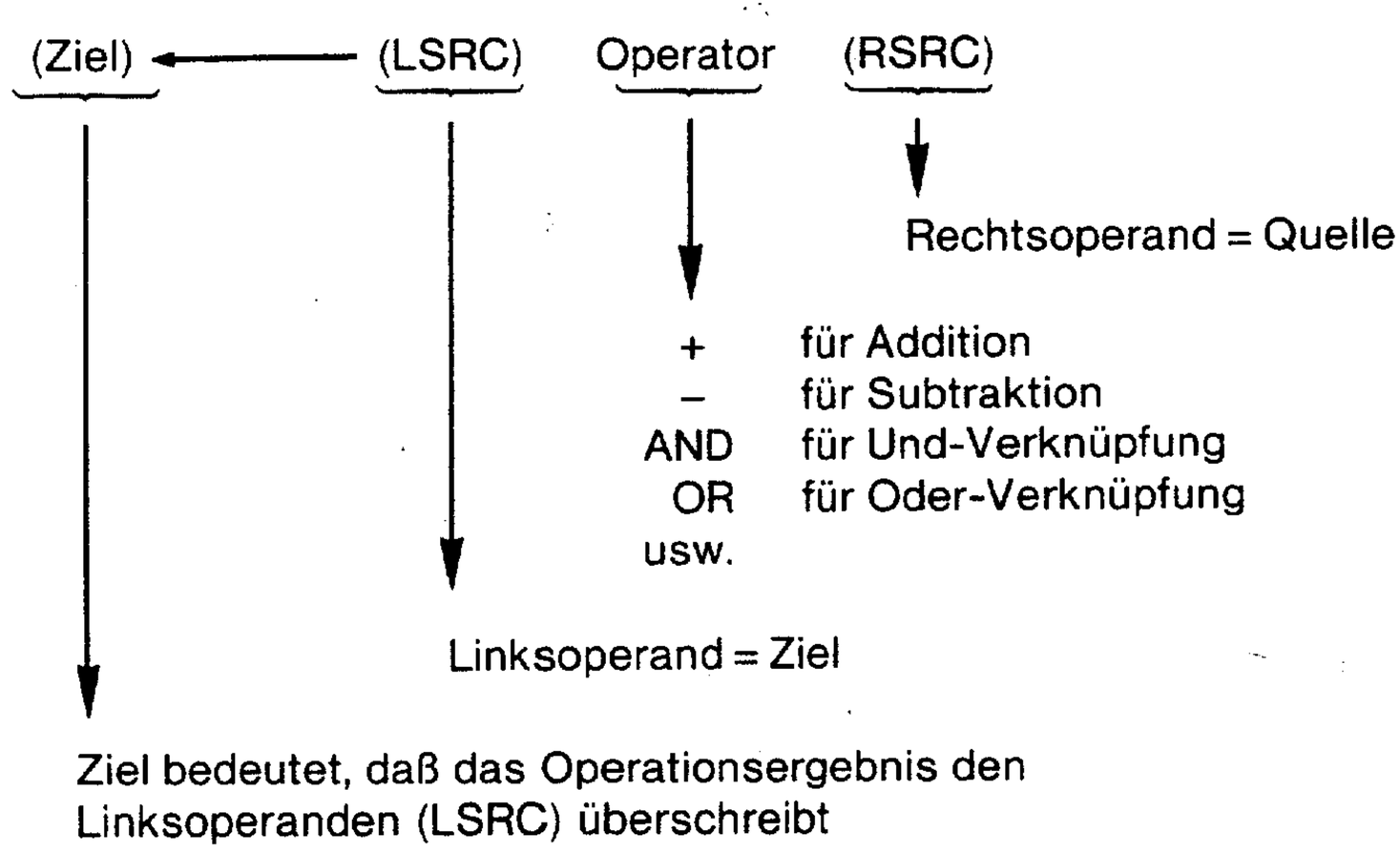
Hinweis: Beim Befehl `MOV BYTE PTR VAR, 6` wurde der **Typoperator** BYTE PTR verwendet. Er gibt an, daß VAR bzw. [BX + 3] auf eine Variable vom Typ BYTE zeigt. Die Typoperatoren werden im Kapitel "Typ-Overrides" näher behandelt.

4.5 Operandentyp und Generierung von Opcodes

Die 8086/80186/80286-Assemblersprache benützt für jede Befehlsklasse eine einheitliche Befehlsmnemonik. Bearbeitet ein Befehl zwei Operanden, gelten folgende Regeln:

