robotron

Anleitung für den
Bediener

Anleitung für den
Programmierer

# Assembler und Debugger

Arbeitsplatzcomputer A 7100 Betriebssystem SCP 1700

| SYSTEMUNTERLAGEN-DOKUMENTATION 6/86 | Assembler und Debugger<br>Anleitung für Bediener und Programmierer | MOS<br>SCP 1700 |
|---|---|---|

Anleitung für Bediener und Programmierer

# Assembler und Debugger

AC A7100

**VEB Robotron-Projekt Dresden**

C 1014-0002-1 M 3030

Die vorliegende Systemunterlagendokumentation, Anleitung für Bediener und Programmierer Assembler und Debugger , entspricht dem Stand von 6/86.



Die Ausarbeitung erfolgte durch ein Kollektiv des VEB Robotron-Elektronik Dresden .

Herausgeber:

VEB Robotron-Projekt Dresden 8010 Dresden, Leningrader Str. 9



Kurzreferat

Diese Anleitung unterstützt den Anwender bei der Programmierung in der Assemblersprache ASM86 des Betriebssystems SCP 1700. Das Grundpaket des SCP 1700 enthält den Assembler ASM86. Zum Test der Programme steht der Debugger DDT86 zur Verfügung. Beide Komponenten werden in dieser Dokumentation beschrieben. Zuerst wird auf die Bedienung des ASM86 eingegangen. Dann werden die Elemente der Assemblersprache definiert, wie Konstanten, Symbole, Operatoren, Ausdrücke usw. Weiterhin werden die Assembler-Direktiven und der Befehlssatz des Assemblers definiert. Auch die Syntax und Anwendung von Code-Makos wird beschrieben. In einem weiteren Abschnitt wird ausführlich die Beschreibung des Debuggers DDT86 erläutert. DDT86 ermöglicht den interaktiven Test und die Korrektur von Maschinencode-Programmen. Die Anwendung des Debuggers wird an einem ausführlichen Beispiel demonstriert.

Inhaltsverzeichnis

Tabellenverzeichnis

# 1. Assembler ASM86

## 1.1. Bedienung

Die Darstellung der Syntax von Kommandos, Befehlen und Direktiven erfolgt in einer allgemeinen Form. Dabei gelten folgende Regeln:

- Elemente in Großbuchstaben stellen syntaktische Konstanten dar, die vom Anwender unverändert genutzt werden müssen. Allerdings ist bei der Anwendung die Groß- oder Kleinschreibung möglich.

- Elemente in Kleinbuchstaben stellen syntaktische Variablen dar. Sie werden im folgenden Text erklärt. Besteht eine Variable aus mehreren Worten, so sind sie durch Bindestrich (-) verbunden.

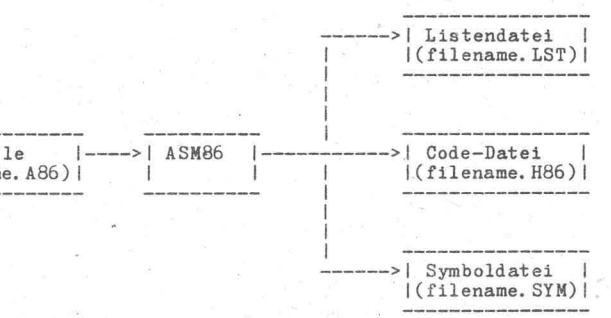
- Elemente in eckigen Klammern [ ] sind wahlweise zu verwenden.

- Alternativen werden durch einen senkrechten Strich | getrennt und können zusätzlich in geschweifte Klammern eingeschlossen sein, um Wiederholungen zu kennzeichnen.

- Punkte (...) zeigen an, daß die vorhergehende Einheit beliebig oft wiederholt werden kann. Falls zur Wiederholung Trennzeichen verwendet werden müssen, steht (, ...).

### 1.1.1. Assembleroperationen

ASM86 übersetzt eine Quelldatei der Assemblersprache in drei Durchläufen. Es entstehen drei Ausgabedateien, einschließlich der Datei in Maschinensprache im Hexadezimalformat.




Die Darstellung zeigt die für ASM86 geltenden Standard-Dateitypen. ASM86 akzeptiert bei einer Quelldatei drei beliebige Buchstaben als Dateityp. Wenn der Dateityp weggelassen wird, sucht ASM86 im Dateiverzeichnis nach A86. Hat die spezifizierte Datei in diesem Fall einen anderen Dateityp als A86 oder fehlt dieser ganz, gibt ASM86 eine Fehlermeldung aus.
Die anderen Dateitypen beziehen sich auf die Ausgabedateien. Die LST-Datei enthält die Übersetzungsliste mit allen Fehlermitteilungen. Die H86-Datei enthält das Programm in Maschinensprache im

Hexadezimalformat. Die SYM-Datei zeigt alle vom Nutzer definierten Symbole.
Der Aufruf von ASM86 erfolgt durch Eingabe eines Kommandos in folgender Form:

ASM86 filespec [¤parameter...]

In Abschnitt 1.1.2. werden die optionalen Parameter erläutert. Die Quelldatei filespec wird wie folgt spezifiziert:

[d:]filename[.typ]

| | |
|---|---|
| d: | zulässiger Buchstabe für die Angabe des Laufwerkes der Quelldatei. Falls sich die Quelle auf dem aktuellen Laufwerk befindet, kann diese Angabe entfallen. |
| filename | zulässiger Dateiname aus 1 bis 8 Zeichen |
| typ | zulässiger Dateityp aus 1 bis 3 Zeichen, gewöhnlich A86 |

Beispiele zulässiger ASM86-Kommandos:

A>ASM86 B:BIOS88

A>ASM86 BIOS88.A86 ¤AA HB PB SB

A>ASM86 D:TEST

Nach dem Aufruf meldet sich ASM86 mit der Ausschrift:

ASM86 V x.x

wobei x.x die Versionsnummer angibt. ASM86 sucht dann nach der angegebenen Quelldatei. Kann die spezifizierte Datei nicht gefunden werden oder hat sie nicht den entsprechenden Dateityp, gibt ASM86 folgende Mitteilung aus:

NO FILE

Falls ein unzulässiger Parameter in der optionalen Parameterliste angegeben wurde, teilt ASM86 mit:

PARAMETER ERROR

Standardmäßig werden die Ausgabedateien auf dem aktuellen Laufwerk abgespeichert. Durch die Benutzung der wahlfreien Parameter bzw. durch eine Laufwerksangabe im Dateinamen der Quelle kann die Ausgabedatei anders gelenkt werden. Während der Übersetzung wird ASM86 unterbrochen, wenn Platten- oder Symboltabellenüberlauf auftritt.

Am Ende einer Übersetzung teilt ASM86 mit:

END OF ASSEMBLY. NUMBER OF ERRORS: n

n ist die Anzahl der Fehler, die ASM86 in der Quelldatei erkannte. Wenn ASM86 einen Fehler in der Quelldatei erkennt, setzt er vor die fehlerhafte Zeile in der Listendatei eine Fehlermitteilung. Jede Fehlermitteilung hat eine Nummer und gibt eine kurze Erklärung des Fehlers. In Anlage 7 sind die Fehlermitteilungen zusammengestellt.

### 1.1.2. Optionale Laufzeitparameter

Das Dollarzeichen ¤ kennzeichnet eine wahlfreie Zeichenkette der Laufzeitparameter. Ein Parameter besteht aus einem Buchstaben für das Parameterkennzeichen, gefolgt von einem Buchstaben für die Gerätespezifikation. Die Parameter zeigt Tabelle 1.

Tabelle 1: Zusammenfassung der Laufzeitparameter

| Parameter | Bedeutung | zulässige Argumente |
|---|---|---|
| A | Quelldatei-Gerät | A, B, C, ... P |
| H | Code-Ausgabedatei-Gerät | A, ... P, X, Y, Z |
| P | Listendatei-Gerät | A, ... P, X, Y, Z |
| S | Symboldatei-Gerät | A, ... P, X, Y, Z |

Alle Parameter sind wählbar und können in der Kommandozeile beliebig angeordnet werden. Das Dollarzeichen steht nur einmal am Anfang der Parameterkette. Leerzeichen können die Parameter trennen, sind aber nicht gefordert. Zwischen dem Parameter und dem Gerätenamen darf kein Leerzeichen stehen. Den Parametern A, H, P und S muß ein Gerätename folgen. Die Geräte werden bezeichnet mit:

A, B, C, ... P oder X, Y, Z.

Die Gerätenamen A bis P bezeichnen die Laufwerke A bis P. X spezifiziert die Bedienkonsole (CON:), Y spezifiziert den Drucker (LST:) und Z unterdrückt die Ausgabe (NUL:).
Falls die Ausgabe zur Konsole geführt wird, kann sie vorübergehend mittels CTRL/S angehalten und durch Eingabe eines zweiten CTRL/S oder eines beliebigen Zeichens fortgesetzt werden.
In Anlage 3 wird das Hexadezimalformat im Detail erklärt.

Tabelle 2: Beispiele für Laufzeitparameter

| Kommandozeile | Ergebnis |
|---|---|
| ASM86 IO | Assemblieren der Datei IO.A86, Erstellen von IO.H86, IO.LST und IO.SYM, alle auf dem Standardlaufwerk |
| ASM86 IO.ASM ¤ AD SZ | Assemblieren der Datei IO.ASM auf Laufwerk D, Erstellen von IO.LST und IO.H86 auf dem Standardlaufwerk, Symboldatei wird unterdrückt |
| ASM86 IO ¤ PY SX | Assemblieren der Datei IO.A86, Erstellen von IO.H86, Liste direkt zum Drucker, Ausgabe der Symboldatei auf der Konsole |

### 1.1.3. Abbruch

ASM86 kann zu jeder Zeit durch Eingabe eines beliebigen Zeichens über die Konsole angehalten werden. Wenn eine Eingabe erfolgt, antwortet ASM86 mit der Frage:

```
USER BREAK. OK(Y/N)?
```

Ein Y bricht sofort den Übersetzungslauf ab und die Steuerung geht an das Betriebssystem zurück. Ein N setzt die Arbeit des ASM86 fort.

## 1.2. Elemente der ASM86-Assemblersprache

### 1.2.1. Zeichensatz

ASM86 benutzt eine Untermenge des KOI7-Zeichensatzes. Die zulässigen Zeichen sind die alphanumerischen, die Spezialzeichen und nichtdruckbaren Zeichen:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

+ - * / = ( ) [ ] ; ' . ! , _ : @ ¤ ?

Leerzeichen, Tabulator, Wagenrücklauf, Zeilenschaltung
```

Kleinbuchstaben werden wie Großbuchstaben behandelt, ausgenommen innerhalb von Zeichenketten, in denen nur alphanumerische, spezielle und Leerzeichen benutzt werden dürfen.

### 1.2.2. Komponenten und Trennzeichen

Eine Komponente (token) ist die kleinste signifikante Einheit des ASM86-Quellprogramms. Benachbarte Komponenten werden durch ein Leerzeichen getrennt. Überall, wo ein Leerzeichen erlaubt ist, dürfen auch mehrere Leerzeichen stehen. ASM86 erkennt Horizontaltabulatoren als Trennzeichen an und interpretiert sie als Leerzeichen. In der Listendatei werden Tabulatoren durch Leerzeichen ersetzt. Die Tabulatorhalts sind auf jeder achten Spalte.

### 1.2.3. Begrenzer

Begrenzer (delimiter) markieren das Ende einer Komponente und geben einem Befehl (instruction) eine spezielle Bedeutung, im Gegensatz zu Trennzeichen, die lediglich das Ende einer Komponente markieren. Beim Auftreten von Begrenzern brauchen Trennzeichen nicht verwendet werden. Trennzeichen nach Begrenzern können jedoch Programme leichter lesbar machen.
Tabelle 3 beschreibt die Trennzeichen und Begrenzer des ASM86. Einige Begrenzer sind außerdem Operatoren und werden in Abschnitt 1.2.6. näher erklärt.

Tabelle 3: Trennzeichen und Begrenzer

| Zeichen | Name | Benutzung |
|---|---|---|
| 20H | Leerzeichen | Trennzeichen |
| 09H | Tabulator | Trennzeichen, erlaubt in Quelldateien, wirksam in Listendateien |
| (CR) | Wagenrücklauf | beendet eine Quellzeile |
| (LF) | Zeilenschaltung | erlaubt nach (CR); innerhalb einer Quellzeile wird es als Leerzeichen interpretiert. |
| ; | Semikolon | Beginn des Kommentarfeldes |
| : | Doppelpunkt | kennzeichnet eine Marke; wird benutzt bei der Segment-Override-Spezifikation |
| . | Punkt | bildet Variable aus Zahlen |
| ¤ | Dollarzeichen | aktueller Speicherplatzzähler |
| + | Plus | Arithmetikoperator für Addition |
| - | Minus | Arithmetikoperator für Subtraktion |
| * | Stern | Arithmetikoperator für Multiplikation |
| / | Schrägstrich | Arithmetikoperator für Division |
| @ | kommerzielles a | zulässig in Bezeichnern |
| _ | Unterstreichung | zulässig in Bezeichnern |
| ! | Ausrufezeichen | logisches Ende für eine Anweisung; auf einer Zeile sind mehrere Anweisungen möglich. |
| ' | Apostroph | begrenzt Zeichenkonstanten |

### 1.2.4. Konstanten

Eine Konstante ist ein zur Zeit der Übersetzung bekannter Wert, der sich während der Programmausführung nicht ändert. Eine Konstante kann entweder eine Integerzahl oder eine Zeichenkette sein.

#### 1.2.4.1. Numerische Konstanten

Eine numerische Konstante ist ein 16-Bit-Wert, der sich auf eine von mehreren Basen bezieht. Die Zahlenbasis (radix) wird durch einen der Zahl folgenden Indikator angegeben. Die Zahlenbasis-Indikatoren zeigt Tabelle 4.

Tabelle 4: Zahlenbasis-Indikatoren für Konstanten

| Indikator | Konstantentyp | Basis |
|---|---|---|
| B | binär | 2 |
| O | oktal | 8 |
| Q | oktal | 8 |
| D | dezimal | 10 |
| H | hexadezimal | 16 |

ASM86 setzt voraus, daß jede Konstante, der kein Zahlenbasis-Indikator folgt, dezimal ist. Zahlenbasis-Indikatoren können mit Klein- oder Großbuchstaben angegeben werden.
Eine Konstante ist demzufolge eine Folge von Ziffern mit einem nachfolgenden Zahlenbasis-Indikator, wobei sich die Ziffern im Bereich der Zahlenbasis bewegen. Binärkonstanten bestehen aus Nullen und Einsen. Oktalkonstanten enthalten Ziffern von 0 ... 7, dezimale Konstanten enthalten Ziffern von 0 ... 9. Hexadezimale Konstanten enthalten dezimale Ziffern und die hexadezimalen Ziffern A (10D), B (11D), C (12D), D (13D), E (14D) und F (15D). Das führende Zeichen einer hexadezimalen Konstanten muß eine Dezimalziffer oder Null sein, da ASM86 sonst nicht in der Lage ist, eine hexadezimale Konstante von einem Bezeichner zu unterscheiden.

Beispiele zulässiger Konstanten:

```
1234    1234D   1100B   1111000011110000B

1234H   0FFEH   33770   13772Q

33770   0FE3H   1234D   0FFFFH
```

#### 1.2.4.2. Zeichenketten

Eine Zeichenkettenkonstante ist eine Reihe von KOI7-Zeichen, die durch Apostrophe begrenzt werden. Alle Befehle, die numerische Konstanten als Argumente zulassen, akzeptieren nur Zeichenkettenkonstanten aus einem oder zwei Zeichen als zulässige Argumente. Alle Befehle behandeln Ein-Zeichen-Konstanten als 8-Bit-Wert. Eine Zwei-Zeichen-Kette wird als 16-Bit-Wert dargestellt, wobei der Wert des zweiten Zeichens im niederwertigen und der des ersten Zeichens im höherwertigen Byte steht.

Der numerische Wert eines Zeichens ist sein KOI7-Code. ASM86 übersetzt innerhalb von Zeichenketten nicht, so daß sowohl Klein- als auch Großbuchstaben verwendet werden können. Es ist zu beachten, daß in Zeichenketten nur alphanumerische, Sonder- und Leerzeichen verwendet werden dürfen.
Die DB-Direktive ist die einzige Anweisung, die mehr als zwei Zeichen in einer Zeichenkette zuläßt. Die Zeichenkette darf 255 Bytes nicht überschreiten. Soll innerhalb der Zeichenkette ein Apostroph gedruckt werden, ist der Apostroph doppelt anzugeben. ASM86 interpretiert zwei Apostrophe als einen.

In den folgenden Beispielen werden zulässige Zeichenketten und ihre Wirkung bei Ausführung gegenübergestellt.