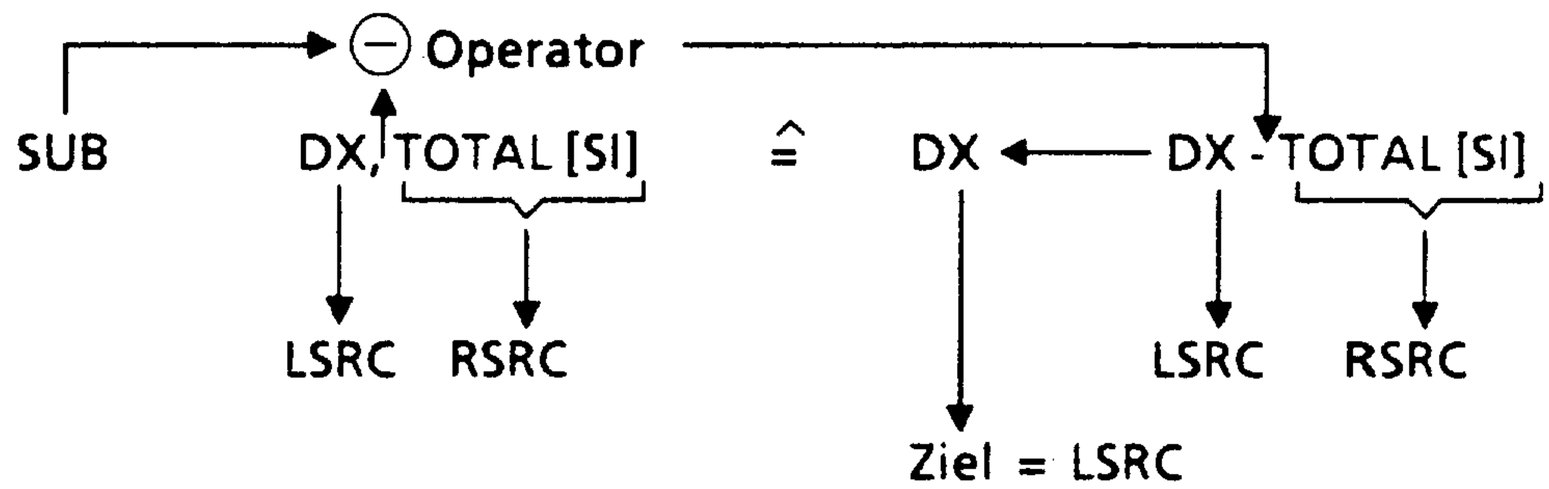


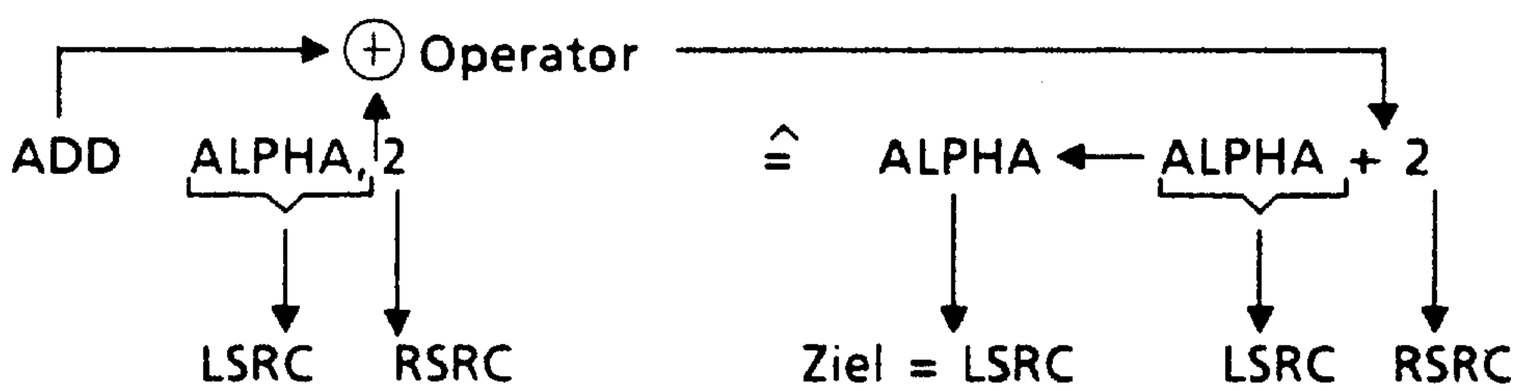
Der Operand rechts vom Komma wird als Quelle (Source), der Operand links vom Komma als Ziel (Destination) betrachtet ①. Danach werden zwei Operanden in folgender Weise verknüpft:

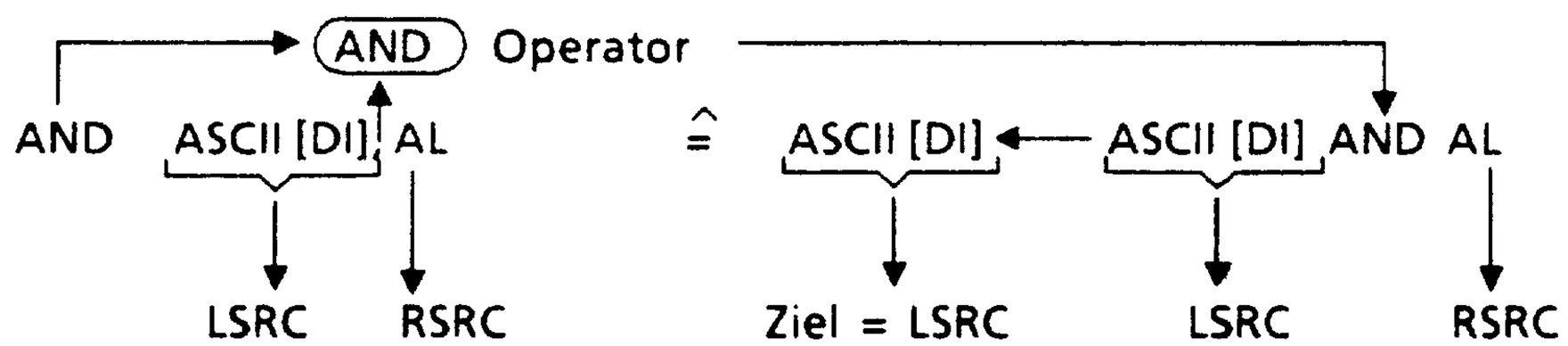


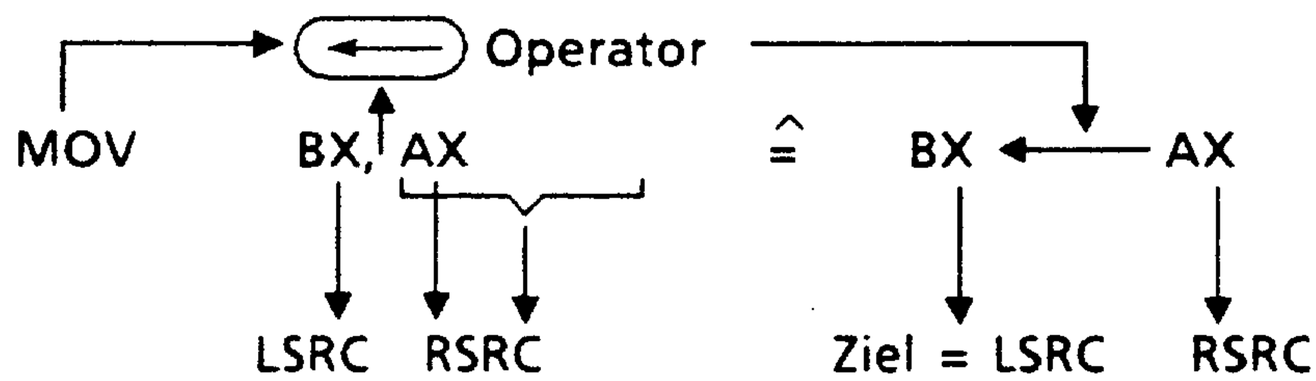
Die angegebenen Regeln bedeuten für den Anwenderprogrammierer, daß er sich das Trennungszeichen durch einen entsprechenden Operator **ersetzt** denken kann.

Beispiele:









Im letzten Beispiel erfolgt keine Verknüpfung, sondern lediglich eine Zuweisung des Rechtsoperanden AX an den Linksoperanden BX.

Obwohl für jede Befehlsklasse eine einheitliche Befehlsmnemonik existiert, setzt der Assembler für die verschiedenen Operationen innerhalb der Befehlsklasse unterschiedliche Opcodes ein. Um einen bestimmten Opcode zu generieren, benützt der Assembler die **Typinformation** über den im Befehl verwendeten Operanden.

Die MOV-Mnemonik wird für folgende Operationen verwendet:

- MOV Byte- oder Word-Register in Byte- oder Word-Register
- Lade Byte- oder Word-Register vom Speicher
- Lade Byte- oder Word-Register mit einer Konstanten
- MOV Byte- oder Word-Register in den Speicher
- MOV Byte- oder Word-Konstante in den Speicher

Für den Anwenderprogrammierer hat dies den Vorteil, daß er mnemotechnisch **nicht** zwischen "`MOV`" und "`Laden`", "`MOV Immediate`" und "`MOV Speicher`", "`MOV Byte`" und "`MOV Word`" usw. unterscheiden muß.

So wird der Assembler aus dem Befehl `MOV VAR_1, 56` die Operation "MOV Byte-Konstante in den Speicher" erkennen, wenn VAR_1 vom Typ BYTE ist und dafür den Opcode `C6H` einsetzen. Sollte VAR_1 vom Typ WORD sein, wird der Assembler die Operation "MOV Word-Konstante in den Speicher" erkennen und dafür den Opcode `C7H` einsetzen.

Die Typinformation über einen im Befehl verwendeten Operanden wird auch benützt, um Programmfehler zu verhindern. D.h. der Assembler wird eine Fehlermeldung ausgeben, wenn z. B. versucht wird einem Byte-Register ein Word-Register zuzuweisen oder wenn in einem MOV-Befehl ein Label als Operand verwendet wird usw.

Die folgenden Befehlsbeispiele sollen zeigen, wie der Assembler die unterschiedlichen Operandentypen benützt, um die entsprechenden Opcodes zu generieren und wie der Assembler Programmierfehler entdecken kann.

Für jedes Befehlsbeispiel liefert der MOV-Befehl einen anderen 8086/80186/80286-Opcode oder einen Fehler.

```
LOC  OBJ                  LINE     SOURCE

                             1             NAME    BEISPIEL_23
                             2
                             3     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                             4
----                         5     DATEN_PROG SEGMENT
                             6
0000 ??                      7     BVAR    DB      ?       ;Byte Variable
0001 ????                    8     WVAR    DW      ?       ;Word Variable
                             9
----                        10     DATEN_PROG ENDS
                            11
----                        12     CODE_PROG SEGMENT
                            13
0000 9090909090             14     START:  MOV     CL,2014 ;FEHLER: Die Konstante ist fuer ein Byte Register
*** ERROR #2 IN 14, OPERANDS DO NOT MATCH THIS INSTRUCTION
                            15                             ;        zu lang.
0005 8BC3                   16             MOV     AX,BX   ;(8BH) Move Word Register in Word Register
0007 BA0800                 17             MOV     DX,8    ;(BAH) Move Konstante in Word Register
000A 9090909090             18             MOV     START,4 ;FEHLER: Im MOV Befehl darf kein Label verwendet
*** ERROR #2 IN 18, OPERANDS DO NOT MATCH THIS INSTRUCTION
                            19                             ;        werden.
000F 891E0100               20             MOV     WVAR,BX ;(89H) Move Word Register in Word Variable
0013 9090909090             21             MOV     AL,WVAR ;FEHLER: Typkonflikt (BYTE,WORD)
*** ERROR #2 IN 21, OPERANDS DO NOT MATCH THIS INSTRUCTION
0018 C60600000A             22             MOV     BVAR,10 ;(C6H) Move Konstante in Byte Variable
001D 8ED8                   23             MOV     DS,AX   ;(8EH) Move Word Register in Segment Register
001F 9090909090             24             MOV     DX,CL   ;FEHLER: Typkonflikt (WORD,BYTE)
*** ERROR #2 IN 24, OPERANDS DO NOT MATCH THIS INSTRUCTION
                            25
----                        26     CODE_PROG ENDS
                            27
                            28             END START
```

4.6 Attribut-Overrides

Wie bereits erklärt, haben Variable drei Attribute:

- Einen Typ
- Ein Segment
- Einen Offset innerhalb des Segments.

Der Assembler erlaubt nun, das Typ- oder Segment-Attribut einer Variablen vorübergehend zu ändern. Zu diesem Zweck stellt der Assembler spezielle **Attribut-Override-Operatoren** zur Verfügung.

In den folgenden Kapiteln werden sowohl der Segment-Override, als auch der Typ-Override diskutiert.

4.6.1 Segment Overrides

```
LOC  OBJ                  LINE     SOURCE

                            1            NAME    BEISPIEL_24
                            2
                            3     ASSUME CS:CODE_PROG,DS:HAUPT_DATEN
                            4
                            5     ;*************************************
----                        6     HAUPT_DATEN SEGMENT                ;*
0000 ??                     7     VAR     DB      ?                  ;*
----                        8     HAUPT_DATEN ENDS                   ;*
                            9     ;*************************************
                           10
                           11     ;*************************************
----                       12     ZUSATZ_DATEN_1 SEGMENT             ;*
0000 ??                    13     VAR_1   DB      ?                  ;*
----                       14     ZUSATZ_DATEN_1 ENDS                ;*
                           15     ;*************************************
                           16
                           17     ;*************************************
----                       18     ZUSATZ_DATEN_2 SEGMENT             ;*
0000 ??                    19     VAR_2   DB      ?                  ;*
----                       20     ZUSATZ_DATEN_2 ENDS                ;*
                           21     ;*************************************
                           22
                           23     ;*************************************
----                       24     ZUSATZ_DATEN_3 SEGMENT             ;*
0000 ??                    25     VAR_3   DB      ?                  ;*
----                       26     ZUSATZ_DATEN_3 ENDS                ;*
                           27     ;*************************************
                           28
----                       29     CODE_PROG SEGMENT
                           30
0000 B8----          R     31     START:  MOV     AX,HAUPT_DATEN     ;DS-Register initialisieren
0003 8ED8                  32             MOV     DS,AX
                           33
----                       34     CODE_PROG ENDS
                           35
                           36             END     START
```

Es sei vorausgesetzt, daß in einem Assembler-Quellprogramm verschiedene Daten-Regionen existieren. Im Beispiel sind dies die logischen Segmente HAUPT_DATEN, ZUSATZ_DATEN_1, ZUSATZ_DATEN_2 und ZUSATZ_DATEN_3 ①.

Das HAUPT_DATEN-Segment soll allgemein benützbare Variablen enthalten. Das bedeutet, daß DS die Basisadresse des HAUPT_DATEN-Segments enthält.