```
          'a'  -->  a

      'Ab''Cd'  -->  Ab'Cd

  'I like SCP'  -->  I like SCP

          ''''  -->  '

'ONLY UPPER CASE'  -->  ONLY UPPER CASE

'only lower case'  -->  only lower case
```

### 1.2.5. Bezeichner

Bezeichner (identifier) sind Zeichenfolgen, die für den Assembler eine besondere symbolische Bedeutung haben. Alle Bezeichner im ASM86 müssen den folgenden Regeln genügen:

- Das erste Zeichen muß ein Buchstabe sein (A ... Z, a ... z).
- Jedes nachfolgende Zeichen kann entweder ein Buchstabe oder eine Zahl sein (0 ... 9). ASM86 ignoriert die Sonderzeichen @ und _. Sie sind jedoch zulässig, so wird z.B. a_b zu ab.
- Bezeichner können innerhalb einer physischen Zeile jede beliebige Länge annehmen.

Es gibt zwei Arten von Bezeichnern. Die erste Art sind Schlüsselworte, die für den Assembler eine vorgegebene Bedeutung haben. Die zweite Art von Bezeichnern sind Symbole, die vom Nutzer definiert werden. Die folgenden sind zulässige Bezeichner:

```
NOLIST
WORD
AH
THIRD STREET
HOW_ARE_YOU_TODAY
VARIABLE@NUMBER@1234567890
```

### 1.2.5.1. Schlüsselworte

Ein Schlüsselwort ist ein Bezeichner, der für den Assembler eine vorgegebene Bedeutung hat. Schlüsselworte sind reserviert. Der Nutzer darf keinen Bezeichner definieren, der mit einem Schlüsselwort identisch ist. In Anlage 4 ist eine komplette Liste aller Schlüsselworte zusammengestellt.
ASM86 erkennt fünf Typen von Schlüsselworten:

- Befehle
- Direktiven
- Operatoren
- Register
- vordefinierte Zahlen

Die Befehls-Schlüsselworte und ihre Wirkung werden in Abschnitt 1.4. beschrieben. Direktiven werden in Abschnitt 1.3. und Operatoren in Abschnitt 1.2.6. definiert. Tabelle 5 enthält die ASM86-Schlüsselworte, die Register darstellen.
Drei Schlüsselworte sind vordefinierte Zahlen: BYTE, WORD und DWORD. Die Werte dieser Zahlen sind 1, 2, 4. Mit jeder dieser Zahlen ist ein Typattribut verbunden. Das Typattribut ist gleich dem numerischen Wert des Schlüsselwortes. Im Abschnitt 1.2.5.2. werden Typattribute vollständig erläutert.

Tabelle 5: Registerschlüsselworte

| Register Symbol | Größe in Byte | numerischer Wert | Bedeutung |
|---|---|---|---|
| AH | 1 | 100B | Akkumulator, höherwertiges Byte |
| BH | 1 | 111B | Basisregister, höherwertiges Byte |
| CH | 1 | 101B | Zählregister, höherwertiges Byte |
| DH | 1 | 110B | Datenregister, höherwertiges Byte |
| AL | 1 | 000B | Akkumulator, niederwertiges Byte |
| BL | 1 | 011B | Basisregister, niederwertiges Byte |
| CL | 1 | 001B | Zählregister, niederwertiges Byte |
| DL | 1 | 010B | Datenregister, niederwertiges Byte |
| AX | 2 | 000B | Akkumulator, Wort |
| BX | 2 | 011B | Basisregister, Wort |
| CX | 2 | 001B | Zählregister, Wort |
| DX | 2 | 010B | Datenregister, Wort |
| BP | 2 | 101B | Basiszeiger (Basepointer) |
| SP | 2 | 100B | Kellerzeiger (Stackpointer) |
| SI | 2 | 110B | Quellindex (Source-Index) |
| DI | 2 | 111B | Zielindex (Destination-Index) |
| CS | 2 | 01B | Code-Segmentregister |
| DS | 2 | 11B | Data-Segmentregister |
| SS | 2 | 10B | Stack-Segmentregister |
| ES | 2 | 00B | Extra-Segmentregister |

### 1.2.5.2. Symbole und ihre Attribute

Ein Symbol ist ein vom Nutzer definierter Bezeichner, dessen Attribute Auskunft über die Informationsart des Symbols geben. Es gibt drei Kategorien von Symbolen:

- Variable
- Marken
- Zahlen

#### Variable

Eine Variable identifiziert die gespeicherten Daten in einem bestimmten Speicherplatz. Alle Variablen haben die folgenden drei Attribute:

- Segment: legt fest, in welchem Segment die Variable definiert wurde.

- Offset: legt fest, wieviel Bytes sich zwischen dem Anfang des Segments und der Speicheradresse befinden, die diese Variable bezeichnet.

- Typ: legt fest, wieviel Bytes behandelt werden, falls zu dieser Variablen zugegriffen wird.

Ein Segment kann ein Code-Segment, Data-Segment, Stack-Segment oder Extra-Segment sein. Das hängt von seinem Inhalt und dem Register ab, das die Startadresse des Segments enthält (siehe Abschnitt 1.3.2.). Ein Segment kann an einer beliebigen durch 16 teilbaren Adresse beginnen. ASM86 benutzt diese Adresse als Segmentwert zur Definition der Variablen.
Der Offset einer Variablen kann eine Zahl zwischen 0 und 0FFFFH oder 65535D sein. Eine Variable muß einen der folgenden Typen haben:

- BYTE
- WORD
- DWORD

BYTE gibt eine Ein-Byte-Variable, WORD eine Zwei-Byte-Variable und DWORD eine Vier-Byte-Variable an. Die Direktiven DB, DW und DD definieren Variable dieser drei Typen (siehe Abschnitt 1.3.). Eine Variable wird z.B. als Name vor einer Speicher-Direktive definiert:

```
VARIABLE DB 0
```

Über die EQU-Anweisung kann auch zu einer anderen Variablen zugegriffen werden.

```
VARIABLE EQU ANOTHER_VARIABLE
```

Marken

Marken kennzeichnen Speicherplätze, die Befehlsanweisungen enthalten. Sprünge (jumps) und Rufe (calls) beziehen sich auf Marken. Alle Marken haben zwei Attribute:

- Segment
- Offset

Markenattribute sind im wesentlichen gleich denen der Variablen. Eine Marke wird im allgemeinen einem Befehl vorangestellt und durch einen Doppelpunkt von ihm getrennt.

```
LABEL:  ADD  AX,BX
```

Eine Marke kann auch vor der EQU-Anweisung stehen (ohne Doppelpunkt), der eine andere Marke zugeordnet wird.

```
LABEL   EQU  ANOTHER_LABEL
```

Zahlen

Zahlen können auch wie Symbole definiert werden. Ein Zahlensymbol wird so behandelt, als wäre die vom Symbol dargestellte Zahl explizit codiert.

```
NUMBER_FIVE  EQU  5
             MOV  AL,NUMBER_FIVE
```

In Abschnitt 1.2.6. werden die Operatoren und ihre Wirkung auf Zahlen und Zahlensymbole beschrieben.

### 1.2.6. Operatoren

Es gibt verschiedene Arten von Operatoren:

- arithmetische Operatoren
- logische Operatoren
- Vergleichsoperatoren
- Segment-Override-Operatoren
- Attributoperatoren

Tabelle 6 definiert die ASM86-Operatoren. In dieser Tabelle stellen a und b zwei Elemente des Ausdrucks dar. Die Gültigkeitsspalte definiert den Typ der Operanden, die der Operator behandeln kann.

Tabelle 6: ASM86-Operatoren

| Syntax | Ergebnis | Gültigkeit |
|---|---|---|
| | **logische Operatoren** | |
| a XOR b | logisches exklusives OR von a und b, Bit für Bit | a,b = Zahl |
| a OR b | logisches OR von a und b, Bit für Bit | a,b = Zahl |
| a AND b | logisches AND von a und b, Bit für Bit | a,b = Zahl |
| NOT a | logische Inversion von a; 0 --> 1, 1 --> 0 | a = 16-Bit-Zahl |
| | **Vergleichsoperatoren** | |
| a EQ b | Ergebnis OFFFFH, falls a=b, sonst Null | a,b = vorzeichenlose Zahl |
| a LT b | Ergebnis OFFFFH, falls a<b, sonst Null | a,b = vorzeichenlose Zahl |
| a LE b | Ergebnis OFFFFH, falls a<=b, sonst Null | a,b = vorzeichenlose Zahl |
| a GT b | Ergebnis OFFFFH, falls a>b, sonst Null | a,b = vorzeichenlose Zahl |
| a GE b | Ergebnis OFFFFH, falls a>=b, sonst Null | a,b = vorzeichenlose Zahl |
| a NE b | Ergebnis OFFFFH, falls a<>b, sonst Null | a,b = vorzeichenlose Zahl |

Tabelle 6: (Fortsetzung)

| Syntax | Ergebnis | Gültigkeit |
|---|---|---|
| | arithmetische Operatoren | |
| a + b | arithmetische Summe aus a und b | a = Variable, Marke oder Zahl, b = Zahl |
| a - b | arithmetische Differenz von a und b | a = Variable, Marke oder Zahl, b = Zahl |
| a * b | vorzeichenlose Multiplikation von a und b | a,b = Zahl |
| a / b | vorzeichenlose Division von a und b | a,b = Zahl |
| a MOD b | Rest von a / b | a,b = Zahl |
| a SHL b | Wert, der sich durch b-malige Linksverschiebung von a ergibt | a,b = Zahl |
| a SHR b | Wert, der sich durch b-malige Rechtsverschiebung von a ergibt | a,b = Zahl |
| +a | ergibt a | a = Zahl |
| -a | ergibt 0 - a | a = Zahl |
| | Segment-Override | |
| seg reg: addr exp | Änderung des Segmentregisters für Assembler | seg-reg = CS, DS, SS oder ES |

Tabelle 6: (Fortsetzung)

| Syntax | Ergebnis | Gültigkeit |
|---|---|---|
| | Attributoperatoren | |
| SEG a | erzeugt eine Zahl, deren Wert der Segment-Wert der Variablen oder Marke ist | a = Marke oder Variable |
| OFFSET a | erzeugt eine Zahl, deren Wert der Offset-Wert der Variablen oder Marke ist | a = Marke oder Variable |
| TYPE a | erzeugt eine Zahl, die den Wert 1, 2 oder 4 hat, wenn a vom Typ BYTE, WORD oder DWORD ist | a = Marke oder Variable |
| LENGTH a | erzeugt eine Zahl, deren Wert das Längenattribut von a ist. Das Längenattribut ist die zur Variablen gehörende Byteanzahl. | a = Marke oder Variable |
| LAST a | falls Länge a $\geq$ 0, dann LAST a = LENGTH a-1; falls a = 0, dann LAST a = 0 | a = Marke oder Variable |
| a PTR b | erzeugt eine virtuelle Variable mit dem Typ von a und den Attributen von b | a = BYTE, WORD oder DWORD<br>b = addr exp |
| .a | erzeugt eine Variable mit dem Offset-Attribut von a. Das Segment-Attribut ist das aktuelle Segment. | a = Zahl |
| ¤ | erzeugt eine Marke mit dem Offset zum aktuellen Speicherplatz. Das Segment-Attribut wird das aktuelle Segment. | kein Argument |

### 1.2.6.1. Operatorbeispiele

Logische Operatoren akzeptieren nur Zahlen als Operanden. Sie führen die Boolschen logischen Operationen AND, OR, XOR und NOT aus.

Beispiel:

```
MASK    EQU     0FCH
SIGNBIT EQU     80H
        MOV     CL,MASK AND SIGNBIT
        MOV     AL,NOT MASK
```

Vergleichsoperatoren behandeln alle Operanden als vorzeichenlose Zahlen. Die Vergleichsoperatoren sind EQ (equal), LT (less than), LE (less than or equal), GE (greater than or equal), GT (greater than) und NE (not equal). Jeder Operator vergleicht zwei Operanden und liefert den Wert OFFFFH, wenn der angegebene Vergleich 'wahr' ist bzw. den Wert Null, falls der Vergleich 'falsch' ist. .

Beispiel:

```
LIMIT1  EQU     10
LIMIT2  EQU     25
        .
        .
        .
        MOV     AX,LIMIT1 LT LIMIT2
        MOV     AX,LIMIT1 NE LIMIT2
```

Additions- und Subtraktionsoperatoren berechnen die arithmetische Summe bzw. Differenz von zwei Operanden. Der erste Operand kann eine Variable, Marke oder Zahl, der zweite Operand muß bei Addition eine Zahl sein.
Wenn eine Zahl zu einer Variablen oder Marke addiert wird, so ist das Ergebnis eine Variable oder Marke, deren Offset der numerische Wert des zweiten Operanden plus dem Offset des ersten Operanden ist.
Die Subtraktion von einer Variablen oder Marke liefert eine Variable oder Marke, deren Offset die Differenz des Offsets des ersten Operanden und der im zweiten Operanden angegebenen Zahl ist.

Beispiel:

```
COUNT   EQU     2
DISP    EQU     5
FLAG    DB      OFFH
        .
        .
        .
        MOV     AL,FLAG+1
        MOV     CL,FLAG+DISP
        MOV     BL,DISP-COUNT
```

Die Multiplikations- und Divisionsoperatoren *, /, MOD, SHL und SHR akzeptieren nur Zahlen als Operanden. * und / behandeln alle Operatoren als vorzeichenlose Zahlen.

Beispiel:

```
        MOV     SI,256/3
        MOV     BL,64/4
BUFFERSIZE EQU  80
        MOV     AX,BUFFERSIZE*2
```

Einstellige Operatoren können sowohl vorzeichenlose, als auch Operatoren mit Vorzeichen sein.

Beispiel:

```
        MOV     CL,+35
        MOV     AL,2--5
        MOV     DL,-12
```

Wenn Variable behandelt werden, entscheidet der Assembler, welches Segmentregister zu benutzen ist. Diese Entscheidung kann durch Spezifizierung eines anderen Registers mittels des Segment-Override übergangen werden. Die Syntax für den Segment-Override-Operator ist:

segment-register : address-expression

Für segment-register steht CS, DS, SS oder ES.

Beispiel:

```
        MOV     AX,SS:BUFFER[BX]
        MOV     CX,ES:ARRAY
```

Ein Attributoperator erstellt eine Zahl, die gleich einem Attributwert seines Variablenoperanden ist. SEG bildet den Segmentwert der Variablen, OFFSET ihren Offset-Wert, TYPE ihren Typwert (1, 2, 4) und LENGTH die Byteanzahl, die zur Variablen gehört. LAST vergleicht die Länge der Variablen mit 0 und, falls sie größer ist, wird die Länge LENGTH um 1 vermindert. Falls LENGTH gleich 0 ist, erfolgt keine Änderung. Attributoperatoren akzeptieren nur Variable als Operanden.

Beispiel:

```
WORDBUFFER DW      0,0,0
BUFFER     DB      1,2,3,4,5
             .
             .
             .
           MOV     AX,LENGTH BUFFER
           MOV     AX,LAST BUFFER
           MOV     AX,TYPE BUFFER
           MOV     AX,TYPE WORDBUFFER
```

Der PTR-Operator erstellt eine virtuelle Variable oder Marke, die nur während der Befehlsausführung zulässig ist. Die Operanden werden nicht verändert. Das temporäre Symbol hat das gleiche Typattribut wie der linke Operator und alle anderen Attribute das des rechten Operators.

Beispiel:

```
        MOV     BYTE PTR [BX],5
        MOV     AL,BYTE PTR [BX]
        INC     WORD PTR [SI]
```

Der Punkt-Operator (.) erstellt eine Variable im aktuellen Data-Segment. Die neue Variable hat ein Segment-Attribut gleich dem des aktuellen Data-Segments und ein Offset-Attribut gleich seinem Operanden. Sein Operand muß eine Zahl sein.

Beispiel:

```
MOV     AX,.0
MOV     BX,ES:.4000H
```

Der Dollar-Operator (¤) erstellt eine Marke mit dem Offset-Attribut gleich dem Wert des aktuellen Speicherplatzzählers (location counter). Der Segmentwert der Marke ist der gleiche wie der des aktuellen Segments. Der Operator benötigt keinen Operanden.

Beispiel:

```
JMP     ¤
JMPS    ¤
JMP     ¤+3000H
```

### 1.2.6.2. Vorrang bei Operatoren

In Ausdrücken werden Variable, Marken oder Zahlen mit Operatoren kombiniert. ASM86 erlaubt verschiedene Arten von Ausdrücken, die in Abschnitt 1.2.7. erklärt werden. In diesem Abschnitt wird die Reihenfolge definiert, in der Operationen ausgeführt werden, falls mehr als ein Operator in einem Ausdruck auftritt.
Im allgemeinen berechnet ASM86 Ausdrücke von links nach rechts; wobei Operatoren mit höherem Vorrang vor denen mit niederem Vorrang ausgeführt werden. Wenn zwei Operatoren den gleichen Vorrang haben, wird von links nach rechts abgearbeitet. Mit runden Klammern können die Vorrangregeln übergangen werden. Falls Klammern gesetzt sind, wird die innerste zuerst ausgeführt. Es sind maximal fünf Schachtelungen mit Klammern zulässig.

Beispiel:

```
15 / 3 + 18 / 9 = 5 + 2 = 7
15 / (3 + 18 / 9) = 15 / (3 + 2) = 15 / 5 = 3
```

Tabelle 7: Vorrang der Operatoren bei ASM86

| Rang-folge | Operatortyp | Operatoren |
|---|---|---|
| 1 | logisch | XOR, OR |
| 2 | logisch | AND |
| 3 | logisch | NOT |
| 4 | Vergleichsoperatoren | EQ, LT, LE, GT, GE, NE |
| 5 | Addition/Subtraktion | +, - |
| 6 | Multiplikation/Division | *, /, MOD, SHL, SHR |
| 7 | einstellige Operatoren | +, - |
| 8 | Segment-Override | segment-override: |
| 9 | Attributoperatoren | SEG, OFFSET, PTR, TYPE, LENGTH, LAST |
| 10 | runde, eckige Klammern | ( ), [ ] |
| 11 | Punkt, Dollar | ., $ |

## 1.2.7. Ausdrücke

ASM86 erlaubt Adreß-, numerische und Klammerausdrücke.
Ein Adreßausdruck ergibt eine Speicheradresse und hat drei Komponenten:

- Segment-Wert
- Offset-Wert
- Typ

Sowohl Variablen als auch Marken sind Adreßausdrücke. Die Komponenten eines Adreßausdruckes sind Zahlen. Sie können mit Operatoren kombiniert werden, z. B. mit PTR, um einen Adreßausdruck zu erhalten.
Ein numerischer Ausdruck darf weder Variable noch Marken enthalten, nur Zahlen und Operanden. Das Ergebnis eines numerischen Ausdrucks ist eine Zahl.
Ausdrücke in eckigen Klammern geben Basis- und Indexadressierungsmodi an. Die Basisregister sind BX und BP, die Indexregister DI und SI. Ein Ausdruck in eckigen Klammern kann aus einem Basisregister, einem Indexregister oder aus einem Basis- und einem Indexregister bestehen.
Der Operator + muß zwischen einem Index- und Basisregister zur Darstellung der Index- und Basisregisteradressen benutzt werden.

Beispiel:

```
MOV     VARIABLE[BX],0
MOV     AX,[BX+DI]
MOV     AX,[SI]
```

## 1.2.8. Anweisungen

Eine Anweisung (statement) teilt dem Assembler mit, welche Handlung auszuführen ist. Es gibt zwei Typen von Anweisungen: Befehle (instruction) und Direktiven (directive). Befehle werden vom Assembler in die Maschinensprache übersetzt. Direktiven werden nicht in den Maschinencode übersetzt. Der Assembler führt bestimmte Steuerfunktionen aus.
Jede Assembleranweisung muß mit Wagenrücklauf (CR) und Zeilenschaltung (LF) oder mit einem Ausrufezeichen (!) abgeschlossen werden, damit der Assembler das Ende der Zeile erkennt. Mehrere Anweisungen (ohne Kommentar) können auf einer Zeile geschrieben werden, wenn sie durch ! getrennt werden.

Der ASM86-Befehlssatz wird im Abschnitt 1.4. definiert. Die Syntax für Befehlsanweisungen ist folgende:

```
[label:] [prefix] mnemonic [operand...][;comment]
```

Es bedeuten:

label — Ein Symbol, gefolgt von Doppelpunkt (:), definiert eine Marke für den aktuellen Wert des Adreßzählers (location counter) im aktuellen Code-Segment. Das Feld ist wahlfrei.

prefix — Bestimmte Präfix-Befehle, wie z.B. LOCK und REP, können vor Maschinenbefehlen stehen. Das Feld ist wahlfrei.

mnemonic — Ein Symbol, das als Maschinenbefehl definiert wird, was vom Assembler selbst oder durch eine EQU-Direktive erfolgen kann. Dieses Feld ist wahlfrei, falls es nicht durch einen Präfix-Befehl eingeleitet wird. Falls es weggelassen wird, dürfen keine Operanden angegeben werden. Die anderen Felder können angegeben werden. ASM86-Mnemoniks sind im Abschnitt 1.4. definiert.

operand — Ein Befehlsmnemonik kann weitere Symbole fordern, die zum Befehl gehörende Operanden darstellen. Befehle können keinen, einen oder zwei Operanden haben.

comment — Jedes Semikolon (;) außerhalb einer Zeichenkette definiert den Beginn eines Kommentars, der mit Wagenrücklauf endet. Kommentare verbessern die Lesbarkeit von Programmen. Das Feld ist wahlfrei.

ASM86-Direktiven werden in Abschnitt 1.3. beschrieben. Die Syntax für Direktiveanweisungen ist folgende:

```
[name] directive operand... [;comment]
```

Es bedeuten:

name — Im Gegensatz zum Markenfeld eines Befehls wird das Namenfeld einer Direktive niemals mit einem Doppelpunkt begrenzt. Direktivenamen sind nur zulässig für DB, DW, DD, RS und EQU. Bei DB, DW, DD und RS ist der Name wahlfrei, bei EQU muß ein Name stehen.

directive — Direktiveschlüsselwort (siehe Abschnitt 1.3.)

operand — Es besteht Analogie zu den Operanden der Befehlsmnemoniks. Einige Direktiven lassen jeden Operanden zu (DB, DW und DD), während andere spezielle Forderungen stellen.

comment — Kommentar (siehe comment bei Befehlsanweisungen)

## 1.3. Assembler-Direktiven

Direktiveanweisungen veranlassen den Assembler, Verwaltungsfunktionen, wie die Zuweisung von Codeteilen zu logischen Segmenten, die bedingte Assemblierung, die Definition von Datenelementen oder die Spezifizierung des Listendateiformats durchzuführen. Die allgemeine Syntax für Direktiveanweisungen ist in Abschnitt 1.2.8. angegeben.

### 1.3.1. Segment-Start-Direktive

Zur Laufzeit muß jeder Speicherbezug eine 16-Bit-Segmentbasis und einen 16-Bit-Offset haben. Diese werden zu einer effektiven 20-Bit-Adresse kombiniert, damit die ZVE eine Speicherzelle physisch adressieren kann. Die 16-Bit-Segmentbasis oder -grenze ist in einem der Segmentregister CS, DS, SS oder ES enthalten. Der Offset-Wert gibt den Abstand des Speicherbezugs von der Segmentgrenze an. Ein 16-Byte-Segment (physisch) ist die kleinste verschiebliche Speichereinheit.
ASM86 kennt vier logische Segmente: Code-Segment, Data-Segment, Stack-Segment und Extra-Segment, die durch die entsprechenden Register CS, DS, SS und ES adressiert werden. Alle Assembleranweisungen müssen einem dieser Segmente zugewiesen sein, so daß auf sie durch die ZVE Bezug genommen werden kann.
Eine Segment-Direktive-Anweisung CSEG, DSEG, SSEG oder ESEG gibt an, daß die folgenden Anweisungen zu einem spezifischen Segment gehören. Die Anweisungen werden durch das entsprechende Segmentregister adressiert. ASM86 weist Anweisungen dem entsprechenden Segment zu bis eine neue Segment-Direktive erkannt wird.
Befehlsanweisungen müssen dem Code-Segment zugewiesen werden. Direktiveanweisungen können jedem Segment zugewiesen werden. ASM86 nutzt diese Zuweisungen, um von einem Segmentregister zu einem anderen zu wechseln. Wenn z.B. ein Befehl auf eine Speichervariable zugreift, muß ASM86 wissen, welches Segment diese Variable enthält, so daß, falls notwendig, ein Segment-Override generiert werden kann.

### 1.3.1.1. CSEG-Direktive

```
CSEG  numeric-expression
CSEG
CSEG  ¤
```

Diese Direktive teilt dem Assembler mit, daß die folgenden Anweisungen zum Code-Segment gehören. Alle Befehlsanweisungen müssen dem Code-Segment zugewiesen werden. Alle Direktiveanweisungen sind innerhalb des Code-Segments zulässig.
Die erste Form wird benutzt, wenn die Segmentposition zum Übersetzungszeitpunkt bekannt ist. Der generierte Code ist nicht verschieblich. Die zweite Form wird benutzt, wenn die Segmentposition nicht bekannt ist. Der generierte Code ist verschieblich. Die dritte Form dient der Fortsetzung des Code-Segments, nachdem es durch eine DSEG-, SSEG- oder ESEG-Direktive unterbrochen worden ist. Das fortsetzende Code-Segment beginnt mit den gleichen Attributen (Speicherplatz- und Befehlszeiger) wie das vorhergehende Code-Segment.

### 1.3.1.2. DSEG-Direktive

```
DSEG  numeric-expression
DSEG
DSEG  ¤
```

Diese Direktive gibt an, daß die folgenden Anweisungen zum Data-Segment gehören. Das Data-Segment enthält in erster Linie die Datenzuweisungsdirektiven DB, DW, DD und RS; aber alle anderen Direktiveanweisungen sind auch zulässig. Befehle sind in Data-Segmenten nicht erlaubt.
Die erste Form wird benutzt, wenn die Segmentposition zum Übersetzungszeitpunkt bekannt ist. Der generierte Code ist nicht verschieblich. Die zweite Form wird benutzt, wenn die Segmentposition nicht bekannt ist. Der generierte Code ist verschieblich. Die dritte Form dient zur Fortsetzung des Data-Segments, wenn es durch eine CSEG-, SSEG- oder ESEG-Direktive unterbrochen worden ist. Das fortsetzende Data-Segment beginnt mit den gleichen Attributen wie das vorhergehende.

### 1.3.1.3. SSEG-Direktive

```
SSEG  numeric-expression
SSEG
SSEG  ¤
```

Die SSEG-Direktive zeigt den Beginn von Quellzeilen für das Stack-Segment an. Das Stack-Segment wird für alle Stack-Operationen benutzt. Im Stack-Segment sind alle Direktiveanweisungen zulässig, aber Befehle sind nicht erlaubt.
Die erste Form wird benutzt, wenn die Segmentposition zum Übersetzungszeitpunkt bekannt ist. Der generierte Code ist nicht verschieblich. Die zweite Form wird benutzt, wenn die Segmentposition nicht bekannt ist. Der generierte Code ist verschieblich. Die dritte Form dient zur Fortsetzung eines durch die CSEG-, DSEG- oder ESEG-Direktiven unterbrochenen Stack-Segments. Die Fortsetzung erfolgt mit den gleichen Attributen des vorhergehenden Stack-Segments.

### 1.3.1.4. ESEG-Direktive

```
ESEG  numeric-expression
ESEG
ESEG  n
```

Diese Direktive führt das Extra-Segment ein. Außer Befehlen sind alle Direktiveanweisungen zulässig. Die erste Form wird benutzt, wenn die Segmentposition zum Übersetzungszeitpunkt bekannt ist. Der generierte Code ist nicht verschieblich. Die zweite Form wird benutzt, wenn die Segmentposition nicht beannt ist. Der generierte Code ist verschieblich. Die dritte Form dient zur Fortsetzung eines durch DSEG-, SSEG- oder CSEG-Direktiven unterbrochenen Extra-Segments. Das fortsetzende Extra-Segment beginnt mit den gleichen Attributen wie das vorhergehende.

### 1.3.2. ORG-Direktive

```
ORG  numeric-expression
```

Die ORG-Direktive setzt den Offset des Speicherplatzzählers im aktuellen Segment auf den im numerischen Ausdruck angegebenen Wert. Alle Elemente des Ausdrucks sind vor seiner Anwendung zu definieren, da Vorwärtsreferenzen mehrdeutig sein können. In den meisten Segmenten ist eine ORG-Direktive nicht nötig. Wenn vor der ersten Befehlsanweisung oder vor dem ersten Datenbyte keine ORG-Direktive angegeben ist, beginnt die Übersetzung ab Position Null relativ zum Anfang des Segments. Ein Segment kann eine beliebige Anzahl von ORG-Direktiven enthalten.

### 1.3.3. IF- und ENDIF-Direktiven

```
IF  numeric-expression
      source-line 1
      source-line 2
            .
            .
            .
      source-line n
ENDIF
```

Die IF- und ENDIF-Direktiven ermöglichen es, eine Gruppe von Quellzeilen in die Übersetzung einzubeziehen oder sie davon auszuschließen. Diese bedingte Übersetzung wird benutzt, um mit einem Quellprogramm mehrere unterschiedliche Versionen herstellen zu können.
Wenn der Assembler eine IF-Direktive findet, entschlüsselt er den numerischen Ausdruck, der dem IF folgt. Falls der numerische Ausdruck ungleich Null ist, werden die folgenden Quellzeilen übersetzt. Ergibt sich aber der Wert Null, werden die Quellzeilen nur ausgelistet, aber nicht übersetzt. Alle Elemente im numerischen Ausdruck müssen vorher definiert sein. Eine Verschachtelung von IF-Direktiven ist nicht erlaubt.

### 1.3.4. INCLUDE-Direktive

```
INCLUDE  filespec
```

Diese Direktive schließt eine andere ASM86-Datei in den Quelltext ein.

Beispiel:

```
INCLUDE  EQUALS.A86
```

Diese Direktive wird benutzt, wenn das Quellprogramm in mehreren verschiedenen Dateien existiert. INCLUDE-Direktiven dürfen nicht verschachtelt werden. Eine Quelldatei, die durch eine INCLUDE-Direktive gerufen wird, darf keine weitere INCLUDE-Direktive enthalten. Falls die Dateispezifikation keinen Dateityp enthält, wird A86 angenommen. Ist kein Laufwerk angegeben, wird das der Quelldatei angenommen.

### 1.3.5. END-Direktive

```
END
```

Die END-Direktive markiert das Ende einer Quelldatei. Alle nachfolgenden Zeilen werden vom Assembler ignoriert. Die END-Direktive ist frei wählbar. Ist sie nicht enthalten, bearbeitet der Assembler die Datei, bis er ein Dateiendezeichen (end of file) 1AH findet.

### 1.3.6. EQU-Direktive

```
symbol  EQU  numeric-expression
symbol  EQU  address-expression
symbol  EQU  register
symbol  EQU  instruction-mnemonic
```

Die EQU-Direktive weist Werte und Attribute den vom Nutzer definierten Symbolen zu. Das geforderte Symbol darf nicht mit einem Doppelpunkt (:) begrenzt werden. Es darf von einer folgenden EQU- oder anderen Direktive nicht wieder als Symbol benutzt werden. Alle Elemente, die im numerischen oder Adreßausdruck benutzt werden, müssen vorher definiert sein.
Die erste Form weist dem Symbol einen numerischen Wert zu, die zweite Form eine Adresse. Die dritte Form weist einem Register einen neuen Namen zu, und die vierte Form definiert einen neuen Befehl.

Beispiel:

```
FIVE     EQU     2*2+1
NEXT     EQU     BUFFER
COUNTER  EQU     CX
MOVVV    EQU     MOV
           .
           .
           .
         MOVVV   AX,BX
```

### 1.3.7. DB-Direktive

[symbol] DB {numeric-expression|string-constant}, ...

Die DB-Direktive definiert und initialisiert Speicherbereiche im Byteformat. Numerische Ausdrücke werden in 8-Bit-Werte verschlüsselt und nacheinander in der Code-Ausgabedatei abgespeichert. Zeichenkettenkonstanten werden in der Ausgabedatei in Übereinstimmung mit den im Abschnitt 1.2.4.2. definierten Regeln abgespeichert.
Die DB-Direktive ist die einzige ASM86-Anweisung, die Zeichenketten aus mehr als zwei Bytes akzeptiert. Innerhalb der Zeichenketten erfolgt keine Übersetzung von Klein- in Großbuchstaben.
Mehrere Ausdrücke oder Konstanten, getrennt durch Komma, dürfen der Definition hinzugefügt werden. Aber sie dürfen eine physische Zeilenlänge nicht überschreiten.
Das wählbare Symbol wird benötigt, um im gesamten Programm auf das Datenfeld zugreifen zu können. Das Symbol hat vier Attribute. Die Segment- und Offset-Attribute bestimmen die Speicherbezüge des Symbols. Das Typattribut bestimmt die einzelnen Bytes und die Länge gibt an, wieviel Bytes (Speichereinheiten) reserviert sind.

Beispiel:

```
TEXT    DB      'SCP SYSTEM',0
AA      DB      'A'+80H
X       DB      1,2,3,4,
          .
          .
          .
MOV     CX, LENGTH TEXT
```

### 1.3.8. DW-Direktive

[symbol] DW {numeric-expression|string-constant}, ...

Die DW-Direktive initialisiert Zwei-Byte-Worte im Speicher. Zeichenkettenkonstanten mit mehr als zwei Bytes sind unzulässig. Sonst initialisiert DW den Speicher in gleicher Weise wie DB, nur daß das niederwertige Byte zuerst gespeichert wird; danach folgt das höherwertige Byte.

Beispiel:

```
CNTR    DW  0
JMPTAB  DW  SUBR1,SUBR2,SUBR3
        DW  1,2,3,4,5,6,7,8
```

### 1.3.9. DD-Direktive

[symbol] DD address-expression[,address-expression...]

Die DD-Direktive initialisiert vier Bytes im Speicher. Das Offset-Attribut des Adreßausdrucks wird in den beiden niederen Bytes gespeichert, das Segment-Attribut in den beiden höheren Bytes.

Beispiel:

```
                CSEG  1234H
                 .
                 .
                 .
LONG_JMPTAB     DD   ROUT1,ROUT2
                DD   ROUT3,ROUT4
```

### 1.3.10. RS-Direktive

[symbol] RS numeric-expression

Die RS-Direktive reserviert Speicherplatz, aber initialisiert ihn nicht. Der numerische Ausdruck gibt die Byteanzahl an, die zu reservieren ist. Die RS-Direktive ergibt für das wahlfreie Symbol kein Byte-Attribut.

Beispiel:

```
BUFFER  RS  80
        RS  4000H
        RS  1
```

### 1.3.11. RB-Direktive

[symbol] RB numeric-expression

Die RB-Direktive reserviert Bytes im Speicher ohne Initialisierung.
Diese Direktive ist gleich der RS-Direktive, nur, daß sie dem Symbol ein Byte-Attribut gibt.

### 1.3.12. RW-Direktive

[symbol] RW numeric-expression

Die RW-Direktive reserviert Zwei-Byte-Worte im Speicher ohne Initialisierung. Der numerische Ausdruck gibt die Anzahl der zu reservierenden Bytes an.

Beispiel:

```
BUFF    RW  128
        RW  4000H
        RW  1
```

### 1.3.13. TITLE-Direktive

```
TITLE string-constant
```

ASM86 druckt die Zeichenkette, die mit der TITLE-Direktive definiert wird, als Kopfzeile auf jede Listenseite in der Listendatei. Die Zeichenkette darf 30 Zeichen nicht überschreiten.

Beispiel:

```
TITLE 'SCP 1700 SYSTEM'
```

### 1.3.14. PAGESIZE-Direktive

```
PAGESIZE numeric-expression
```

Mit der PAGESIZE-Direktive wird eine Zeilenanzahl pro Listenseite definiert. Der numerische Ausdruck gibt die Zeilenanzahl an. Standard sind 66 Zeilen pro Seite.

### 1.3.15. PAGEWIDTH-Direktive

```
PAGEWIDTH numeric-expression
```

Die PAGEWIDTH-Direktive definiert die Zahl der zu druckenden Spalten, wenn die Ausgabedatei gedruckt wird. Der Standardwert ist 120. Wenn die Ausgabe nicht auf dem Terminal erfolgt, ist der Standard 79. Als Spalte wird ein Zeichen einer Zeile verstanden.

### 1.3.16. EJECT-Direktive

```
EJECT
```

Die EJECT-Direktive führt zu einem Seitenvorschub während des Druckvorgangs. Die EJECT-Direktive selbst wird auf der ersten Zeile der nächsten Seite ausgegeben.

### 1.3.17. SIMFORM-Direktive

```
SIMFORM
```

Die SIMFORM-Direktive ersetzt ein FF (form-feed) in der auszudruckenden Datei durch eine korrekte Anzahl von LF (line-feed). Diese Direktive wird dann benutzt, wenn die Ausgabe auf einem Drucker erfolgt, der FF nicht ausführen kann.

### 1.3.18. NOLIST- und LIST-Direktive