Der Initialisierungscode des Programms wird aus diesem Grund DS mit HAUPT_DATEN laden ②. Gleichzeitig wird `ASSUME DS: HAUPT_DATEN` für die Fortdauer des Programms wirksam bleiben ③.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_25
                            2
                            3      ASSUME CS:CODE_PROG,DS:HAUPT_DATEN
                            4
                            5      ;*************************************
----                        6      HAUPT_DATEN SEGMENT                ;*
0000 ??                     7      VAR     DB      ?                  ;*
----                        8      HAUPT_DATEN ENDS                   ;*
                            9      ;*************************************
                           10
                           11      ;*************************************
----                       12      ZUSATZ_DATEN_1 SEGMENT             ;*
0000 ??                    13      VAR_1   DB      ?                  ;*
----                       14      ZUSATZ_DATEN_1 ENDS                ;*
                           15      ;*************************************
                           16
                           17      ;*************************************
----                       18      ZUSATZ_DATEN_2 SEGMENT             ;*
0000 ??                    19      VAR_2   DB      ?                  ;*
----                       20      ZUSATZ_DATEN_2 ENDS                ;*
                           21      ;*************************************
                           22
                           23      ;*************************************
----                       24      ZUSATZ_DATEN_3 SEGMENT             ;*
0000 ??                    25      VAR_3   DB      ?                  ;*
----                       26      ZUSATZ_DATEN_3 ENDS                ;*
                           27      ;*************************************
                           28
----                       29      CODE_PROG SEGMENT
                           30
0000 B8----          R     31      START:  MOV     AX,HAUPT_DATEN     ;DS-Register initialisieren
0003 8ED8                  32              MOV     DS,AX
0005 C606000035            33              MOV     VAR,35H            ;VAR im HAUPT_DATEN Segment mit 35H
                           34                                         ;laden
                           35      ASSUME ES:ZUSATZ_DATEN_1
000A B8----          R     36              MOV     AX,ZUSATZ_DATEN_1  ;ES-Register initialisieren
000D 8EC0                  37              MOV     ES,AX
000F 26C606000076          38              MOV     VAR_1,76H          ;VAR_1 im ZUSATZ_DATEN_1 Segment mit
                           39                                         ;76H laden
                           40      ASSUME ES:ZUSATZ_DATEN_2                                          ①
0015 B8----          R     41              MOV     AX,ZUSATZ_DATEN_2  ;ES-Register initialisieren
0018 8EC0                  42              MOV     ES,AX
001A 26C606000086          43              MOV     VAR_2,86H          ;VAR_2 im ZUSATZ_DATEN_2 Segment mit
                           44                                         ;86H laden
                           45      ASSUME DS:ZUSATZ_DATEN_3
0020 B8----          R     46              MOV     AX,ZUSATZ_DATEN_3  ;DS-Register initialisieren
0023 8ED8                  47              MOV     DS,AX
0025 C60600003E            48              MOV     VAR_3,3EH          ;VAR_3 im ZUSATZ_DATEN_3 Segment mit
                           49                                         ;3EH laden
                           50
----                       51      CODE_PROG ENDS
                           52
                           53              END     START
```

Soll auch auf die Datenregionen ZUSATZ_DATEN_1, ZUSATZ_DATEN_2 und ZUSATZ_DATEN_3 gelegentlich zugegriffen werden, ist dies möglich, wenn ES oder DS auf diese Extra-Daten-Segmente zeigen.

Jedesmal, wenn der Zugriff von einem Extra-Daten-Segment auf ein anderes wechselt, muß das ES- oder DS-Register mit Hilfe der Segment-Ladebefehle umgeladen werden. Zusätzlich bekommt der Assembler durch ASSUME-Anweisungen mitgeteilt, welche Segment-Register beim Zugriff auf Variable zu benützen sind ①.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_26
                            2
                            3      ASSUME CS:CODE_PROG,DS:HAUPT_DATEN
                            4
                            5      ;*********************************
----                        6      HAUPT_DATEN SEGMENT              ;*
0000 ??                     7      VAR     DB      ?                ;*
----                        8      HAUPT_DATEN ENDS                 ;*
                            9      ;*********************************
                           10
                           11      ;*********************************
----                       12      ZUSATZ_DATEN_1 SEGMENT           ;*
0000 ??                    13      VAR_1   DB      ?                ;*
----                       14      ZUSATZ_DATEN_1 ENDS              ;*
                           15      ;*********************************
                           16
                           17      ;*********************************
----                       18      ZUSATZ_DATEN_2 SEGMENT           ;*
0000 ??                    19      VAR_2   DB      ?                ;*
----                       20      ZUSATZ_DATEN_2 ENDS              ;*
                           21      ;*********************************
                           22
                           23      ;*********************************
----                       24      ZUSATZ_DATEN_3 SEGMENT           ;*
0000 ??                    25      VAR_3   DB      ?                ;*
----                       26      ZUSATZ_DATEN_3 ENDS              ;*
                           27      ;*********************************
                           28
----                       29      CODE_PROG SEGMENT
                           30
0000 B8----          R     31      START:  MOV     AX,HAUPT_DATEN      ;DS-Register initialisieren
0003 8ED8                  32              MOV     DS,AX
0005 C606000035            33              MOV     VAR,35H             ;VAR im HAUPT_DATEN Segment mit 35H
                           34                                          ;laden
000A B8----          R     35              MOV     AX,ZUSATZ_DATEN_1   ;ES-Register initialisieren
000D 8EC0                  36              MOV     ES,AX
000F 26C606000076          37              MOV     ES:VAR_1,76H        ;VAR_1 im ZUSATZ_DATEN_1 Segment mit
                           38                                          ;76H laden                          ①
0015 B8----          R     39              MOV     AX,ZUSATZ_DATEN_2   ;ES-Register neu initialisieren
0018 8EC0                  40              MOV     ES,AX
001A 26C606000086          41              MOV     ES:VAR_2,86H        ;VAR_2 im ZUSATZ_DATEN_2 Segment mit
                           42                                          ;86H laden
0020 B8----          R     43              MOV     AX,ZUSATZ_DATEN_3   ;ES-Register neu initialisieren
0023 8EC0                  44              MOV     ES,AX
0025 26C60600003E          45              MOV     ES:VAR_3,3EH        ;VAR_3 im ZUSATZ_DATEN_3 Segment mit
                           46                                          ;3EH laden
                           47
----                       48      CODE_PROG ENDS
                           49
                           50              END     START
```

Es gibt in Assembler eine bequeme Alternative, die Wirkung der augenblicklich gültigen ASSUME-Anweisung aufzuheben. Soll z.B. auf die Variablen VAR_1, VAR_2 und VAR_3 in den Extra-Daten-Segmenten ZUSATZ_DATEN_1, ZUSATZ_DATEN_2 und ZUSATZ_DATEN_3 relativ zu ES gelegentlich zugegriffen werden, kann als Alternative zur ASSUME-Anweisung der sogenannte **Segment-Override**-Operator verwendet werden.

Dieser Operator ist nichts weiter als der Name eines Segmentregisters, gefolgt von einem Doppelpunkt (:). Er muß vor dem Namen einer zu adressierenden Variablen angegeben werden.

Der Operator teilt dem Assembler mit, welches Segment-Register er benützen muß, um eine bestimmte Variable zu adressieren. Vorausgesetzt ist dabei, daß das alternative Segment-Register, z.B. ES, die Basisadresse des Segments enthält, in der sich die Variable befindet ①.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_27
                            2
                            3      ASSUME CS:CODE_PROG,DS:HAUPT_DATEN
                            4
                            5      ;*************************************
----                        6      HAUPT_DATEN SEGMENT                ;*
0000 ??                     7      VAR     DB      ?                  ;*
----                        8      HAUPT_DATEN ENDS                   ;*
                            9      ;*************************************
                           10
                           11      ;*************************************
----                       12      ZUSATZ_DATEN_1 SEGMENT             ;*
0000 (10                   13      VAR_1   DB      10 DUP (?)         ;*
     ??
     )
----                       14      ZUSATZ_DATEN_1 ENDS                ;*
                           15      ;*************************************
                           16
----                       17      CODE_PROG SEGMENT
                           18
0000 B8----          R     19      START:  MOV     AX,HAUPT_DATEN          ;DS-Register initialisieren
0003 8ED8                  20              MOV     DS,AX
0005 C606000035            21              MOV     VAR,35H                 ;VAR im HAUPT_DATEN Segment mit 35H
                           22                                              ;laden
000A B8----          R     23              MOV     AX,ZUSATZ_DATEN_1       ;ES-Register initialisieren
000D 8EC0                  24              MOV     ES,AX
000F BB0400                25              MOV     BX,4                    ;BX = Offsetwert 4
0012 26C6079D              26              MOV     BYTE PTR ES:[BX],9DH    ;[BX] zeigt zum vierten BYTE
                           27                                              ;im ZUSATZ_DATEN_1 Segment.
                           28                                              ;Es wird mit 9DH geladen.
                           29
----                       30      CODE_PROG ENDS
                           31
                           32              END     START
```

①

Der Segment-Override-Operator kann auch für sogenannte anonyme Referenzen verwendet werden. Dabei werden Speicheradressen ohne Angabe von Variablennamen spezifiziert.

Zum Beispiel benützt die indirekte Adresse [BX] im Normalfall (default) DS als Basisregister. Diese Zusammengehörigkeit kann aufgehoben werden, wenn dem Adreßzeiger [BX] z.B. der Segment-Override-Operator ES: vorangestellt wird.

Beispiel: `MOV CX, ES: [BX]`

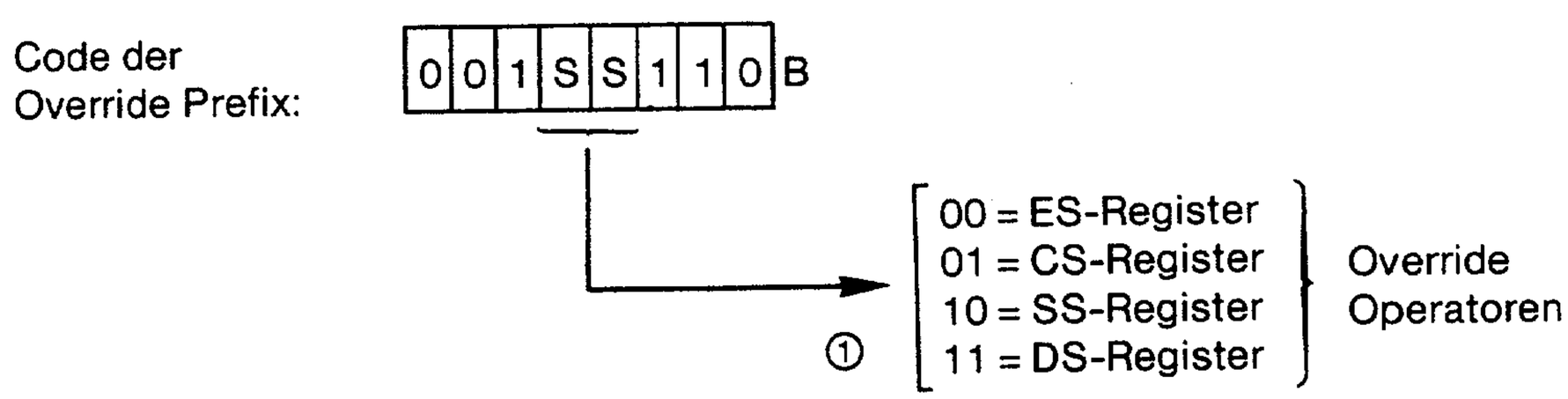
Die indirekte Adresse [BX] (anonyme Referenz) benützt nun ES als Basisregister.

Im Programmlisting ist für eine anonyme Referenz mit Segment-Override-Operator der Befehl `MOV BYTE PTR ES: [BX], 9DH` angegeben ①. Dabei ist BYTE PTR ein Typ-Operator, der angibt, daß ES: [BX] auf eine Variable vom Typ BYTE zeigt (siehe Kapitel: Typ-Overrides).

Wird ein Segment-Override-Operator in Verbindung mit einer Variablen benützt, dient er als **Kurzschreibweise** für eine ASSUME-Anweisung. Das Programm BEISPIEL_26 enthält die Anweisung `ASSUME DS: DATEN_PROG`. Sie ist wirksam für die Fortdauer des Programms.

Die Segment Overrides in den Befehlen `MOV ES: VAR_1, 76H`, `MOV ES: VAR_2, 86H` und `MOV ES: VAR_3, 3EH` besagen nun folgendes: Es ist für die Befehle ohne Bedeutung, daß `ASSUME DS: DATEN_PROG` für das gesamte Programm wirksam ist. Die Kurzschreibweise ES: hebt diese Wirkung für den jeweils auszuführenden Befehl auf. D.h. der Assembler benützt zur Adressierung der Variablen nicht DS, sondern ES.

Erkennt der Assembler einen Segment-Override-Operator in einem Befehl, kennzeichnet er den übersetzten Befehl mit einer sogenannten **Segment-Override-Prefix**. Dies ist ein Byte, das dem übersetzten Befehl vorangestellt wird. In diesem Byte ist der Code des Override-Operators enthalten ①.



Beispiel: `MOV ES:VAR_1, 76H` ≙ `26 C6 06 00 00 76H`

Segment Override Prefix
für ES: Operator

```
LOC  OBJ                    LINE     SOURCE

                              1              NAME    BEISPIEL_28
                              2
                              3     ASSUME CS:CODE_PROG,SS:STACK_PROG
                              4
----                          5     STACK_PROG SEGMENT
                              6
0000 (10                      7              DW      10 DUP (?)
     ????
     )
0014                          8     TOP_STACK LABEL WORD
                              9
----                         10     STACK_PROG ENDS
                             11
----                         12     CODE_PROG SEGMENT
                             13
0000 B8----          R       14     START:  MOV     AX,STACK_PROG          ;SS Register initialisieren
0003 8ED0                    15             MOV     SS,AX
0005 BD1400                  16             MOV     BP,OFFSET TOP_STACK    ;BP Register initialisieren
0008 8946F8                  17             MOV     SS:[BP-8],AX           ;ASM86 generiert keine
                             18                                            ;Override Prefix fuer SS:   ①
                             19
----                         20     CODE_PROG ENDS
                             21
                             22             END     START
```

Es gibt Fälle, bei denen die Angabe eines Segment-Override-Operators **nicht** garantiert, daß ein Segment-Override-Prefix-Byte generiert wird. Zur Erinnerung: Der Segment-Override-Operator teilt dem Assembler mit, welches Segment-Register er bei einer Speicheradressierung verwenden soll. Der Assembler darf aus dieser Information bestimmen, daß **kein** Segment-Override-Prefix-Byte für den Befehl gebraucht wird.

Im angegebenen Programmlisting ist als Beispiel der Befehl `MOV SS:[BP-8], AX` codiert ①. Dem Assembler wird darin mitgeteilt, einen Befehl zu generieren, der das Segment-Register SS mit (BP-8) kombiniert, um eine Speicheradresse zu bilden. Da SS und BP von Natur aus zusammengehören, ist **kein** Segment-Override-Prefix-Byte notwendig. Der Assembler wird daher auch keine SS-Prefix generieren ①.

4.6.1.1 Zusammenfassung

Die angegebene Tabelle zeigt, daß logische Basisadressen von verschiedenen Segment-Registern geliefert werden. Welche Segment-Register benützt werden, hängt vom Typ des Speicherzugriffs ab.

Der Assembler bezieht darüber seine Informationen aus der ASSUME-Anweisung und berücksichtigt bei anonymen Referenzen die Default-Segment-Register.

Der Segment-Override-Operator teilt dem Assembler explizit mit, welches Segment-Register er für einen bestimmten Befehl benützen soll. Alternativen zeigt die Tabelle.

Logische Basisadressen:

Typ des Speicherzugriffs	Default Segment Register	Alternative Segment Register	Offset
Instruction Fetch	CS	Keine	IP
Stack Operation	SS	Keine	SP, BP
Variable	DS	CS, ES, SS	Effektive Adresse
String Source	DS	CS, ES, SS	SI
String Destination	ES	Keine	DI
BP als Basisregister benützt	SS	CS, DS, ES	Effektive Adresse
BX als Basisregister benützt	DS	CS, ES, SS	Effektive Adresse

a=&b; → { mov ax, offset b
mov a, ax

c=*a; → { mov bx, a
mov al, [bx]
mov c, al

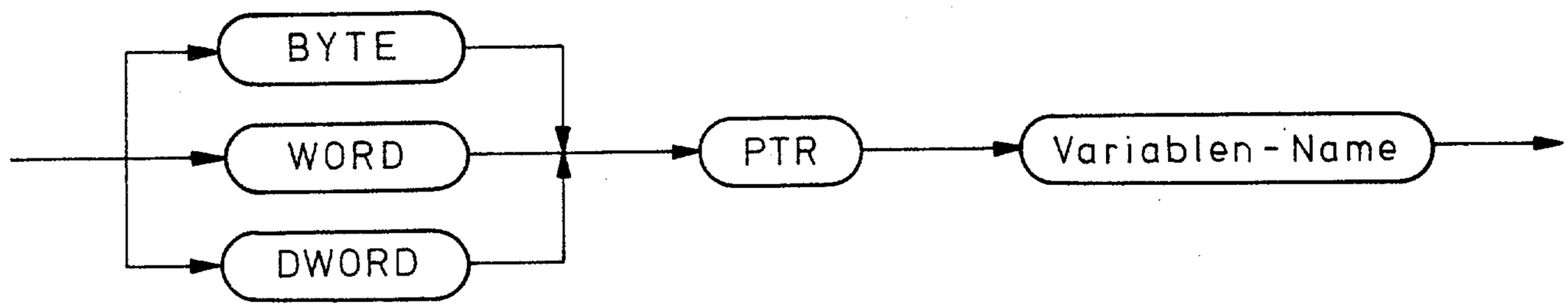
4.6.2 Typ-Overrides

Jede Variable ist durch einen **Typ** gekennzeichnet, der die Anzahl der Bytes für ein Symbol festlegt. Der Typ einer Variablen teilt dem Assembler mit, welche Art von Befehl er für die Variable generieren muß. Hat z.B. VAR den Typ WORD, dann spezifiziert `SUB VAR, 4` eine 16-Bit-Konstante (0004H) und subtrahiert sie von der Position, die durch VAR festgelegt ist.

Zusätzlich kann der Typ von VAR bei einer falschen Anwendung der WORD-Variablen benützt werden. Der Assembler wird nämlich eine Fehlermeldung ausgeben, wenn z.B. `MOV AL, VAR` codiert wird, da AL ein BYTE-Register, VAR dagegen eine WORD-Variable ist.

Möchte der Anwenderprogrammierer, daß für `SUB VAR, 3` ein BYTE-Befehl generiert wird, muß er zunächst klären, daß er z.B. zum ersten Byte der WORD-Variablen VAR zugreifen will. Dies ist möglich, wenn der sog. **Typ-Override-Operator** angewendet wird.

Syntax des Typ-Override-Operators:



Der Ausdruck **BYTE PTR Variablen-Name** besagt folgendes: Benütze das Segment und den Offset der Variablen mit dem Namen **Variablen-Name**, aber vom explizit angegebenen Typ BYTE.

Um den Typ-Konflikt beim Befehl `MOV VAR, AL` zu vermeiden, könnte nun wie folgt codiert werden: `MOV BYTE PTR VAR, AL`. In diesem Fall gibt BYTE PTR VAR an, daß VAR zum ersten Byte der WORD-Variablen VAR zeigt.

Möchte der Anwenderprogrammierer, daß für `SUB VAR, 4` ein BYTE-Befehl generiert wird, müßte er wie folgt codieren: `SUB BYTE PTR VAR, 4`. In diesem Fall wird vom ersten Byte der WORD-Variablen VAR die BYTE-Konstante 4 subtrahiert.

Programmierbeispiel:

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_29
                            2
                            3      ASSUME CS:CODE_PROG,DS:DATEN_PROG
                            4
----                        5      DATEN_PROG SEGMENT
                            6
0000 ????                   7      VAR     DW      ?
                            8
----                        9      DATEN_PROG ENDS
                           10
----                       11      CODE_PROG SEGMENT
                           12
0000 A20100                13      START:  MOV     BYTE PTR VAR+1,AL       ;Das zweite Byte der WORD-Variablen
                           14                                              ;VAR mit AL laden
0003 802E010004            15              SUB     BYTE PTR VAR+1,4        ;Vom zweiten Byte der WORD-Variablen
                           16                                              ;VAR  Byte-Konstante 4 subtrahieren
                           17
----                       18      CODE_PROG ENDS
                           19
                           20              END     START
```

Der Typ-Override-Operator wird auch benützt, um Zweideutigkeiten, die manchmal bei anonymen Referenzen auftreten, zu vermeiden. Dazu zwei Beispiele: `INC [BX + 5]` und `MOV [BP], 7`

Von den Befehlsklassen INC und MOV existieren sowohl die BYTE-, als auch die WORD-Versionen. In keinem der angegebenen Befehle wird dem Assembler mitgeteilt, ob er eine BYTE- oder eine WORD-Operation durchführen soll. Der Assembler wird daher wegen ungenügender Typ-Information eine Fehlermeldung ausgeben.

Mit Hilfe des Typ-Override-Operators kann diese Zweideutigkeit vermieden werden. Die Befehle `INC BYTE PTR [BX + 5]` und `MOV BYTE PTR [BP], 7` teilen dem Assembler mit, daß er die BYTE-Versionen der Befehle generieren muß. Die Befehle `INC WORD PTR [BX + 5]` und `MOV WORD PTR [BP], 7` teilen dem Assembler mit, daß er die WORD-Versionen der Befehle generieren muß.

Programmierbeispiel:

```
LOC  OBJ                    LINE     SOURCE

                             1              NAME    BEISPIEL_30
                             2
                             3      ASSUME CS:CODE_PROG
                             4
----                         5      DATEN_PROG SEGMENT
                             6
0000 (10                     7              DW      10 DUP (?)
     ????
     )
                             8
----                         9      DATEN_PROG ENDS
                            10
----                        11      STACK_PROG SEGMENT
                            12
0000 (10                    13              DW      10 DUP (?)
     ????
     )
0014                        14      STACK_TOP LABEL WORD
                            15
----                        16      STACK_PROG ENDS
                            17
----                        18      CODE_PROG SEGMENT
                            19
0000 B8----          R      20      START:  MOV     AX,DATEN_PROG          ;DS Register initialisieren
0003 8ED8                   21              MOV     DS,AX
0005 B8----          R      22              MOV     AX,STACK_PROG          ;SS Register initialisieren
0008 8ED0                   23              MOV     SS,AX
000A BC1400                 24              MOV     SP,OFFSET STACK_TOP    ;SP Register initialisieren
000D FE4705                 25              INC     BYTE PTR [BX+5]        ;Byte Version
0010 FF4705                 26              INC     WORD PTR [BX+5]        ;Word Version
0013 C6460007               27              MOV     BYTE PTR [BP],7        ;Byte Version
0017 C746000700             28              MOV     WORD PTR [BP],7        ;Word Version
                            29
----                        30      CODE_PROG ENDS
                            31
                            32              END     START
```

Nicht alle anonymen Referenzen führen zwangsläufig zu Zweideutigkeiten. Existieren in einem Befehl zwei Operanden, braucht nur der Typ **eines** Operanden angesehen werden.

So wird zum Beispiel der Befehl `MOV [BX], AX` eine WORD-Operation durchführen. Dies ist zurückzuführen auf die Tatsache, daß AX ein WORD-Register ist. In diesem Fall ist es bedeutungslos, daß dem Zeiger [BX] keine Typ-Information mitgegeben wird.

4.6.2.1 Zusammenfassung

Der Assembler benützt den Typ einer Variablen oder eines Register-Operanden, um zu bestimmen, daß ein Befehl mit einem Byte, einem Word oder einem Double Word operiert.

Möchte der Anwenderprogrammierer auf eine Variable zugreifen und dabei einen anderen Typ als den definierten benützen, kann er das gegebene Typ Attribut explizit mit dem **PTR-Operator** überschreiben.

Bei anonymen Referenzen mit dem Speicher unterscheiden einige Befehle nicht zwischen BYTE- und WORD-Operationen. In diesem Fall wird der Typ-Operator gebraucht, um explizit eine BYTE- oder WORD-Operation zu kennzeichnen.

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_31
                               2
                               3     ASSUME CS:CODE_PROG,DS:HAUPT_DATEN
                               4
----                           5     HAUPT_DATEN SEGMENT
                               6
0000 (4                        7     PACKED_DEZIMAL          DB      4 DUP(?)
     ??
     )
                               8
----                           9     HAUPT_DATEN ENDS
                              10
----                          11     EXTRA_DATEN SEGMENT
                              12
0000 09                       13     UNPACKED_DEZIMAL        DB      9,8,7,6,5,4,3,2
0001 08
0002 07
0003 06
0004 05
0005 04
0006 03
0007 02
                              14
----                          15     EXTRA_DATEN ENDS
                              16
----                          17     CODE_PROG SEGMENT
```

```
LOC  OBJ                  LINE     SOURCE

                           18
0000 B8----          R     19     START:  MOV     AX,HAUPT_DATEN              ;DS Register initialisieren
0003 8ED8                  20             MOV     DS,AX
0005 B8----          R     21             MOV     AX,EXTRA_DATEN              ;ES Register initialisieren
0008 8EC0                  22             MOV     ES,AX
000A BB0000                23             MOV     BX,OFFSET PACKED_DEZIMAL    ;[BX] zeigt zum Anfang von
                           24                                                 ;PACKED_DEZIMAL
000D BE0000                25             MOV     SI,0                        ;[SI] entspricht dem Index
                           26                                                 ;des ersten Elements im
                           27                                                 ;Feld UNPACKED_DEZIMAL
0010 8BFE                  28             MOV     DI,SI                       ;[DI] entspricht dem Index
                           29                                                 ;des ersten Elements im
                           30                                                 ;Feld PACKED_DEZIMAL
0012 B90400                31             MOV     CX,4                        ;CX ist der Wierholungszaehler
0015 268B04                32     GEPACK: MOV     AX,WORD PTR ES:UNPACKED_DEZIMAL[SI]                    ①
                           33                                                 ;Lade zwei ungepackte Dezimal-
                           34                                                 ;zahlen in AX
0018 D0E4                  35             SHL     AH,1                        ;Hoeherwertiges Byte mit 16
001A D0E4                  36             SHL     AH,1                        ;multiplizieren
001C D0E4                  37             SHL     AH,1
001E D0E4                  38             SHL     AH,1
0020 02C4                  39             ADD     AL,AH                       ;Packe zwei Dezimalzahlen in
                           40                                                 ;ein Byte
0022 8801                  41             MOV     [BX][DI],AL                 ;Speichere gepackte Dezimalzahl
0024 83C602                42             ADD     SI,2                        ;Bilde Index zum naechsten
                           43                                                 ;WORD (zwei ungepackte Dezimal-
                           44                                                 ;zahlen)
0027 47                    45             INC     DI                          ;Bilde Index zur naechsten
                           46                                                 ;gepackten Dezimalzahl
0028 E2EB                  47             LOOP    GEPACK                      ;Wiederhole bis der Zaehler
                           48                                                 ;CX = 0 ist
                           49
----                       50     CODE_PROG ENDS
                           51
                           52             END     START
```

Das angegebene Programmbeispiel entspricht in seiner Funktion dem BEISPIEL_16. Jeweils zwei ungepackte Dezimalzahlen, die im Feld UNPACKED_DEZIMAL gespeichert sind, werden in ein Byte gepackt und im Feld PACKED_DEZIMAL hinterlegt.

Die ungepackten Dezimalzahlen sind im EXTRA_DATEN-Segment definiert, während für die gepackten Dezimalzahlen Speicherplatz im HAUPT_DATEN-Segment reserviert ist.

Im Befehl `MOV AX, WORD PTR ES: UNPACKED_DEZIMAL [SI]` teilt der Segment-Override-Operator ES: dem Assembler mit, daß er bei diesem Befehl als Basiskomponente der UNPACKED_DEZIMAL-Adresse ES verwenden soll (Hinweis: Es ist `ASSUME ES: NOTHING` wirksam).

Die Feldvariable UNPACKED_DEZIMAL ist vom Typ BYTE vereinbart. Da ein Wordtransfer durchgeführt werden soll, ist der Typ-Override-Operator WORD PTR notwendig, um dies dem Assembler mitzuteilen ①.

Bei den Mikroprozessoren 8086/80186/80286 ist es möglich auf die High- und Low-Bytes der WORD-Register AX, BX, CX und DX unabhängig zuzugreifen. Dafür stellt der Assembler eigene Namen, wie AH, AL, BH, BL, CH, CL und DH, DL zur Verfügung.

In Analogie dazu wäre es wünschenswert, wenn für das Low- und High-Byte einer WORD-Variablen oder für das Low- und High-Word einer DOUBLE_WORD-Variablen ebenfalls eigene Namen existierten.

Das angegebene Programmlisting zeigt, wie die Bytes in einer WORD-Variablen durch eigene Namen zu erreichen sind.

```
LOC  OBJ                LINE     SOURCE

                          1              NAME    BEISPIEL_32
                          2
                          3     ASSUME  CS:CODE_PROG,DS:DATEN_PROG
                          4
----                      5     DATEN_PROG SEGMENT
                          6
0000 E823                 7     WORD_VAR        DW      23E8H
  0000                    8     LOW_BYTE  EQU BYTE PTR WORD_VAR      ①
  0001                    9     HIGH_BYTE EQU BYTE PTR WORD_VAR+1
                         10
----                     11     DATEN_PROG ENDS
                         12
----                     13     CODE_PROG SEGMENT
                         14
0000 A00000              15     START:  MOV     AL,LOW_BYTE     ;AL = niederwertiges Byte der
                         16                                     ;WORD-Variablen WORD_VAR
0003 8A260100            17             MOV     AH,HIGH_BYTE    ;AH = hoeherwertiges Byte der
                         18                                     ;WORD-Variablen WORD_VAR
                         19                             ②
----                     20     CODE_PROG ENDS
                         21
                         22             END     START
```

Mit Hilfe des Typ-Override-Operators ist es möglich, ein Byte in einer WORD-Variablen zu adressieren. Wendet man die EQU-Direktive an, so kann z.B. das Symbol LOW_BYTE als Name für das erste Byte der WORD-Variablen WORD_VAR und z.B. HIGH_BYTE als Name für das zweite Byte der WORD-Variablen WORD_VAR definiert werden ①.

Die so vereinbarten Namen LOW_BYTE und HIGH_BYTE können nun benützt werden, um die einzelnen Bytes in der WORD-Variablen WORD_VAR zu adressieren ②.

In den bisherigen Kapiteln wurde neben Anderem die Adressierung von Variablen unter Anwendung der verschiedenen Adressierungsarten behandelt.

Es soll nun am Beispiel der direkten Offset-Adressierung gezeigt werden, wie 8086/80186/80286-Befehle zu codieren sind, wenn der Offset explizit als Zahl angegeben werden soll.

In diesem Fall ist dem Assembler mitzuteilen, daß die benützte Zahl **keine** Konstante, sondern ein Offset-Wert ist. Eine Zahl bekommt damit eine neue Bedeutung, bzw. einen **neuen Typ**. Mit Hilfe des Typ-Override-Operators ist es möglich, eine Zahl als Offset-Wert zu spezifizieren.

Beispiel:

Es sei angenommen, daß ES die Basisadresse einer Datentabelle enthält. In diese Tabelle soll nun ein Word bei Offset 8 relativ zum Beginn der Tabelle aus dem CX-Register geladen werden.

Es liegt nun nahe, folgenden Befehl unter Anwendung der direkten Offset-Adressierung zu codieren: `MOV ES: 8, CX`. Die Zahl 8 ist hier direkt als Offset angegeben.

Da der Assembler dem Befehl nicht entnehmen kann, welcher Variablentyp zu adressieren ist, wird er eine entsprechende Fehlermeldung ausgeben. Damit die Zahl 8 aber trotzdem als Offset einer Word-Variablen benützt werden kann, ist ihr **Typ** von Zahl in WORD umzuwandeln. Dies ist möglich, wenn der Typ-Override-Operator angewendet wird.

Die korrekte Codierung, das CX-Register in ein WORD bei der Position ES: 8 zu laden, ist:

```
MOV ES: WORD PTR 8, CX
```

Wird eine Zahl benützt, um sie als Offsetwert zu spezifizieren, setzt der Assembler für die zugehörigen Segment-Register den speziellen Basiswert NOTHING voraus.

Dies bedeutet, daß der Segment-Override-Operator **in jedem Fall** notwendig ist.

```
LOC  OBJ                    LINE     SOURCE

                              1              NAME    BEISPIEL_33
                              2
                              3      ASSUME CS:CODE_PROG,DS:DATEN_PROG,ES:EXTRA_DATEN_PROG        ①
                              4
----                          5      DATEN_PROG SEGMENT
                              6
0000 (10                      7              DW      10 DUP (?)              ;Daten Tabelle
     ????
     )
                              8
----                          9      DATEN_PROG ENDS
                             10
----                         11      EXTRA_DATEN_PROG SEGMENT
                             12
0000 (10                     13              DW      10 DUP (?)              ;Daten Tabelle
     ????
     )
                             14
----                         15      EXTRA_DATEN_PROG ENDS
                             16
----                         17      CODE_PROG SEGMENT
                             18
0000 B8----          R       19      START:  MOV     AX,DATEN_PROG           ;DS Register initialisieren
0003 8ED8                    20              MOV     DS,AX
0005 B8----          R       21              MOV     AX,EXTRA_DATEN_PROG     ;ES Register initialisieren
0008 8EC0                    22              MOV     ES,AX
000A 26890E0800              23              MOV     ES:WORD PTR 8,CX        ;ES:8 mit CX laden      ②
000F 89160600                24              MOV     DS:WORD PTR 6,DX        ;DS:6 mit DX laden
                             25
----                         26      CODE_PROG ENDS
                             27
                             28              END     START
```

Im angegebenen Programmlisting ist ASSUME DS: DATEN_PROG, ES: EXTRA_DATEN_PROG wirksam ①.

Werden die Befehle MOV WORD PTR 8, CX bzw. MOV WORD PTR 6, DX codiert, wird der Assembler eine Fehlermeldung ausgeben. Er ist nicht informiert, welche Basisregister er für die Offsetwerte 8 bzw. 6 benützen soll.

Diese Information bekommt der Assembler von den Segment-Override-Operatoren. Korrekte Codierungen sind z.B.: MOV ES: WORD PTR 8, CX bzw. MOV DS: WORD PTR 6, DX ②.

4.7 Der TYPE-Operator

Um auf die Elemente eines Feldes zuzugreifen, wird in der Regel das SI- oder das DI-Register benützt. Abhängig vom Typ der Feldvariablen, ist es notwendig diese Register mit dem richtigen Indexwert zu versorgen, um die nächste Dateneinheit zu erreichen.

Beispiel 1: `FELD_1 DB 0, 1, 2, 3;` Feldvariable vom Typ BYTE

Da die Feldelemente vom Typ BYTE sind, kann der Indexwert des nächsten Elements z.B. mit `INC SI` oder `INC DI` gebildet werden.

Beispiel 2: `FELD_2 DW 0, 1, 2, 3;` Feldvariable vom Typ WORD

Hier sind die Feldelemente vom Typ WORD, so daß der Indexwert des nächsten Feldelements z.B. mit `ADD SI, 2` oder `ADD DI, 2` gebildet werden muß. Sollte eine Feldvariable vom Typ DWORD existieren, ist das nächste Feldelement z.B. mit `ADD SI, 4` oder `ADD DI, 4` zu erreichen usw.

Wie die Beispiele zeigen, ist der Indexwert für das nächste Feldelement abhängig vom Typ der Feldvariablen, also von der Anzahl der Bytes, aus der ein Feldelement zusammengesetzt ist.

Damit die Index-Register bei der Bearbeitung von Feldvariablen, abhängig von ihrem Typ, richtig inkrementiert werden, kann der **TYPE**-Operator benützt werden. Dieser Operator liefert einen Wert, der dem Typ eines Operanden entspricht, d. h. die Anzahl von Bytes, aus der dieser Operand zusammengesetzt ist.

Syntaxdiagramm:

- **TYPE** ist ein Schlüsselwort und muß angegeben werden.
- **Variablen-Name** ist der Name einer Variablen.

Je nach Typ der angegebenen Variablen liefert **TYPE** folgende Werte:

- Eine BYTE-Variable liefert den Wert 1
- Eine WORD-Variable liefert den Wert 2
- Eine DWORD-Variable liefert den Wert 4
- Eine QWORD-Variable liefert den Wert 8
- Eine TBYTE-Variable liefert den Wert 10

Anwendungsbeispiel:

Im folgenden Beispiel werden die Feldelemente der beiden Felder FELD_1 und FELD_2 addiert, wobei die Ergebnisse die Elemente in FELD_2 überschreiben.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME BEISPIEL_33_1
                            2
                            3     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                            4
----                        5     DATEN_PROG SEGMENT
                            6
0000 8000                   7     FELD_1  DW      128,612,79
0002 6402
0004 4F00
0006 3F00                   8     FELD_2  DW      63,854,102
0008 5603
000A 6600
                            9
----                       10     DATEN_PROG ENDS
                           11
----                       12     CODE_PROG SEGMENT
                           13
0000 B8----          R     14     START:  MOV     AX,DATEN_PROG          ;DS Register initialisieren
0003 8ED8                  15             MOV     DS,AX
0005 BE0000                16             MOV     SI,OFFSET FELD_1       ;SI = Anfangsadresse von FELD_1
0008 BF0600                17             MOV     DI,OFFSET FELD_2       ;DI = Anfangsadresse von FELD_2
000B B90300                18             MOV     CX,3                   ;CX = Wiederholungszaehler
000E F8                    19             CLC                            ;Loesche Carry Flag
000F 8B04                  20     SUMME:  MOV     AX,[SI]                ;AX = 1. Summand
0011 1105                  21             ADC     [DI],AX                ;Bilde 1.Summand+2.Summand+Carry und
                           22                                            ;ueberschreibe 2.Summanden mit
                           23                                            ;der Summe
0013 83C602                24             ADD     SI,TYPE FELD_1         ;Bilde Index zum naechsten Summanden
                           25                                            ;in FELD_1
0016 83C702                26             ADD     DI,TYPE FELD_2         ;Bilde Index zum naechsten Summanden
                           27                                            ;in FELD_2
0019 E2F4                  28             LOOP    SUMME                  ;Wiederhole, bis alle Summen gebildet
                           29                                            ;sind
                           30
----                       31     CODE_PROG ENDS
                           32
                           33             END     START
```

Beide Felder sind vom Typ WORD, so daß in einfacher Weise mit `ADD SI, TYPE FELD_1` und `ADD DI, TYPE FELD_2` die Indexwerte für die nächsten Summanden in beiden Feldern gebildet werden können.

Die Additionen werden solange durchgeführt, bis die Summe beider Felder gebildet ist. Sollte sich bei der Summenbildung von zwei Feldelementen ein Übertrag ergeben, wird dieser von `ADC [DI], AX` berücksichtigt.