```
NOLIST
LIST
```

Die NOLIST-Direktive unterdrückt die Ausgabe der folgenden Zeilen. Die Fortsetzung der Ausgabe bewirkt die LIST-Direktive.

## 1.4. ASM86-Befehlssatz

Der ASM86-Befehlssatz enthält alle Maschinenbefehle. Die allgemeine Syntax für Befehlsanweisungen wurde im Abschnitt 1.2.8. erläutert. Die folgenden Abschnitte behandeln die spezifische Syntax und die für jeden Befehl erforderlichen Operanden. Die Befehle werden in Tabellen leicht verständlich dargestellt.

Die Tabellen mit Befehlsdefinitionen zeigen die ASM86-Befehle als Kombination von Mnemoniks und Operanden. Ein Mnemonik ist eine symbolische Darstellung des Befehls und seine Operanden sind die geforderten Parameter. Befehle können ohne oder mit einem oder zwei Operanden versehen sein. Wenn zwei Operanden vorhanden sind, ist der linke Operand der Zieloperand (destination). Beide Operanden sind durch Komma getrennt.
Die Tabellen der ASM86-Befehle sind in funktionelle Gruppen unterteilt. Innerhalb der Tabellen sind die Befehle alphabetisch geordnet. Tabelle 8 zeigt die Symbole, die in den Tabellen zur Definition der Operanden benutzt werden.

Tabelle 8: Symbole für Operandentypen

| Symbol | Operandentyp |
|---|---|
| numb | beliebiger numerischer Ausdruck |
| numb8 | beliebiger numerischer Ausdruck, der einen 8-Bit-Wert ergibt |
| acc | Akkumulatorregister, AX oder AL |
| reg | allgemeines Register, kein Segmentregister |
| reg16 | allgemeines 16-Bit-Register, kein Segmentregister |
| segreg | beliebiges Segmentregister: CS, DS, SS oder ES |
| mem | beliebiger Adreßausdruck, mit oder ohne Basis- und/oder Index-Adressierungsmodi, z. B.:<br>VARIABLE<br>VARIABLE+3<br>VARIABLE[BX]<br>VARIABLE[SI]<br>VARIABLE[BX+SI]<br>[BX]<br>[BP+DI] |
| simpmem | beliebiger Adreßausdruck ohne Basis- und Index-Adressierungsmodi, z. B.:<br>VARIABLE<br>VARIABLE+4 |
| mem\|reg | beliebiger Ausdruck, symbolisiert durch reg\|mem |
| mem\|reg16 | beliebiger Ausdruck, symbolisiert durch mem\|reg, muß 16 Bits sein |
| label | beliebiger Adreßausdruck, der eine Marke ergibt |
| lab8 | beliebige Marke, die sich innerhalb einer Distanz von +/-128 Bytes von dem Befehl befindet |

Die ZVE besitzt 9 Einzelbit-Flagregister, die den Status der ZVE anzeigen. Der Anwender kann zu diesen Registern nicht direkt zugreifen, aber er kann sie im Ergebnis einer Operation austesten. Die Wirkung der Befehle auf die Flagregister wird in den Tabellen der Befehlsdefinitionen beschrieben. Dabei werden die in der Tabelle 9 dargestellten Symbole benutzt.

Tabelle 9: Flagregistersymbole

| Symbol | Bedeutung |
|---|---|
| AF | Auxiliary-Carry-Flag |
| CF | Carry-Flag |
| DF | Direction-Flag |
| IF | Interrupt-Enable-Flag |
| OF | Overflow-Flag |
| PF | Parity-Flag |
| SF | Sign-Flag |
| TF | Trap-Flag |
| ZF | Zero-Flag |

### 1.4.1. Datenübertragungsbefehle

Es gibt vier Klassen von Operationen zur Datenübertragung

- allgemeine Operationen
- akkumulatorspezifische Operationen
- Adreßoperationen
- Flagoperationen

Nur SAHF und POPF beeinflussen die Flags. In Tabelle 10 ist zu beachten, daß bei acc = AL ein Byte, aber bei acc = AX ein Wort übertragen wird.

Tabelle 10: Datenübertragungsbefehle

| Syntax | Ergebnis |
|---|---|
| IN acc,numb8 | Datenübertragung vom Eingabekanal zum Akkumulator, angegeben durch numb8 (0-255) |
| IN acc,DX | Datenübertragung vom Eingabekanal zum Akkumulator, angegeben durch DX (0-FFFFH) |
| LAHF | Übertragung der Flags zum AH-Register |
| LDS reg16,mem | Übertragung des Segmentteils der Speicheradresse (DWORD-Variable) zum DS-Segmentregister, Übertragung des Offsetteils zu einem allgemeinen Register (16 Bit) |
| LEA reg16,mem | Übertragung des Offsets einer Speicheradresse zu einem 16-Bit-Register |
| LES reg16,mem | Übertragung des Segmentteils der Speicheradresse zum ES-Segment-Register, Übertragung des Offsetteils zu einem allgemeinen Register (16 Bit) |
| MOV reg,mem\|reg | überträgt Speicher oder Register zum Register |
| MOV mem\|reg,reg | überträgt Register zum Speicher oder Register |
| MOV mem\|reg,numb | überträgt Direktdaten (immediate data) zum Speicher oder Register |
| MOV segreg,mem\|reg16 | überträgt Speicher oder Register zum Segmentregister |
| MOV mem\|reg16,segreg | überträgt Segmentregister zum Speicher oder Register |
| OUT numb8,acc | überträgt Daten vom Akkumulator zum Ausgabekanal, angegeben durch numb8 (0-255) |
| OUT DX,acc | überträgt Daten vom Akkumulator zum Ausgabekanal, angegeben durch DX (0FFFFH) |
| POP mem\|reg16 | überträgt oberes Stackelement zum Speicher oder Register |

Tabelle 10: (Fortsetzung)

| Syntax | Ergebnis |
|---|---|
| POP segreg | überträgt oberes Stackelement zum Segmentregister. Das CS-Segmentregister ist dabei nicht erlaubt. |
| POPF | überträgt oberes Stackelement zu den Flags |
| PUSH mem\|reg16 | legt Speicher oder Register als oberes Stackelement ab |
| PUSH segreg | legt Segmentregister als oberes Stackelement ab |
| PUSHF | legt Flags als oberes Stackelement ab |
| SAHF | überträgt das AH-Register zu den Flags |
| XCHG reg,mem\|reg | wechselt Register und Speicher oder Register |
| XCHG mem\|reg,reg | wechselt Speicher oder Register und Register |
| XLAT mem\|reg | führt eine Tabellenübertragung aus. Die Anfangsadresse der Tabelle muß in BX stehen. AL wird ersetzt durch den AL-Offset von BX. |

### 1.4.2. Arithmetische, logische und Verschiebebefehle

Die ZVE führt die vier mathematischen Grundoperationen auf verschiedenen Wegen aus. Sie unterstützt sowohl 8- als auch 16-Bit-Operationen und führt außerdem die Arithmetik mit und ohne Berücksichtigung des Vorzeichens aus.
Sechs der neun Flagbits werden im Ergebnis der meisten arithmetischen Operationen gelöscht oder gesetzt, um das Ergebnis der Operation anzuzeigen. Tabelle 11 faßt die Wirkungen der arithmetischen Befehle auf die Flagbits zusammen. Tabelle 12 definiert die arithmetischen Befehle und Tabelle 13 die logischen und Verschiebebefehle.

Tabelle 11: Wirkung der Arithmetikbefehle auf Flags

| Syntax | Ergebnis |
|---|---|
| CF | wird gesetzt, falls die Operation einen Übertrag (Addition) oder ein Borgen (Subtraktion) in das höchste Bit des Ergebnisses bewirkt; andernfalls ist CF gelöscht |
| AF | wird gesetzt, falls die Operation einen Übertrag (Addition) oder ein Borgen (Subtraktion) in das niedrigste Bit des Ergebnisses bewirkt; andernfalls ist AF gelöscht |
| ZF | wird gesetzt, wenn das Ergebnis der Operation Null ist; andernfalls bleibt ZF gelöscht |
| SF | wird gesetzt, wenn das Ergebnis negativ ist |
| PF | wird gesetzt, falls die Modulo-2-Summe der niedrigsten 8 Bits des Ergebnisses der Operation Null ist (gerade Parität); andernfalls wird PF gelöscht (ungerade Parität) |
| OF | wird gesetzt, wenn das Ergebnis der Operation einen Überlauf ergibt, wobei die Größe des Ergebnisses die Speicherkapazität der Zieladresse überschreitet |

Tabelle 12: Arithmetische Befehle

| Syntax | Ergebnis |
|---|---|
| AAA | Anpassung ungepackter BCD (KOI7) für Addition - AL-Anpassung (BCD = binär codierte Dezimalzahlen) |
| AAD | Anpassung ungepackter BCD (KOI7) für Division - AL-Anpassung |
| AAM | Anpassung ungepackter BCD (KOI7) für Multiplikation - AX-Anpassung |
| AAS | Anpassung ungepackter BCD (KOI7) für Subtraktion - AL-Anpassung |
| ADC reg,mem\|reg | Addition mit Übertrag (carry) Speicher oder Register zu Register |
| ADC mem\|reg,reg | Addition mit Übertrag (carry) Register zu Speicher oder Register |
| ADC mem\|reg,numb | Addition mit Übertrag von Direktdaten zu Speicher oder Register |
| ADD reg,mem\|reg | Addition Speicher oder Register zu Register |
| ADD mem\|reg,reg | Addition Register zu Speicher oder Register |
| ADD mem\|reg,numb | Addition von Direktdaten zu Speicher oder Register |
| CBW | Konvertierung eines Bytes in AL zu einem Wort in AH durch Vorzeichenexpansion |
| CWD | Konvertierung eines Wortes in AX zu einem Doppelwort in DX/AX durch Vorzeichenexpansion |
| CMP reg,mem\|reg | Vergleich Register mit Speicher oder Register |
| CMP mem\|reg,reg | Vergleich Speicher oder Register mit Register |
| CMP mem\|reg,numb | Vergleich Datenkonstante mit Speicher oder Register |
| DAA | Dezimalanpassung für Addition (AL) |
| DAS | Dezimalanpassung für Subtraktion (AL) |

Tabelle 12: (Fortsetzung)

| Syntax | Ergebnis |
|---|---|
| DEC mem\|reg | Subtraktion von 1 vom Speicher oder Register (Dekrement) |
| DIV mem\|reg | Division (ohne Vorzeichen) Akkumulator (AX oder AL) durch Speicher oder Register. Bei Byteergebnis enthält AL = Quotient und AH = Rest. Bei Wortergebnis ist: AX = Quotient und DX = Rest |
| IDIV mem\|reg | Division (mit Vorzeichen) Akkumulator (AX oder AL) durch Speicher oder Register. Quotient und Rest werden wie bei DIV abgespeichert. |
| IMUL mem\|reg | Multiplikation (mit Vorzeichen) Speicher oder Register mit Akkumulator (AX oder AL). Falls Byteergebnis, dann Ergebnis in AH und AL. Falls Wortergebnis, dann Ergebnis in DX und AX. |
| INC mem\|reg | Addition von 1 zu Speicher oder Register (Inkrement) |
| MUL mem\|reg | Multiplikation (ohne Vorzeichen) Speicher oder Register mit Akkumulator (AX oder AL), das Ergebnis wird wie bei IMUL gespeichert |
| NEG mem\|reg | Zweierkomplement von Speicher oder Register |
| SBB reg,mem\|reg | Subtraktion (mit Borgen) Speicher oder Register von Register |
| SBB mem\|reg,reg | Subtraktion (mit Borgen) Register von Speicher oder Register |
| SBB mem\|reg,numb | Subtraktion (mit Borgen) Direktdaten von Speicher oder Register |
| SUB reg,mem\|reg | Subtraktion Speicher oder Register vom Register |
| SUB mem\|reg,reg | Subtraktion Register vom Speicher oder Register |
| SUB mem\|reg,numb | Subtraktion Direktdaten vom Speicher oder Register |

Tabelle 13: Logische und Verschiebebefehle

| Syntax | Ergebnis |
|---|---|
| AND reg,mem\|reg | bitweises logisches UND von Register und Speicher oder Register |
| AND mem\|reg,reg | bitweises logisches UND von Speicher oder Register und Register |
| AND mem\|reg,numb | bitweises logisches UND von Speicher oder Register und Direktdaten |
| NOT mem\|reg | Einerkomplement von Speicher oder Register |
| OR reg,mem\|reg | bitweises logisches ODER von Register und Speicher oder Register |
| OR mem\|reg,reg | bitweises logisches ODER von Speicher oder Register und Register |
| OR mem\|reg,numb | bitweises logisches ODER von Speicher oder Register und Direktdaten |
| RCL mem\|reg,1 | zyklische Verschiebung Speicher oder Register um 1 Bit nach links durch das Übertragflag (C-Flag) |
| RCL mem\|reg,CL | zyklische Verschiebung Speicher oder Register nach links durch C-Flag, Verschiebezahl in CL-Register |
| RCR mem\|reg,1 | zyklische Rechtsverschiebung um 1 Bit durch C-Flag |
| RCR mem\|reg,CL | zyklische Rechtsverschiebung Speicher oder Register durch C-Flag, Verschiebezahl in CL-Register |
| ROL mem\|reg,1 | zyklische Linksverschiebung Speicher oder Register um 1 Bit |
| ROL mem\|reg,CL | zyklische Linksverschiebung Speicher oder Register, Verschiebezahl in CL-Register |
| ROR mem\|reg,1 | zyklische Rechtsverschiebung Speicher oder Register um 1 Bit |
| ROR mem\|reg,CL | zyklische Rechtsverschiebung Speicher oder Register, Verschiebezahl in CL-Register |

Tabelle 13: (Fortsetzung)

| Syntax | Ergebnis |
|---|---|
| SAL mem\|reg,1 | Linksverschiebung Speicher oder Register um 1 Bit; in niederes Bit wird Null eingetragen |
| SAL mem\|reg,CL | Linksverschiebung Speicher oder Register; Verschiebezahl in CL-Register; in niedere Bits werden Nullen aufgefüllt |
| SAR mem\|reg,1 | Rechtsverschiebung Speicher oder Register um 1 Bit und Nachschieben des ursprünglich höchsten Bits |
| SAR mem\|reg,CL | Rechtsverschiebung von Speicher oder Register; Verschiebezahl in CL-Register; Nachschieben des ursprünglich höchsten Bits |
| SHL mem\|reg,1 | Linksverschiebung Speicher oder Register um 1 Bit; Nachschieben von Null in niederes Bit (SHL=SAL) |
| SHL mem\|reg,CL | Linksverschiebung Speicher oder Register; Verschiebezahl in CL-Register; Nachschieben von Nullen in die niederen Bits (SHL=SAL) |
| SHR mem\|reg,1 | Rechtsverschiebung Speicher oder Register um 1 Bit und Nachschieben von Null ins höchste Bit |
| SHR mem\|reg,CL | Rechtsverschiebung Speicher oder Register; Verschiebezahl in CL-Register und Nachschieben von Nullen in die höheren Bits |
| TEST reg,mem\|reg | bitweises logisches UND von Register und Speicher oder Register; setzt die Flags, aber verändert Zieloperanden nicht |
| TEST mem\|reg,reg | bitweises logisches UND von Speicher oder Register und Register; setzt die Flags, aber verändert Zieloperanden nicht |
| TEST mem\|reg,numb | bitweises logisches UND von Speicher oder Register und Direktdaten; setzt die Flags, aber verändert Zieloperanden nicht |

Tabelle 13: (Fortsetzung)

| Syntax | Ergebnis |
|---|---|
| XOR reg,mem\|reg | bitweises logisches exklusives ODER von Register und Speicher oder Register |
| XOR mem\|reg,reg | bitweises logisches exklusives ODER von Speicher oder Register und Register |
| XOR mem\|reg,numb | bitweises logisches exklusives ODER von Speicher oder Register und Direktdaten |

### 1.4.3. Zeichenkettenbefehle

Zeichenkettenbefehle (string-instructions) haben keinen, einen oder zwei Operanden. Die Operanden geben nur den Operandentyp an, ob es eine Byte- oder Wortoperation ist. Wenn zwei Operanden angegeben sind, wird der Quelloperand durch das SI-Register und der Zieloperand durch das DI-Register adressiert. Die DI- und SI-Register werden immer für die Adressierung benutzt. Es ist zu beachten, daß bei Zeichenkettenbefehlen der durch DI adressierte Zieloperand sich immer im Extra-Segment (ES) befinden muß.
Der Quelloperand wird im allgemeinen durch das DS-Register adressiert. Jedoch kann durch die Benutzung eines Segment-Override-Präfixes ein anderes Register festgelegt werden.

Beispiel:

```
MOVS  WORD PTR[DI],CS:WORD PTR[SI]
```

Tabelle 14: Zeichenkettenbefehle

| Syntax | Ergebnis |
|---|---|
| CMPS mem\|reg,mem\|reg | Subtraktion der Quelle vom Ziel und Setzen der Flags; die Operanden werden nicht verändert |
| CMPSB | CMPS mit Byteoperand |
| CMPSW | CMPS mit Wortoperand |
| LODS mem\|reg | Übertragung eines Bytes oder Wortes vom Quelloperanden zum Akkumulator |
| LODSB | LODS mit Byteoperand |
| LODSW | LODS mit Wortoperand |
| MOVS mem\|reg,mem\|reg | Übertragung eines Bytes oder Wortes von der Quelle zum Ziel |
| MOVSB | MOVS mit Byteoperand |
| MOVSW | MOVS mit Wortoperand |
| SCAS mem\|reg | Subtraktion des Zieloperanden vom Akkumulator (AX oder AL); Setzen der Flags, aber Rückkehr ohne Ergebnis |
| SCASB | SCAS mit Byteoperand |
| SCASW | SCAS mit Wortoperand |
| STOS mem\|reg | Übertragung eines Bytes oder Wortes vom Akkumulator zum Zieloperanden |
| STOSB | STOS mit Byteoperand |
| STOSW | STOS mit Wortoperand |

Tabelle 15 definiert die Präfixe für Zeichenkettenbefehle. Ein Präfix wiederholt seinen Zeichenkettenbefehl entsprechend der Anzahl im CX-Register, dessen Inhalt nach jeder Iteration um 1 erniedrigt wird. Präfixmnemoniks stehen vor dem Zeichenkettenbefehlsmnemoniks der Anweisungszeile.

Tabelle 15: Präfixbefehle

| Syntax | Ergebnis |
|---|---|
| REP | Wiederholung bis CX-Register Null ist |
| REPE | Wiederholung bis CX-Register Null und das Z-Flag ungleich Null ist |
| REPNE | Wiederholung bis CX-Register Null und das Z-Flag Null ist |
| REPNZ | siehe REPNE |
| REPZ | siehe REPE |

### 1.4.4. Steuerübertragungsbefehle

Es gibt vier Klassen von Steuerübertragungsbefehlen:

- Rufe, Sprünge und Rücksprünge
- bedingte Sprünge
- Wiederholungssteuerung
- Unterbrechungen (interrupts)

Alle Steuerübertragungsbefehle veranlassen die Programmfortsetzung ab einem bestimmten neuen Platz im Speicher, möglicherweise in einem neuen Code-Segment. Die Übertragung der Steuerung kann absolut oder in Abhängigkeit einer bestimmten Bedingung erfolgen. Tabelle 16 definiert die Steuerübertragungsbefehle. In den Definitionen bedingter Sprünge, vorwärts und rückwärts, handelt es sich um vorzeichenlose Werte. Größer als und kleiner bezieht sich auf Werte mit Vorzeichen.

Tabelle 16: Steuerübertragungsbefehle

| Syntax | | Ergebnis |
|---|---|---|
| CALL | label | Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack und Sprung zur Zielmarke (label) |
| CALL | mem\|reg16 | Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack und Sprung zum Speicherplatz, der durch den Inhalt des Speicherplatzes bzw. Registers angezeigt ist |
| CALLF | label | Ablegen des CS-Registerinhaltes auf dem Stack; Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack (nach CS); Sprung zur Zielmarke |
| CALLF | mem | Ablegen des CS-Registerinhaltes auf dem Stack; Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack und Sprung zum Speicherplatz, der durch den Inhalt des Doppelwortes im Speicher angegeben ist |
| INT | numb8 | Ablegen des Flag-Registers (wie PUSHF) auf dem Stack; Löschen der TF- und IF-Flags und Übertragung der Steuerung mit einem Indirektruf über einen der 256 Unterbrechungsvektorelemente (benutzt drei Ebenen des Stacks) |
| INTO | | Falls das OF-Flag (Overflow-Flag) gesetzt ist, werden die Flagregister (wie PUSHF) auf dem Stack abgelegt. TF und IF werden gelöscht und die Steuerung mit einem Indirektruf über Unterbrechungsvektorelement 4 (Platz 10H) übertragen. Falls OF gelöscht ist, wird keine Operation ausgeführt. |
| IRET | | Übertragung der Steuerung zur Rückkehradresse, die durch die vorhergehende Unterbrechungsoperation gerettet wurde. Gerettete Flagregister werden zurückgespeichert, ebenso CS und IP (gibt drei Ebenen des Stacks frei). |

Tabelle 16: (Fortsetzung)

| Syntax | | Ergebnis |
|---|---|---|
| JA | lab8 | Sprung, falls "nicht darunter oder gleich" oder "darüber" ((CF oder ZF) = 0) |
| JAE | lab8 | Sprung, falls "nicht darunter" oder "darüber oder gleich" (CF = 0) |
| JB | lab8 | Sprung, falls "darunter" oder "nicht darüber oder gleich (CF = 1) |
| JBE | lab8 | Sprung, falls "darunter oder gleich" oder "nicht darüber" ((CF oder ZF) = 1) |
| JC | lab8 | siehe JB |
| JCXZ | lab8 | Sprung zur Zielmarke, wenn CX-Register gleich Null |
| JE | lab8 | Sprung, falls "gleich" oder "Null" (ZF = 1) |
| JG | lab8 | Sprung, falls "nicht kleiner oder gleich" oder "größer" (((SF xor OF) or ZF) = 0) |
| JGE | lab8 | Sprung, falls "nicht kleiner" oder "größer oder gleich" ((SF xor CF) = 0) |
| JL | lab8 | Sprung, falls "kleiner" oder "nicht größer oder gleich" ((SF xor OF) = 1) |
| JLE | lab8 | Sprung, falls "kleiner oder gleich" oder "nicht größer" (((SF xor OF) or ZF) = 1) |
| JMP | label | Sprung zur Zielmarke |
| JMP | mem\|reg16 | Sprung zum Speicherplatz, der durch den Inhalt von Speicher oder Register angegeben ist |
| JMPF | label | Sprung zur Zielmarke, möglicherweise in einem anderen Code-Segment |
| JMPS | lab8 | Sprung zur Zielmarke innerhalb einer Distanz von +/-128 Bytes von dem Befehl |

Tabelle 16: (Fortsetzung)

| Syntax | | Ergebnis |
|---|---|---|
| JNA | lab8 | siehe JBE |
| JNAE | lab8 | siehe JB |
| JNB | lab8 | siehe JAE |
| JNBE | lab8 | siehe JA |
| JNC | lab8 | siehe JAE |
| JNE | lab8 | Sprung, falls "nicht gleich" oder "nicht Null" (ZF = 0) |
| JNG | lab8 | siehe JLE |
| JNGE | lab8 | siehe JL |
| JNL | lab8 | siehe JGE |
| JNLE | lab8 | siehe JG |
| JNO | lab8 | Sprung, falls "kein Überlauf" (OF = 0) |
| JNP | lab8 | Sprung, falls "keine Parität" oder "ungerade Parität" (PF = 0) |
| JNS | lab8 | Sprung, falls "kein Vorzeichen" (SF = 0) |
| JNZ | lab8 | siehe JNE |
| JO | lab8 | Sprung, falls "Überlauf" (OF = 1) |
| JP | lab8 | Sprung, falls "Parität" oder "gerade Parität" (PF = 1) |
| JPE | lab8 | siehe JP |
| JPO | lab8 | siehe JNP |
| JS | lab8 | Sprung, falls "Vorzeichen" (SF = 1) |
| JZ | lab8 | siehe JE |
| LOOP | lab8 | Dekrement von CX um 1 und Sprung zur Zielmarke, wenn CX ungleich Null |
| LOOPE | lab8 | Dekrement von CX um 1 und Sprung zur Zielmarke, falls CX ungleich Null und das ZF-Flag gesetzt ist - "Schleife, solange Null" oder "Schleife, solange gleich" |

Tabelle 16: (Fortsetzung)

| Syntax | Ergebnis |
|---|---|
| LOOPNE lab8 | Dekrement von CX um 1 und Sprung zur Zielmarke, falls CX ungleich Null und ZF = 0 - "Schleife, solange ungleich Null" oder "Schleife, solange nicht gleich" |
| LOOPNZ lab8 | siehe LOOPNE |
| LOOPZ lab8 | siehe LOOPE |
| RET | Rücksprung zur Rückkehradresse, die vom vorhergehenden CALL auf dem Stack abgelegt wurde; Weiterzählen des Stackpointers um 2 |
| RET numb | siehe RET, aber Weiterzählen des Stackpointers um 2 + numb |
| RETF | Rücksprung zur Adresse, die von CALLF auf dem Stack abgelegt wurde; Weiterzählen des Stackpointers um 4 |
| RETF numb | Rücksprung zur Adresse, die vom letzten CALLF auf dem Stack abgelegt wurde; Weiterzählen des Stackpointers um 4 + numb |

### 1.4.5. Prozessorsteuerbefehle

Prozessorsteuerbefehle beeinflussen die Flagregister. Darüber hinaus können einige von diesen Befehlen die ZVE mit externer Gerätetechnik synchronisieren.

Tabelle 17: Prozessorsteuerbefehle

| Syntax | Ergebnis |
|---|---|
| CLC | Löschen des CF-Flags |
| CLD | Löschen des DF-Flags; veranlaßt, daß bei Zeichenkettenbefehlen das Operandenregister selbständig erhöht wird |
| CLI | Löschen des IF-Flags; damit sind externe Unterbrechungen nicht maskierbar |
| CMC | Komplement des CF-Flag |
| ESC numb8,mem\|reg | keine Operation, außer, daß die effektive Adresse berechnet und auf den Adreßbus gebracht wird;<br>(ESC wird vom Numeric-CO-Prozessor benutzt.)<br>numb8 muß im Bereich von 0 ... 63 liegen. |
| HLT | veranlaßt den Prozessor, in den Haltzustand zu gehen, bis eine Unterbrechung erkannt wird |
| LOCK | Präfix-Befehl, der den Prozessor veranlaßt, für die Dauer der Operation, die dem LOCK folgt, den Bus zu sperren (bus-lock). Der LOCK-Präfix darf jedem anderen Befehl vorangesetzt werden. Bus-lock hindert Co-Prozessoren, den Bus zu benutzen. |
| NOP | keine Operation |
| STC | Setzen des CF-Flags |
| STD | setzt das DF-Flag; veranlaßt, daß bei Zeichenkettenbefehlen das Operandenregister selbständig erniedrigt wird |
| STI | Setzen des IF-Flags; externe Unterbrechungen werden maskierbar |
| WAIT | veranlaßt den Prozessor, in den Wartezustand zu gehen, falls das Signal auf seinen Test-Pin nicht behauptet wird |

## 1.5. Code-Makro-Möglichkeiten

### 1.5.1. Einführung

Mit ASM86 hat der Anwender die Möglichkeit, eigene Befehle oder Befehlsfolgen zu definieren. Das erfolgt mit Hilfe der Code-Makros. Diese haben allerdings nur eine geringe Ähnlichkeit mit den herkömmlichen Assembler-Makros, die gewöhnlich beliebige Assemblerbefehle enthalten können.
Die Code-Makros von ASM86 enthalten nur Code-Makro-Direktiven. Während die herkömmlichen Assembler-Makros in der Symboltabelle des Assemblers geführt werden, sind die Code-Makros in einer anderen (internen) Symboltabelle definiert.
Mit der herkömmlichen Makrotechnik kann die wiederholte Anwendung gleicher Befehlsblöcke im gesamten Programm vereinfacht werden. Code-Makros generieren dagegen nur eine Bitfolge in die Objektdatei. So sind die gesamten Maschinenbefehle über Code-Makros definiert und weitere neue Befehle oder Befehlsfolgen können zusätzlich entstehen.
ASM86 behandelt einen Code-Makro wie einen Befehl. Ein Code-Makro wird auch wie ein Befehl aufgerufen.
Das folgende Beispiel zeigt, wie der Code-Makro MYCODE aufgerufen wird. MYCODE ist ein durch einen Code-Makro definierter Befehl.

Beispiel:

```
        .
        .
        .
XCHG    BX,WORD3
MYCODE  PARM1,PARM2
MUL     AX,WORD4
        .
        .
        .
```

Der Aufruf hat zwei Operanden, welche die aktuellen Parameter des Code-Makros sind. Aktuelle Parameter sind immer Symbole, Konstanten oder Register. Bei der Definition des Code-Makros werden zwei Operanden als formale Parameter (Platzhalter) benutzt. ASM86 klassifiziert diese zwei Operanden in Typ, Größe usw. Die Namen der formalen Parameter sind nicht festgelegt, sie sind frei wählbar. Beim Aufruf der Code-Makros ersetzt ASM86 die formalen Parameter durch die aktuellen Parameter; formale Parameter sind nur Platzhalter. Sie zeigen an, wo und wie die Operanden zu verwenden sind.
Ein Code-Makro hat die allgemeine Form:

```
CODEMACRO name [formal-parameter,...]
    code-macro-body
ENDM
```

Ein Code-Makro wird durch das Schlüsselwort CODEMACRO eingeleitet. Der Name ist ein Bezeichner und identifiziert den Code-Makro.
Formale Parameter haben die allgemeine Form:

```
formal-name:specifier-letter[modifier-letter][range]
```

Ein formaler Parameter besteht aus dem

- formalen Namen (formal-name) und

- Attributbuchstaben (specifier-letter).

Der Modifikationsbuchstabe (modifier-letter) und die Bereichsattribute (range) sind wahlfrei anzugeben.
Mögliche Spezifikationsbuchstaben sind:
A, C, D, E, M, R, S und X.
Mögliche Modifikationsbuchstaben sind: b, d, w und sb.
Innerhalb der Code-Makros ist die Groß- und Kleinschreibung beliebig. Zur besseren Übersicht werden in diesem Abschnitt

- Spezifikatoren (specifier-letter) in Großbuchstaben und

- Modifikatoren (modifier-letter) in Kleinbuchstaben angegeben.

In den folgenden Abschnitten wird die Liste der formalen Parameter (Spezifikatoren, Modifikatoren, Bereichsangaben) im einzelnen beschrieben.
Der Körper eines Code-Makros beschreibt die Bitmuster und formalen Parameter. Nur die folgenden Direktiven sind in Code-Makros zulässig:

```
SEGFIX      RELB    DB
NOSEGFIX    RELW    DW
MODRM       DBIT    DD
```

Die Code-Makro-Direktiven DB, DW und DD sind zwar auch als ASM86-Direktiven vorhanden, haben aber in Code-Makros eine andere Bedeutung. Diese Anweisungen werden in Abschnitt 1.5.4.5. genauer erläutert.
CODEMACRO, ENDM und die Code-Makro-Direktiven sind reservierte Worte. Die formale Definition der Syntax eines Code-Makros wird in der Syntaxbeschreibung in der Anlage 6 definiert.
Die folgenden Beispiele sind typische Code-Makro-Definitionen:

Beispiele:

```
CODEMACRO   AAA
   DB  37H
ENDM


CODEMACRO   DIV DIVISOR:Eb
   SEGFIX   DIVISOR
   DB       0F6H
   MODRM    6,DIVISOR
ENDM


CODEMACRO   ESC OPCODE:Db(0,63),SRC:Eb
   SEGFIX   SRC
   DBIT     5(1BH),3(OPCODE(3))
   MODRM    OPCODE,SRC
ENDM
```

### 1.5.2. Attribute

Jeder formale Parameter muß einen Attributbuchstaben (specifier-letter) haben, der den benötigten Operandentyp des formalen Parameters angibt. Dieser muß beim Aufruf mit dem Typ des aktuellen Parameters übereinstimmen.
Tabelle 18 definiert die 8 möglichen Attribute.

Tabelle 18: Operandenattribute von Code-Makros

| Buchstabe | Operandentyp |
|---|---|
| A | Akkumulatorregister AX oder AL |
| C | Code, nur ein Markenausdruck |
| D | Daten, eine Zahl, die wie ein Direktwert genutzt wird |
| E | effektive Adresse, entweder M (Speicheradresse) oder R (Register) |
| M | Speicheradresse, die entweder eine Variable oder ein geklammerter Registerausdruck sein kann |
| R | nur ein allgemeines Register |
| S | nur ein Segmentregister |
| X | direkter Speicherbezug |

### 1.5.3. Modifikatoren

Der Modifikationsbuchstabe (modifier-letter) ist eine weitere, wahlfreie Anforderung an den Operanden. Die Bedeutung des Modifikationsbuchstabens hängt vom Typ des Operanden ab. Für Variable muß der Modifikator folgenden Typ besitzen:

b für Byte
w für Wort
d für Doppelwort
sb für Byte mit mit Vorzeichen

Für Zahlen fordern die Modifikatoren eine bestimmte Größe:

b von -256 bis 255 und
w für weitere Zahlen

Tabelle 19 faßt die Code-Makro-Modifikatoren zusammen.

Tabelle 19: Operandenmodifikatoren von Code-Makros

| Variable | | Zahlen | |
|---|---|---|---|
| Modifikator | Typ | Modifikator | Größe |
| b | Byte | b | -256 bis 255 |
| w | Wort | w | alles andere |
| d | D-Wort | | |
| sb | Byte mit Vorzeichen | | |

### 1.5.4. Bereichsattribute

Der wahlfreie Bereich (range) wird entweder durch einen oder durch zwei Ausdrücke angegeben, die durch Komma getrennt und in runde Klammern eingeschlossen sind. Folgende Formate sind zulässig:

```
(numberb)
(register)
(numberb,numberb)
(numberb,register)
(register,numberb)
(register,register)
```

numberb ist eine 8-Bit-Zahl, keine Adresse. Das folgende Beispiel gibt an, daß der Eingabekanal (input-port) durch das DX-Register spezifiziert werden muß.

```
CODEMACRO  IN  DST:Aw,PORT:Rw(DX)
```

Im nächsten Beispiel wird angegeben, daß das Register CL den Zähler (COUNT) der zyklischen Verschiebung (rotation) enthalten muß:

```
CODEMACRO  ROR  DST:Ew,COUNT:Rb(CL)
```

Im letzten Beispiel wird gezeigt, daß der formale Parameter (OPCODE) Direktdaten darstellt, die im Bereich von 0 bis 63 liegen können:

```
CODEMACRO  ESC  OPCODE:Db(0,63),ADDS:Eb
```

### 1.5.5. Code-Makro-Direktiven

Code-Makro-Direktiven definieren die Bitmuster des generierten Codes und bestimmen, wie die Operanden zu behandeln sind. Direktiven sind reservierte Worte. Diejenigen Direktiven, die auch in der Assemblersprache vorkommen (DB, DW, DD) haben innerhalb von Code-Makros eine andere Bedeutung.
Nur die neun hier definierten Direktiven sind innerhalb von Code-Makro-Definitionen zulässig.

#### 1.5.5.1. SEGFIX

SEGFIX veranlaßt ASM86 zu entscheiden, ob ein Segment-Override-Präfix-Byte für den Zugriff zu einem gegebenen Speicherplatz nötig ist. Wenn es so ist, wird es als erstes Byte des Befehls angegeben, andernfalls erfolgt durch ASM86 keine Reaktion.
SEGFIX hat die Form:

```
SEGFIX  formal-name
```

wobei formal-name der Name eines formalen Parameters ist, der die Speicheradresse repräsentiert. Da der formale Parameter eine Speicheradresse repräsentiert, muß er eines der Attribute E, M, oder X haben.

#### 1.5.5.2. NOSEGFIX

NOSEGFIX wird für Operanden in Befehlen benutzt, die das ES-Register für diesen Operanden benutzen müssen. Diese Anwendung beschränkt sich nur auf den Zieloperanden folgender Befehle: CMPS, MOVS, SCAS, STOS.
NOSEGFIX hat die Form:

```
NOSEGFIX  segreg,formal-name
```

wobei segreg eines der Segmentregister ES, CS, SS oder DS ist, und formal-name ist der Name der Speicheradresse als formaler Parameter. Dieser muß ein Attribut E, M oder X haben. Von dieser Direktive wird kein Code generiert, aber ein Fehlertest wird ausgeführt.

Beispiel:

```
CODEMACRO  MOVS   SI_PTR:Ew,DI_PTR:Ew
   NOSEGFIX       ES,DI_PTR
   SEGFIX         SI_PTR
   DB             0A5H
ENDM
```

#### 1.5.5.3. MODRM

Diese Direktive veranlaßt ASM86, das MODRM-Byte zu generieren, das dem Operationscode-Byte (opcode-byte) in vielen Befehlen folgt. Das MODRM-Byte enthält entweder den Indextyp oder die Registernummer, die im Befehl benutzt wird. Es gibt außerdem an, welches Register zu benutzen ist oder gibt mehr Informationen zur Spezifizierung eines Befehls.
Das MODRM-Byte enthält die Informationen in drei Feldern:

Das Modusfeld (mod) belegt die zwei höchsten Bits des Bytes und bildet, verbunden mit dem Register/Speicherfeld (r/m), 32 mögliche Werte: 8 Register und 24 Indexierungsmodi.

Das Registerfeld (reg) belegt die drei nächsten Bits nach dem Modusfeld. Es gibt entweder eine Registerzahl oder drei weitere Bits der Operationscode-Information an. Die Bedeutung des Registerfeldes ist durch das Operationscode-Byte bestimmt.

Das Register/Speicherfeld (r/m) belegt die letzten drei Bits im Byte. Es spezifiziert ein Register als einen Platz eines Operanden oder bildet einen Teil des Adressierungsmodus in Verbindung mit dem oben beschriebenen Modusfeld.
MODRM hat die Formen:

```
MODRM  formal-name,formal-name
MODRM  NUMBER7,formal-name
```

wobei NUMBER7 ein Wert von 0 bis 7 ist und formal-name ist der Name des formalen Parameters. Die folgenden Beispiele zeigen die Anwendung von MODRM:

Beispiele:

```
CODEMACRO  RCR DST:Ew,COUNT:Rb(CL)
   SEGFIX       DST
   DB           0D3H
   MODRM        3,DST
ENDM

CODEMACRO  OR DST:Rw,SRC:Ew
   SEGFIX       SRC
   DB           0BH
   MODRM        DST,SRC
ENDM
```

### 1.5.5.4. RELB und RELW

Diese Direktiven werden für Sprungbefehle verwendet. ASM86 generiert einen Abstand (displacement) zwischen dem Ende des Befehls und der Marke, die als Operand angegeben ist. RELB generiert ein Byte und RELW zwei Bytes für den Abstand.
Die Direktiven haben folgende Form:

```
RELB  formal-name
RELW  formal-name
```

wobei formal-name der Name des formalen Parameters mit einem C(Code)-Attribut ist.

Beispiel:

```
CODEMACRO  LOOP  PLACE:Cb
   DB            0E2H
   RELB          PLACE
ENDM
```

### 1.5.5.5. DB, DW und DD

Diese Direktiven unterscheiden sich von denjenigen, die außerhalb der Code-Makros angewendet werden.
Die Direktiven haben folgende Formen:

```
DB  formal-name | NUMBERB
DW  formal-name | NUMBERW
DD  formal-name
```

wobei NUMBERB eine Ein-Byte-Zahl ist. NUMBERW ist eine Zwei-Byte-Zahl und formal-name ist der Name des formalen Parameters.

Beispiel:

```
CODEMACRO  XOR  DST:Ew,SRC:Db
   SEGFLX        DST
   DB            81H
   MODRM         6,DST
   DW            SRC
ENDM
```

### 1.5.5.6. DBIT

Diese Direktive behandelt Bits innerhalb eines Bytes oder einer kleineren Speichereinheit.
Die Form ist:

```
DBIT  field-description,...
```

wobei field-description zwei Formen haben kann:

```
number combination
number (formal-name(rshift))
```

wobei number im Bereich von 1 bis 16 liegt und die Zahl der zu setzenden Bits darstellt. Die gewünschte Bitkombination wird durch combination angegeben. Die Gesamtheit aller numbers, die in der field-description aufgelistet sind, darf 16 nicht überschreiten.
Die zweite darunter gezeigte Form enthält einen formalen Parameternamen (formal-name), der den Assembler veranlaßt, eine bestimmte Zahl auf eine bestimmte Position zu legen. Diese Zahl bezieht sich normalerweise auf das in der ersten Zeile des Code-Makros angegebene Register. Die Zahlen, die in diesem speziellen Fall für jedes Register benutzt werden, sind:

```
AL: 0   AH: 4   AX: 0   SP: 4   ES: 0
CL: 1   CH: 5   CX: 1   BP: 5   CS: 1
DL: 2   DH: 6   DX: 2   SI: 6   SS: 2
BL: 3   BH: 7   BX: 3   DI: 7   DS: 3
```

Der Parameter rshift in der innersten Klammer gibt eine Zahl von Rechtsverschiebungen an.

```
Beispiel: 0  keine Verschiebung
          1  verschiebt um 1 Bit nach rechts
          2  verschiebt 2 Bit nach rechts usw.
```

Die folgende Definition benutzt diese Form:

```
CODEMACRO   DEC   DST:Rw
   DBIT   5(9H),3(DST(0))
ENDM
```

Die ersten fünf Bit des Bytes haben den Wert 9H. Falls die Restbits Null sind, wird der hexadezimale Wert des Bytes 48H.
Wenn der Befehl

```
     DEC   DX
```

übersetzt wird und DX hat den Wert 2H, dann ist 48H + 2H = 4AH, was den Endwert des Bytes zur Ausführung darstellt.
Ist aber folgende Definition gegeben:

```
     DBIT   5(9H),3(DST(1))
```

dann würde die Registerzahl einmal nach rechts verschoben werden und das Ergebnis würde 48H + 1H = 49H sein, was falsch ist.

# 2. Debugger DDT86

## 2.1. Einführung

DDT86 (Dynamic Debugging Tool) ermöglicht dem Anwender, Programme unter SCP 1700 interaktiv zu testen und Fehler zu beseitigen. Der Leser dieses Abschnitts sollte mit dem 1700-Prozessor, dem Betriebssystem SCP 1700 und dem Assembler ASM86 vertraut sein.

### 2.1.1. Start

Der Start des DDT86 erfolgt durch Eingabe eines der folgenden Kommandos:

```
DDT86
DDT86 <filespec>
```

Das erste Kommando lädt DDT86 und startet es. Nach der Meldung und Ausgabe der Kommandoanforderung (-) ist DDT86 zur Aufnahme von Bedienerkommandos bereit.
Das zweite Kommando ist ähnlich dem ersten, nur lädt DDT86 noch die durch <filespec> spezifizierte Datei, nachdem es selbst geladen und gestartet wurde. Falls der Dateityp der Dateispezifikation <filespec> fehlt, wird CMD angenommen. DDT86 kann keine Datei vom Typ H86 laden. Das zweite Kommando entspricht der Folge:

```
A>DDT86
DDT86 V 1.0
-E<filespec>
```

An diesem Punkt ist das geladene Programm zur Ausführung bereit.

### 2.1.2. Kommandoformat

DDT86 gibt als Aufforderung zur Kommandoeingabe einen Bindestrich (-) aus. Der Bediener kann als Antwort eine Kommandozeile eingeben oder mit CTRL/C den Test beenden (siehe Abschnitt 2.1.4.).
Eine Kommandozeile kann bis zu 64 Zeichen lang sein und muss mit RETURN abgeschlossen werden.
Während der Kommandoeingabe sind die Editierfunktionen (CTRL/X, CTRL/H, CTRL/R usw.) zur Korrektur falscher Eingaben nutzbar.
DDT86 bearbeitet eine Kommandozeile erst nach Eingabe von RETURN.
Das erste Zeichen einer Kommandozeile bestimmt das Kommando. Die einzelnen Kommandos sind in Abschnitt 2.2. beschrieben.
Tabelle 20 faßt die Kommandos des DDT86 zusammen.

Tabelle 20: DDT86-Kommandos

| Kommando | Wirkung |
|---|---|
| A | Eingabe von Assembleranweisungen |
| B | Vergleich zweier Speicherblöcke |
| D | Anzeige des Speicherinhaltes hexadezimal und KOI7 |
| E | Laden eines Programms zur Ausführung |
| F | Füllen eines Speicherblocks mit einer Konstanten |
| G | Testbeginn mit wahlweisen Unterbrechungspunkten |
| H | Hexadezimale Arithmetik |
| I | Aufstellen von Standard-FCB und Kommandoparameter |
| L | Auflisten des Speicherinhaltes im Mnemonikformat |
| M | Umspeichern eines Speicherblockinhaltes |
| R | Lesen einer Plattendatei in den Speicher |
| S | Prüfen und Ändern des Speicherinhaltes |
| T | Programmabarbeitung mit Protokoll |
| U | Programmabarbeitung ohne Protokoll |
| V | Speicherbelegung der gelesenen Datei |
| W | Schreiben eines Speicherblockinhalts auf die Platte |
| X | Prüfen und Ändern des ZVE-Status |

Dem Kommandozeichen können ein oder mehrere Argumente folgen. Das können hexadezimale Werte, Dateinamen oder andere Informationen in Abhängigkeit vom jeweiligen Kommando sein.
Die Argumente untereinander werden mit Komma oder Leerzeichen getrennt. Zwischen dem Kommandozeichen und dem ersten Argument ist kein Leerzeichen erlaubt.

### 2.1.3. Spezifizierung einer 20-Bit-Adresse

Die meisten DDT86-Kommandos fordern als Operanden eine oder mehrere Adressen. Da der WM86-Prozessor Adressen bis zu 1-MByte-Speicher adressieren kann, werden 20 Bit für die Adressierung benötigt.

Die Eingabe einer 20-Bit-Adresse geschieht wie folgt:

```
ssss:oooo
```

Es bedeuten:

ssss: 16-Bit-Segmentnummer (wahlweise)
oooo 16-Bit-Offset

DDT86 vereinigt diese beiden Werte zu einer effektiven Adresse von 20 Bit nach folgender Rechnung:

```
  ssss0
+ oooo
------
  eeeee
```

Der Wert von ssss (Segmentnummer) kann ein hexadezimaler 16-Bit-Wert oder der Name eines Segmentregisters sein, oder er kann entfallen. Falls ein Segmentregister spezifiziert wird, ist der

Wert von ssss der Inhalt des Registers im ZVE-Status des Nutzers, wie er mit dem X-Kommando ausgegeben werden kann. Falls ssss weggelassen wurde, nimmt DDT86 dafür einen dem Kommando entsprechenden Standardwert an (siehe Abschnitt 2.3.).

### 2.1.4. Beenden

Die Arbeit mit DDT86 wird durch Eingabe von CTRL/C nach der Kommandoanforderung beendet. Die Steuerung geht an die Exekutive zurück. Falls DDT86 zur Korrektur einer Datei benutzt wurde, sollte diese vorher mit dem W-Kommando auf die Diskette geschrieben werden, bevor DDT86 beendet wird.

### 2.1.5. Unterbrechungsbehandlung

DDT86 arbeitet mit Unterbrechungen (interrupts), die erlaubt oder nicht erlaubt sein können. DDT86 sichert den Unterbrechungsstatus des unter DDT86 ausgeführten Programms.
Wenn DDT86 die Steuerung der ZVE hat, d.h. wenn DDT86 gestartet wurde oder wenn es die Steuerung vom Testprogramm zurückerhält, ist die Bedingung des Unterbrechungsflags die gleiche wie die zum Zeitpunkt des DDT86-Starts. Ausgenommen davon sind einige kritische Regionen, wo Unterbrechungen nicht erlaubt sind.
Während das zu testende Programm die ZVE-Steuerung hat, bestimmt der ZVE-Nutzerstatus, der mittels X-Kommando angezeigt werden kann, den Status des Unterbrechungsflags.

## 2.2. Beschreibung der DDT86-Kommandos

### 2.2.1. Kommando A (Assemble)

Das A-Kommando assembliert Maschinenbefehle direkt in den Speicher. Die Kommandoform ist:

```
As
```

s ist die 20-Bit-Adresse, von der die Eingabe des Assemblercodes beginnen soll. DDT86 antwortet auf dieses Kommando mit der Anzeige der Adresse, ab der assembliert werden soll. Danach gibt der Bediener entsprechend der Assemblersyntax Befehle ein (siehe Abschnitt 2.4.).
DDT86 konvertiert die Eingabe in den Maschinencode, trägt die Werte in den Speicher ein und zeigt die Adresse des nächsten verfügbaren Speicherplatzes an. Dieser Prozeß wird so lange fortgeführt, bis der Bediener eine Leerzeile oder eine Zeile eingibt, die nur einen Punkt (.) enthält.
Auf eine falsche Eingabe antwortet DDT86 mit einem Fragezeichen (?) und wiederholt die Anzeige der aktuellen Adresse.

### 2.2.2. Kommando B (Block Compare)

Das B-Kommando (block compare) vergleicht zwei Speicherblöcke und zeigt alle Differenzen auf dem Bildschirm an. Die Kommandoform ist:

```
Bs1,f1,s2
```

Es bedeuten:

s1 20-Bit-Adresse vom Beginn des ersten Blocks
f1 Offset des letzten Bytes im ersten Block
s2 20-Bit-Adressen vom Beginn des zweiten Blocks

Falls in s2 kein Segment spezifiziert ist, wird der gleiche Wert benutzt wie für s1.

Alle Differenzen in den beiden Blöcken werden auf dem Bildschirm angezeigt. Die Form ist:

```
a1 b1 a2 b2
```

a1,a2 Adressen in den Blöcken
b1,b2 Inhalte der angezeigten Adressen

Wenn keine Differenzen angezeigt werden, sind die Blöcke identisch.

### 2.2.3. Kommando D (Display)

Das D-Kommando zeigt den Inhalt des Speichers als 8-Bit- oder 16-Bit-Werte hexadezimal und im KOI7-Format an. Die Kommandoformen sind:

```
D
Ds
Ds,f
DW
DWs
DWs,f
```

Es bedeuten:

s 20-Bit-Adresse (Anfangsadresse)
f 16-Bit-Offset innerhalb des in s spezifizierten Segments, wo die Anzeige zu beenden ist

Die Speicheranzeige erfolgt auf einer oder mehreren Zeilen. Jede Zeile zeigt die Werte bis zu 16 Speicherplätzen an. Für die ersten drei Kommandoformen sieht die Anzeige wie folgt aus:

```
ssss:oooo bb bb ... bb cc ... c
```

Es bedeuten:

ssss angezeigtes Segment
oooo Offset im angezeigten Segment
bb Speicherinhalt hexadezimal
cc Speicherinhalt im KOI7-Format. Ein nichtdruckbares Zeichen wird durch einen Punkt (.) angezeigt.

Das Kommando D zeigt den Speicherinhalt ab der aktuellen Adresse in 12 Zeilen an. Das Kommando Ds beginnt mit der Anzeige bei der durch s spezifizierten Adresse und zeigt ebenfalls 12 Zeilen an. Das kommando Ds,f zeigt den Speicherinhalt innerhalb der Grenzen zwischen s und f an.
Die Bedeutung der nächsten drei Kommandoformen sind analog den D-, Ds- und Ds,f-Kommandos, aber es werden die Speicherinhalte in 16-Bit-Werten statt in 8-Bit-Werten angezeigt:

```
ssss:oooo  wwww wwww ... wwww cccc ... cc
```

Während einer langandauernden Anzeige kann das D-Kommando durch Eingabe eines beliebigen Zeichens abgebrochen werden.

### 2.2.4. Kommando E (Load for Execution)

Das E-Kommando bewirkt das Laden in den Speicher, so daß mit einem nachfolgenden G-, T- oder U-Kommando die Programmausführung beginnen kann.
Das E-Kommando hat folgende Form:

```
E<filespec>
```

<filespec> ist die Spezifikation der Datei, die geladen werden soll. Ist kein Dateityp angegeben, wird CMD angenommen. Die Inhalte der Segmentregister und IP-Register werden gemäß der Informationen im Dateikopf der geladenen Datei gesetzt.
Ein E-Kommando gibt Blöcke des Speichers frei, die durch vorherige E- oder R-Kommandos oder durch Programme belegt wurden, wenn diese unter DDT86 ausgefu/hrt worden sind.
Auf diese Weise kann jedenfalls nur eine Datei für die Abarbeitung geladen werden.
Wenn das Laden beendet ist, zeigt DDT86 die Anfangs- und Endadresse jedes Segments der geladenen Datei an.
Mit dem V-Kommando kann diese Information zu einem späteren Zeitpunkt erneut abgefragt werden.
Falls die Datei nicht existiert oder nicht erfolgreich in den verfügbaren Speicher geladen werden kann, gibt DDT86 eine Fehlermitteilung aus.

### 2.2.5. Kommando F (Fill)

Durch das F-Kommando wird ein Speicherbereich mit einer Byte- oder Wortkonstanten gefüllt. Die Kommandoformen sind:

```
Fs,f,b
FWs,f,w
```

Es bedeuten:

- s 20-Bit-Adresse (Anfangsadresse) des zu füllenden Bereichs
- f 16-Bit-Offset (Endadresse) innerhalb des Blockes des durch s spezifizierten Segments
- b 8-Bit-Wert als Eingabekonstante
- w 16-Bit-Wert als Eingabekonstante

Durch das erste Kommando speichert DDT86 den 8-Bit-Wert b auf den Bereich s bis f. Durch das zweite Kommando wird der 16-Bit-Wert w in der Standardform, zuerst das niedere Byte, danach das höhere

Byte, gespeichert.
Falls s größer f bzw. b größer 255 (dezimal) ist, gibt DDT86 ein Fragezeichen aus (?). Falls der eingegebene Wert nicht erfolgreich zurückgelesen werden kann (falls irgendwelche Fehlerbedingungen auftreten), informiert DDT86 ebenfalls mit einer Fehlerausschrift. Ursache kann dann ein defekter oder nichtvorhandener RAM auf der angezeigten Speicheradresse sein.

### 2.2.6. Kommando H (Hexadecimal Math)

Das H-Kommando berechnet die Summe und Differenz aus zwei 16-Bit-Werten. Die Kommandoform ist:

```
Ha,b
```

Die Parameter a und b sind die hexadezimalen Werte, aus denen die Summe und Differenz zu berechnen ist. DDT86 zeigt die Summe (ssss) und Differenz (dddd) auf 16 Bits abgeschnitten auf der nächsten Zeile an:

```
ssss dddd
```

### 2.2.7. Kommando G (Go)

Das G-Kommando überträgt die Steuerung an das zu testende Programm und setzt wahlweise einen oder zwei Unterbrechungspunkte.
Die Kommandoformen sind:

```
G
G,b1
G,b1,b2
Gs
Gs,b1
Gs,b1,b2
```

Es bedeuten:

s 20-Bit-Adresse, ab der das Programm gestartet werden soll
b1,b2 20-Bit-Adressen der Unterbrechungspunkte 1 und 2

Falls kein Segmentwert für eine dieser drei Adressen gegeben ist, wird der Standardsegmentwert aus dem CS-Register benutzt.
In den ersten drei Kommandoformen ist keine Startadresse spezifiziert. DDT86 benutzt in diesen Fällen die 20-Bit-Adresse, die sich aus den Nutzerregistern CS und IP ableitet.
Die erste Form überträgt die Steuerung an das Nutzerprogramm ohne einen Unterbrechungspunkt zu setzen. Die nächsten zwei Formen setzen einen oder zwei Unterbrechungspunkte, bevor die Steuerung an das Nutzerprogramm übertragen wird. Die nächsten drei Kommandoformen sind den ersten drei Formen analog, aber die Nutzerregister CS und IP werden auf s gesetzt.
Ist die Steuerung an das zu testende Programm gegeben, so arbeitet es echtzeitgemäß, bis ein Unterbrechungspunkt erreicht wird. Bei einem Unterbrechungspunkt erhält DDT86 die Steuerung zurück, löscht alle Unterbrechungspunkte und zeigt die Adresse an, bei der der Test unterbrochen wurde:

```
*ssss:oooo
```

ssss entspricht dem Registerinhalt CS und oooo dem Inhalt des Registers IP, bei dem der Unterbrechungspunkt erreicht wurde. Wenn an einem Unterbrechungspunkt die Steuerung an DDT86 geht, ist der Befehl auf dieser Adresse noch nicht ausgeführt worden.

### 2.2.8. Kommando I (Input Command Tail)

Das I-Kommando füllt einen Dateisteuerblock FCB (file control block) und einen Kommandoparameterpuffer in der Basisseite (base page) des DDT86 und kopiert diese Information in die Basisseite der zuletzt mit dem E-Kommando geladenen Datei. Die allgemeine Kommandoform ist:

```
I<command tail>
```

Kommandoparameter <command tail> ist die Zeichenkette, die gewöhnlich einen oder mehrere Dateinamen enthält. Der erste Dateiname wird in den Standard-FCB ab 005CH eingetragen. Die Zeichen der Kommandoparameter werden ebenso in den Standardkommandopuffer ab 0080H der Basisseite kopiert. Die Länge der Zeichenkette wird auf die Adresse 0080H gespeichert, danach folgt die Zeichenkette, und diese wird mit einer binären Null abgeschlossen.
Falls die Datei mit dem E-Kommando geladen worden ist, kopiert DDT86 den FCB und Kommandopuffer von der DDT86-Basisseite in die Basisseite des geladenen Programms. Der Platz der DDT86-Basisseite kann aus dem SS-Register im ZVE-Status des Nutzers erhalten werden, wenn DDT86 gestartet wurde.
Der Beginn der Basisseite des mit E geladenen Programms ist der für DS (Data-Segment) angezeigte Wert, wenn der Ladevorgang abgeschlossen ist.

### 2.2.9. Kommando L (List)

Das L-Kommando listet den Speicherinhalt im Mnemonikformat der Assemblersprache auf. Die allgemeinen Kommandoformen sind:

```
L
Ls
Ls,f
```

Es bedeuten:

s 20-Bit-Anfangsadresse
f 16-Bit-Offset innerhalb des in s spezifizierten Segments, wo die Ausgabe zu beenden ist

Ohne Parameter gibt das L-Kommando 12 Zeilen reassemblierten Maschinencode ab der aktuellen Adresse aus. Ist der Parameter s angegeben, wird vor der Ausgabe die Anfangsadresse auf s gestellt und anschliessend werden 12 Zeilen ausgegeben. Die letzte Form reassembliert Code von s bis f.
In allen Fällen wird die List-Adresse auf die noch nicht ausgelistete Adresse gestellt und damit ein nachfolgendes L-Kommando vorbereitet.
Wenn DDT86 die Steuerung von einem zu testenden Programm (siehe G-, T- und U-Kommando) erhält, wird die List-Adresse auf den aktuellen Wert aus den Registern CS und IP gesetzt.

Langdauernde Ausgaben können durch Eingabe eines beliebigen Zeichens abgebrochen werden. Durch CTRL/S wird die Ausgabe temporär gestoppt. Die Syntax der Mnemonikausgabe, die das L-Kommando ausführt, ist in Abschnitt 2.4. beschrieben.

### 2.2.10. Kommando M (Move)

Das M-Kommando bewirkt den Transport eines Datenblockinhaltes von einem Speicherbereich zu einem anderen. Die Kommandoform ist:

```
Ms,f,d
```

Es bedeuten:

s 20-Bit-Anfangsadresse des zu transportierenden Blockes
f Offset des zu übertragenden letzten Bytes innerhalb des mit s spezifizierten Segments
d 20-Bit-Adresse des ersten Bytes des Zielblockes

Falls für d kein Segment spezifiziert ist, wird der gleiche Wert genommen wie für s. Es ist zu beachten, wenn d zwischen s und f liegt, daß dann ein Teil des zu transportierenden Blockes vor seiner Übertragung überschrieben wird, weil die Daten beginnend ab Platz s übertragen werden.

### 2.2.11. Kommando R (Read)

Das R-Kommando liest eine Datei in einen zusammenhängenden Speicherbereich. Die Kommandoform ist:

```
R<filespec>
```

Als <filespec> ist die Dateispezifikation der zu lesenden Datei anzugeben. Der Dateityp muß angegeben werden.
DDT86 liest die Datei in den Speicher und zeigt die Anfangs- und Endadresse des belegten Bereichs an. Mit dem V-Kommando kann die Belegung zu einem späteren Zeitpunkt zur Anzeige gebracht werden. Der Standardzeiger für die Speicherausgabe (für nachfolgende D-Kommandos) wird auf den Anfang des durch die Datei belegten Speicherbereichs gesetzt.
Das R-Kommando gibt keine Speicheradresse frei, die durch frühere R- oder E-Kommandos belegt wurden. So können mehrere Dateien in den Speicher gelesen werden, ohne daß sie sich überlappen.
Die Anzahl der zu ladenden Dateien ist auf sieben begrenzt. Das ist die von BDOS erlaubte Anzahl minus 1 für DDT86 selbst. Falls die zu ladende Datei nicht existiert oder nicht genügend Speicher vorhanden ist, gibt DDT86 eine Fehlermitteilung aus.

### 2.2.12. Kommando S (Set)

Mit Hilfe des S-Kommandos kann der Inhalt von Bytes oder Worten im Speicher geändert werden. Die Kommandoformen sind:

```
Ss
SWs
```

s ist die 20-Bit-Adresse, deren Inhalt geändert werden soll. DDT86 zeigt die Speicheradresse und den aktuellen Inhalt auf der folgenden Zeile an. Im Ergebnis der ersten Kommandoform wird ausgegeben:

```
ssss:oooo  bb
```

und im Ergebnis der zweiten Form:

```
ssss:oooo  wwww
```

bb und wwww sind die Speicherinhalte in Byte- bzw. Wortformat. Als Reaktion auf die obigen Ausgaben kann der Bediener die Inhalte ändern oder nicht. Wenn ein erlaubter hexadezimaler Wert eingegeben wurde, wird der Inhalt überschrieben. Wird kein Wert angegeben (nur RETURN), bleibt der Inhalt unverändert, und die nächste Adresse wird mit Inhalt angezeigt. DDT86 setzt die Anzeigen kontinuierlich fort bis ein Punkt (.) oder ein unerlaubter Wert eingegeben wurde.
DDT86 gibt eine Fehlermitteilung aus, falls eine Eingabe nicht erfolgreich zurückgelesen werden kann. Ursachen kann ein defekter oder nicht vorhandener RAM für die entsprechende Adresse sein.

### 2.2.13. Kommando T (Trace)

Das T-Kommando bewirkt eine Programmverfolgung im Testmodus (tracing) für 1 bis OFFFFH Programmschritte. Die Kommandoformen sind:

```
T
Tn
TS
TSn
```

n ist die Anzahl der Befehle, die auszuführen ist, bevor die Steuerung an das Bediengerät zurückgeht.
Ehe ein Befehl ausgeführt wird, zeigt DDT86 den aktuellen ZVE-Status und den reassemblierten Befehl an. In den ersten beiden Formen werden die Segmentregister nicht ausgegeben. Das ermöglicht die Anzeige des gesamten ZVE-Status auf einer Zeile.
Die nächsten zwei Formen sind analog den ersten beiden, nur daß alle Register angezeigt werden. Dadurch muß der reassamblierte Befehl auf der nächsten Zeile ausgegeben werden, wie beim X-Kommando.
Bei allen Kommandoformen geht die Steuerung beim Test an das Programm bei der Adresse, die sich aus den Registern CS und IP ergibt. Wenn n nicht angegeben wurde, wird nur ein Befehl ausgeführt. Andernfalls führt DDT86 n Befehle aus und zeigt für jeden Schritt den ZVE-Status an.
Der Testlauf kann vor dem Erreichen des n-ten Befehls durch Eingabe eines beliebigen Zeichens über das Bediengerät abgebrochen werden. Nach einem T-Kommando wird die für das L-Kommando benutzte List-Adresse auf die Adresse des nächsten auszufu/hrenden Befehls gesetzt.
Es ist zu beachten, daß DDT86 die Routinen des BDOS nicht interpretiert. Eine gesamte BDOS-Routine wird wie ein Befehl betrachtet.

Auf eine Besonderheit sei hier verwiesen. Beim Laden des Stack-Segmentregisters wird von der ZVE der folgende Befehl ohne Unterbrechungsmöglichkeit angeschlossen. Im Trace-Modus wird der nächste Befehl zwar ausgeführt, aber nicht interpretiert. Dieser Befehl muß das Laden des zugehörigen Stack-Pointers SP enthalten. Also beispielsweise:

```
MOV  SS,AX
MOV  SP,OFFSET STACK_PTR
```

Wird diese Befehlsfolge nicht eingehalten, so treten unkontrollierte Fehler bei der Abarbeitung im Trace-Modus auf.

### 2.2.14. Kommando U (Untrace)

Das U-Kommando ist mit dem T-Kommando bis auf die Ausnahme identisch, daß der ZVE-Status nur vor der Ausführung des ersten Befehls angezeigt wird anstatt vor jedem Schritt. Die Kommandoformen sind:

```
U
Un
US
USn
```

n ist hier wie beim T-Kommando die Anzahl der auszuführenden Befehle bis zur Rückgabe der Steuerung an das Bediengerät. Das U-Kommando kann durch Eingabe eines beliebigen Zeichens über das Bediengerät abgebrochen werden.

### 2.2.15. Kommando V (Value)

Das V-Kommando zeigt Informationen über die Speicherbelegung der letzten mit dem E- oder R-Kommando geladene Datei. Die Kommandoform ist:

```
V
```

Wenn die letzte Datei mit dem E-Kommando geladen wurde, zeigt das V-Kommando die Anfangs- und Endadresse jedes in der Datei enthaltenen Segmentes an. Falls die letzte Datei mit dem R-Kommando gelesen wurde, zeigt das V-Kommando die Anfangs- und Endadresse des Speicherblockes an, auf den die Datei gelesen wurde. Wenn weder das E- noch das R-Kommando benutzt wurden, antwortet DDT86 auf das V-Kommando mit "?".

### 2.2.16. Kommando W (Write)

Das W-Kommando schreibt den Inhalt eines zusammenhängenden Speicherblockes zur Platte. Die Kommandoformen sind:

```
W<filespec>
W<filespec>,s,f
```

<filespec> gibt die Spezifikation der Datei auf der Platte an, welche die Daten empfangen soll.
s und f sind die Anfangs- bzw. Endadresse des Speicherblockes, der geschrieben werden soll (20-Bit-Adressen).
Falls das Segment in f nicht spezifiziert ist, wird die gleiche Segmentadresse benutzt, die auch für s benutzt wurde.
Werden die Parameter s und f nicht angegeben, übernimmt DDT86 dafür die Werte von der letzten mit einem R-Kommando gelesenen Datei. Wurde kein R-Kommando benutzt, antwortet DDT86 mit "?". Diese erste Kommandoform ist sinnvoll, wenn Dateien nach einer Korrektur rückgeschrieben werden sollen, deren Länge sich durch die Korrektur nicht geändert haben.

Werden die Parameter s und f als 20-Bit-Werte angegeben, so werden die niedrigsten vier Bits von s ignoriert, d.h. der zu schreibende Block muß immer an einer Paragraphengrenze beginnen. Wenn die Datei mit dem im W-Kommando spezifizierten Namen bereits existiert, dann löscht DDT86 diese Datei, ehe eine neue Datei geschrieben wird.

### 2.2.17. Kommando X (Examine CPU State)

Das X-Kommando erlaubt dem Bediener den ZVE-Status des zu testenden Programms zu prüfen und zu ändern. Die Kommandoformen sind:

```
X
Xr
Xf
```

r ist der Name eines der 1700-ZVE-Register und f die Abkürzung eines der ZVE-Flags.
Das Kommando in der Form X zeigt den ZVE-Status wie folgt an:

```
          AX   BX   CX  ...  SS   ES   IP
--------- xxxx xxxx xxxx ... xxxx xxxx xxxx
<befehl>
```

Die neun Striche am Anfang der Zeile zeigen den Status der neun ZVE-Flags. Jede dieser Positionen kennzeichnet als Strich ein nichtgesetztes (0) Flag. Ein gesetztes Flag (1) wird mit einem Abkürzungszeichen für den Flagnamen gekennzeichnet.
Die Abkürzungszeichen sind in der unten stehenden Tabelle 21 angegeben. <befehl> ist der reassemblierte Befehl auf dem nächsten auszuführenden Platz, der sich aus den Registern CS und IP ergibt.

Tabelle 21: Flagnamen-Abkürzungen

| Zeichen | Name |
|---|---|
| O | Überlauf (Overflow) |
| D | Richtung (Direction) |
| I | Unterbrechung erlaubt (Interrupt Enable) |
| T | Trap (Trap) |
| S | Vorzeichen (Sign) |
| Z | Null (Zero) |
| A | Hilfsübertrag (Auxiliary Carry) |
| P | Parität (Parity) |
| C | Übertrag (Carry) |

Das Kommando der Form Xr ermöglicht dem Bediener, das Register im ZVE-Status des zu testenden Programms zu ändern. r ist der Name eines der 16-Bit-ZVE-Register. DDT86 reagiert auf dieses Kommando mit Ausgabe des Namens und aktuellem Inhalt des Registers. Gibt der Bediener dann ein (CR) ein, bleibt der Inhalt unverändert. Wurde ein zulässiger Wert eingegeben, wird der Registerinhalt entsprechend verändert. In beiden Fällen wird anschließend der nächste Registername und -inhalt ausgegeben. Dieser Prozeß wird fortgesetzt bis ein Punkt (.) oder ein unzulässiger Wert eingegeben bzw. das letzte Register bearbeitet

wurde.
Das Kommando der Form Xf erlaubt dem Bediener, eines der Flags im ZVE-Status des zu testenden Programms zu ändern. DDT86 gibt nach diesem Kommando den Flagnamen und -inhalt aus. Mit Eingabe von (RETURN) bleibt der Inhalt unverändert. Ein zulässiger Wert (0 oder 1) wird in das Flag eingetragen.
Es kann mit einem Xf-Kommando immer nur ein Flag geprüft oder geändert werden.

## 2.3. Standardsegmentwerte

DDT86 besitzt einen internen Mechanismus, der die aktuellen Segmentwerte aufbewahrt. Diese werden für die Bildung der Segmentspezifikation in den Kommandos von DDT86 benötigt, falls kein Segmentwert angegeben wurde.
DDT86 unterteilt seinen Kommandosatz in zwei Kommandotypen entsprechend des Segments, das benutzt wird, wenn im Kommando kein Segmentwert spezifiziert ist. Der erste Kommandotyp bezieht sich auf das Code-Segment. Dazu gehören die Kommandos A, L und W.
Diese Kommandos benutzen den interen Typ-1-Segmentwert, wenn kein Segmentwert im Kommando spezifiziert wurde.
Bei Neustart setzt DDT86 den Wert des Typ-1-Segmentwertes auf Null und ändert ihn, wenn eine der folgenden Handlungen geschieht:

- Wenn eine Datei mit dem E-Kommando geladen wurde, setzt DDT86 den Typ-1-Segmentwert auf den Wert des CS-Registers.

- Wenn eine Datei mit dem R-Kommando gelesen wurde, setzt DDT86 den Typ-1-Segmentwert auf das Basissegment, wohin die Datei gelesen wurde.

- Wenn ein X-Kommando den Wert des CS-Registers ändert, wird der Typ-1-Segmentwert auf diesen neuen CS-Wert geändert.

- Wenn DDT86 die Steuerung von einem Nutzerprogamm nach einem G-, T- oder U-Kommando erhält, setzt es den Wert des Typ-1-Segments auf den des CS-Registers.

- Wenn ein Segmentwert explizit in einem A- oder L-Kommando spezifiziert wird, setzt DDT86 den Wert des Typ-1-Segments auf diesen spezifizierten Wert.

Der zweite Kommandotyp bezieht sich auf das Data-Segment. Dazu gehören die Kommandos D, F, M und S.

Diese Kommandos benutzen den internen Typ-2-Segmentwert, wenn ein Kommando keinen Segmentwert spezifiziert.
Bei Neustart setzt DDT86 den Typ-2-Segmentwert auf Null und ändert ihn bei folgenden Ereignissen:

- Wenn eine Datei mit dem E-Kommando geladen wurde, setzt DDT86 den Typ-2-Segmentwert auf den Wert des DS-Registers.

- Wenn eine Datei mit dem R-Kommando gelesen wurde, wird dem Typ-2-Segmentwert der Basis-Segmentwert zugeordnet, auf den die Datei gelesen wurde.

- Wenn mittels X-Kommando das DS-Register geändert wurde, trägt DDT86 diesen Wert als Typ-2-Segmentwert ein.

- Wenn DDT86 die Steuerung von einem zu testenden Programm nach einem G-, T- oder U-Kommando erhält, setzt es den Typ-2-Segmentwert auf den des DS-Registers.

- Wenn ein Segmentwert explizit in einem D-, F-, M- oder S-Kommando spezifiziert ist, übernimmt DDT86 diesen für den Typ-2-Segmentwert.

Wenn rechnende Programme identische Werte in CS- und DS-Registern benutzen, greift DDT86 mit allen Kommandos auf die gleichen Segmentwerte zu, wenn das nicht explizit verhindert wird.
Es ist zu beachten, daß das G-Kommando nicht unter diese beiden Gruppen fällt, da es standardmäßig das CS-Register benutzt.

Tabelle 22: DDT86-Standardsegmentwerte

| Kommando | Typ-1 | Typ-2 |
|---|---|---|
| A | x | |
| B | | x |
| D | | x |
| E | u | u |
| F | | x |
| G | u | u |
| H | | |
| I | | |
| L | x | |
| M | | x |
| R | u | u |
| S | | x |
| T | u | u |
| U | u | u |
| V | | |
| W | x | |
| X | u | u |

x — benutzt diesen Standardsegmentwert, falls keine Angabe im Kommando erfolgt; ändert den Standardsegmentwert, falls im Kommando ein Segmentwert spezifiziert wurde

u — Ändert diesen Segmentwert

## 2.4. Assemblersprachsyntax für A- und L-Kommando

Im allgemeinen ist die Syntax der A- und L-Kommandos identisch mit der Syntax der Assemblersprache von ASM86. Es gibt nur geringfügige Abweichungen, die im folgenden aufgezählt werden.

- DDT86 setzt voraus, daß alle eingegebenen numerischen Werte hexadezimal sind.

- Bis zu drei Präfixe (LOCK, REPEAT, SEGMENT OVERRIDE)) können in einem Assemblerbefehl angewendet werden, aber sie müssen vor dem Operationscode des Befehls stehen. Ein Präfix kann aber auch allein auf einer Zeile stehen.

- Die Unterscheidung von Befehlen für Byte- und Wortzeichenverarbeitung geschieht wie folgt:

| Byte | Wort |
|---|---|
| LODSB | LODSW |
| STOSB | STOSW |
| SCASB | SCASW |
| MOVSB | MOVSW |
| CMPSB | CMPSW |

- Die Mnemoniks für Befehle des Nah- und Ferntransfers der Steuerung sind folgende:

| kurz (short) | normal | weit (far) |
|---|---|---|
| JMPS | JMP | JMPF |
| | CALL | CALLF |
| | RET | RETF |

- Wenn der Operand eines CALLF- oder JMPF-Befehls eine absolute 20-Bit-Adresse ist, so wird sie in folgender Form eingegeben:

```
ssss:oooo
```

Dabei ist ssss das Segment und 0000 der Offset der Adresse.

- Operanden, die sich sowohl auf Bytes als auch auf Worte beziehen könnten, müssen das Präfix "BYTE" oder "WORD" vorausgesetzt bekommen. Diese Präfixe können durch "BY" oder "WO" abgekürzt werden. Zum Beispiel:

```
INC  BYTE[BP]
NOT  WORD[1234]
```

Falsche Anwendung führt zur Fehlermitteilung.

- Indirektoperanden werden durch Einschließen in eckige Klammern von den Direktoperanden unterschieden. Zum Beispiel:

```
ADD  AX,5   ;Addiere 5 zum Registerinhalt AX
ADD  AX,[5] ;Addiere den Inhalt von Zelle 5 zu AX
```

Registerformen der indirekten Speicheroperanden sind:

```
[pointer register]
[index register]
[pointer register + index register]
```

"pointer register" sind BX und BP, "index register" sind SI und DI. Vor alle diese Registerformen können numerische Offsets gestellt werden. Zum Beispiel:

```
ADD  BX,[BP+SI]
ADD  BX,3[BP+SI]
ADD  BX,1D47[BP+SI]
```

## 2.5. Beispiel für die Arbeit mit DDT86

Im folgenden wird ein Beispiel für die Arbeit mit DDT86 gegeben. Ein einfaches Sortierprogramm wird interaktiv getestet. Kommentare in eckigen Klammern [] erläutern die Kommandos.

[Quelltext des Testprogramms]

```
A>type sort.a86
;
;       simple sort program
;
sort:
        mov     si,0            ;initialize index
        mov     bx,offset nlist ;bx = base of list
        mov     sw,0            ;clear switch flag
comp:
        mov     al,[bx+si]      ;get byte from list
        cmp     al,1[bx+si]     ;compare with next byte
        jna     inci            ;don't switch if in order
        xchg    al,1[bx+si]     ;do first part of switch
        mov     [bx+si],al      ;do second part
        mov     sw,1            ;set switch flag
inci:
        inc             si;increment index
        cmp     si,count        ;end of list?
        inz     comp            ;no, keep going
        test    sw,1            ;done - any switches?
        jnz     sort            ;yes, sort some more
done:
        jmp     done            ;get here when list ordered
;
        dseg
        org     100h            ;leave space for base page
;
nlist   db      3,8,4,6,31,6,4,1
count   equ     offset $ - offset nlist
sw      db      0
        end
```

[Assemblieren des Testprogramms]

```
A>asm sort

ASM86 V 1.0
END OF PASS 1
END OF PASS 2
END OF ASSEMBLY. NUMBER OF ERRORS:   0
```

[Ausgabe der Listendatei]

```
A>type sort.lst
ASM86 V 1.0  SOURCE: SORT.A86


                ;
                ;       simple sort program
                ;
                sort:
0000 BE0000             mov     si,0            ;initialize index
0003 BB0001             mov     bx,offset nlist ;bx = base of list
0006 C606080100         mov     sw,0            ;clear switch flag
                comp:
000B 8A00               mov     al,[bx+si]      ;get byte from list
000D 3A4001             cmp     al,1[bx+si]     ;compare with next byte
0010 760A               jna     inci            ;don't switch if in order
0012 864001             xchq    al,1[bx+si]     ;do first part of switch
0015 8800               mov     [bx+si],al      ;do second part
0017 C606080101         mov     sw,1            ;set switch flag
                inci:
001C 46                 inc     si              ;increment index
001D 83FE08             cmp     si,count        ;end of list?
0020 75E9               jnz     comp            ;no, keep going
0022 F606080101         test    sw,1            ;done - any switches?
0027 75D7               jnz     sort            ;yes, sort some more
                done:
0029 E9FDFF             jmp     done            ;get here when list ordered
                ;
                        dseg
                        org     100h            ;leave space for base page
                ;

0100 030804061F06 nlist  db     3,8,4,6,31,6,4,1
     0401
  0008            count  equ    offset $ - offset nlist
0108 00           sw     db     0
                         end


END OF ASSEMBLY. NUMBER OF ERRORS:    0
```

```
        [Ausgabe der Symboltabelle]
A>type sort.sym
0000 VARIABLES
0100 NLIST      0108  SW

0000 NUMBERS
0008 COUNT

0000 LABELS
000B COMP       0029 DONE       001C INCI       0000 SORT

        [Ausgabe der Code-Datei]
A>type sort.h86
:0400000300000000F9
:1B000081BE0000BB0001C6060801008A003A4001760A864001880C606080016C
:11001B81014683FE0875E9F606080101750D7E9FDFFBE
:0901008203080406 1F0604010035
:00000001FF

        [Generieren der CMD-Datei]
A>gencmd sort

BYTES READ    0039
RECORDS WRITTEN 04

        [Aufruf DDT86 und Laden SORT.CMD]
A>ddt sort
DDT86 V 1.0
     START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

        [Anzeige der initialisierten Registerinhalte]
-x
            AX   BX   CX   DX   SP   BP   SI   DI   CS   DS   SS   ES   IP
---------- 0000 0000 0000 0000 119E 0000 0000 0000 047D 0480 0491 0480 0000
MOV    SI,0000
```

```
        [Reassemblieren des Beginns des Code-Segments]
-l
047D:0000 MOV     SI,0000
047D:0003 MOV     BX,0100
047D:0006 MOV     BYTE,[0108],00
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE[0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B

        [Anzeige des Beginns des Data-Segments]
-d100,10f
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ............

        [Reassemblieren des restlichen Codes]
-l
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
047D:002C ADD     [BX+SI],AL
047D:002E ADD     [BX+SI],AL
047D:0030 DAS
047D:0031 ADD     [BX+SI],AL
047D:0033 ??=     6C
047D:0034 POP     ES
047D:0035 ADD     [BX],CL
047D:0037 ADD     [BX+SI],AX
047D:0039 ??=     6F
```

```
        [Ausführen des Programms von IP=0,
         Unterbrechungspunkt auf 29H]
-g,29
*047D:0029   [Unterbrechungspunkt erreicht]

        [Anzeige der sortierten Liste]
-d100,10f
0480:0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

        [Die Liste ist fehlerhaft, erneut Datei laden]
-esort
     START     END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

        [Testen der ersten drei Befehle]
-t3
            AX   BX   CX   DX   SP   BP   SI   DI   IP
------Z-P 0000 0100 0000 0000 119E 0000 0008 0000 0000 MOV   SI,0000
------Z-P 0000 0100 0000 0000 119E 0000 0000 0000 0003 MOV   BX,0100
------Z-P 0000 0100 0000 0000 119E 0000 0000 0000 0006 MOV   BYTE [0108],00
*047D:000B

        [Weitere Befehle testen]
-t3
            AX   BX   CX   DX   SP   BP   SI   DI   IP
------Z-P- 0000 0100 0000 0000 119E 0000 0000 0000 000B MOV   AL,[BX+SI]
------Z-P- 0003 0100 0000 0000 119E 0000 0000 0000 000D CMP   AL,01[BX+SI]
-----S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 0010 JBE   001C
*047D:001C

        [Anzeige der unsortierten Liste]
-d100,10f
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ................
```

```
        [Anzeige der nächsten Befehle, die ausgeführt werden sollen]
-l
047D:001C INC     SI
047D:001D CMD     SI,0008
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
047D:002C ADD     [BX+SI],AL
047D:002E ADD     [BX+SI],AL
047D:0030 DAS
047D:0031 ADD     [BX+SI],AL
047D:0033 ??=     6C
047D:0034 POP     ES

        [Testen weiterer Befehle]
-t3
              AX   BX   CX   DX   SP   BP   SI   DI   IP
-----S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 001C INC   SI
--------C 0003 0100 0000 0000 119E 0000 0001 0000 001D CMP   SI,0008
-----S-APC 0003 0100 0000 0000 119E 0000 0001 0000 0020 JNZ   0008
*047D:000B

        [Anzeige der weiteren Befehle]
-l
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
```

```
-t3
          AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-APC 0003 0100 0000 0000 119E 0000 0001 0000 000B MOV   AL,[BX+SI]
----S-APC 0008 0100 0000 0000 119E 0000 0001 0000 000D CMP   AL,01[BX+SI]
--------- 0008 0100 0000 0000 119E 0000 0001 0000 0010 JBE   001C
*047D:0012

-l
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     0008
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
047D:002C ADD     [BX+SI],AL
047D:002E ADD     [BX+SI],AL
047D:0030 DAS

        [Weiter bis der Austausch durchgeführt wurde]
-g,20
*047D:0020

        [Anzeige der Liste]
-d100,10f
0480:0100 03 04 08 06 1F 06 04 01 01 00 00 00 00 00 00 00 ................

        [4 und 8 sind offenbar richtig vertauscht]
-t
          AX   BX   CX   SP   BP   SI   DI   IP
----S-APC 0004 0100 0000 119E 0000 0002 0000 0020 JNZ   000B
*047D:000B
```

```
        [Anzeige der nächsten Befehle]
-l
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029

        [Erneut laden, testen der Grenzbedingungen]
-esort
     START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

        [Um schneller voranzukommen, wird die Listenlänge auf 3 gesetzt]
-a1d
047:001D cmp si,3
047D:0020
        [Anzeige der unsortierten Liste]
-d100
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ................
0480:0110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0480:0120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 20 ..............
        [Unterbrechungspunkte, wenn die ersten 3 Elemente sortiert
         sein sollten]
-g,29
*047D:0029

        [Anzeige der Liste]
-d100,10f
0480:0100 03 04 06 08 1F 06 04 01 00 00 00 00 00 00 00 00 ................
```

```
        [Das vierte Element scheint auch bereits sortiert zu sein]
-esort
     START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

        [Wiederholter Versuch mit einigen Testschritten]
-a1d
047D:001D cmp si,3
047D:0020

-t9
          AX   BX   CX   DX   SP   BP   SI   DI   IP
-----Z-P 0006 0100 0000 0000 119E 0000 0003 0000 0000 MOV   SI,0000
-----Z-P 0006 0100 0000 0000 119E 0000 0000 0000 0003 MOV   BX,0100
-----Z-P 0006 0100 0000 0000 119E 0000 0000 0000 0006 MOV   BYTE [0108],00
-----Z-P 0006 0100 0000 0000 119E 0000 0000 0000 000B MOV   AL,[BX+SI]
-----Z-P 0003 0100 0000 0000 119E 0000 0000 0000 000D CMP   AL,01[BX+SI]
---S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 0010 JBE   001C
---S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 001C INC   SI
-------C 0003 0100 0000 0000 119E 0000 0001 0000 001D CMP   SI,0003
---S-A-C 0003 0100 0000 0000 119E 0000 0001 0000 0020 JNZ   000B
*047D:000B

-l
047D:000B MOV    AL,[BX+SI]
047D:000D CMP    AL,01[BX+SI]
047D:0010 JBE    001C
047D:0012 XCHG   AL,01[BX+SI]
047D:0015 MOV    [BX+SI],AL
047D:0017 MOV    BYTE [0108],01
047D:001C INC    SI
047D:001D CMP    SI,0003
047D:0020 JNZ    000B
047D:0022 TEST   BYTE [0108],01
047D:0027 JNZ    0000
047D:0029 JMP    0029
```

```
-t3
          AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-A-C 0003 0100 0000 0000 119E 0000 0001 0000 000B MOV   AL,[BX+SI]
----S-A-C 0008 0100 0000 0000 119E 0000 0001 0000 000D CMP   AL,01[BX+SI]
-------- 0008 0100 0000 0000 119E 0000 0001 0000 0010 JBE   001C
*047D:0012

-l
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0003
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01

-t3
          AX   BX   CX   DX   SP   BP   SI   DI   IP
-------- 0008 0100 0000 0000 119E 0000 0001 0000 0012 XCHG    AL,01[BX+SI]
-------- 0004 0100 0000 0000 119E 0000 0001 0000 0015 MOV     [BX+SI],AL
-------- 0004 0100 0000 0000 119E 0000 0001 0000 0017 MOV     BYTE [0108],01
*047D:001C

-d100,10f
0480:0100 03 04 08 06 1F 06 04 01 01 00 00 00 00 00 00 00 ................
```

```
        [So weit in Ordnung]
-t3
              AX   BX   CX   DX   SP   BP   SI   DI   IP
--------- 0004 0100 0000 0000 119E 0000 0001 0000 001C INC   SI
--------- 0004 0100 0000 0000 119E 0000 0002 0000 001D CMP   SI,0003
----S-APC 0004 0100 0000 0000 119E 0000 0002 0000 0020 JNZ   000B
*047D:000B

-l
047D:000B MOV    AL,[BX+SI]
047D:000D CMP    AL,01[BX+SI]
047D:0010 JBE    001C
047D:0012 XCHG   AL,01[BX+SI]
047D:0015 MOV    [BX+SI],AL
047D:0017 MOV    BYTE [0108],01
047D:001C INC    SI
047D:001D CMP    SI,0003
047D:0020 JNZ    000B
047D:0022 TEST   BYTE [0108],01
047D:0027 JNZ    0000
047D:0029 JMP    0029

-t3
              AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-APC 0004 0100 0000 0000 119E 0000 0002 0000 000B MOV   AL,[BX+SI]
----S-APC 0008 0100 0000 0000 119E 0000 0002 0000 000D CMP   AL,01[BX+SI]
--------- 0008 0100 0000 0000 119E 0000 0002 0000 0010 JBE   001C
*047D:0012
```

```
        [Hier wird schon das dritte und vierte Element der Liste
         verglichen.
         Wieder Laden]
-esort
     START       END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F

-l
047D:0000 MOV     SI,0000
047D:0003 MOV     BX,0100
047D:0006 MOV     BYTE [0108],00
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B

        [Korrigieren der Länge]
-a1d
047D:001D cmp si,7
047D:0020

        [Versuch mit korrigierter Länge]
-g,29
*047D:0029

        [Anzeige der Liste]
-d100,10f
0480:0100 01 03 04 04 06 06 08 1F 00 00 00 00 00 00 00 00 ................
```

```
        [Es scheint zu gehen. Nun soll die Korrektur in der Datei
         installiert werden. Dazu muß die gesamte Datei
         einschließlich Dateiheader gelesen werden.]

-rsort.cmd
  START      END
2000:0000 2000:01FF

        [Die ersten 80H Bytes enthalten den Dateiheader. Der Code
         beginnt daher bei 80H.]
-180
2000:0080 MOV    SI,0000
2000:0083 MOV    BX,0100
2000:0086 MOV    BYTE [0108],00
2000:008B MOV    AL,[BX+SI]
2000:008D CMP    AL,01[BX+SI]
2000:0090 JBE    009C
2000:0092 XCHG   AL,01[BX+SI]
2000:0095 MOV    [BX+SI],AL
2000:0097 MOV    BYTE [0108],01
2000:009C INC    SI
2000:009D CMP    SI,0008
2000:00A0 JNZ    008B

        [Installieren der Korrektur]

-a9d
2000:009D cmp si,7

        [Rückschreiben der Datei zur Platte. (Die Länge der
         Datei bleibt unverändert. Daher wurde keine Länge
         spezifiziert.)]

-wsort.cmd

-esort

     START      END
CS 047D:0000 047D:002F
DS 0480:0000 0480:010F
```

```
        [Überprüfen, ob die Korrektur installiert ist]
-l
047D:0000 MOV     SI,0000
047D:0003 MOV     BX,0100
047D:0006 MOV     BYTE, [0108],00
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0007
047D:0020 JNZ     000B

        [Abarbeiten]
-g,29
*047D:0029

        [Anzeige der sortierten Liste. Rückgabe der Steuerung!]
-d100,10f
0480:0100 01 03 04 04 06 06 08 1F 00 00 00 00 00 00 00 00 ................
-^C
A>
```

## Anlage 1 ASM86-Aufruf

**Kommando:** ASM86

**Syntax:** ASM86 filespec [¤parameter...]

**filespec** Spezifikation der Assembler-Quelldatei (Laufwerk und Dateityp sind wahlfrei)

**parameter** Buchstabe für Typ, gefolgt von 1 Buchstaben für das Gerät (siehe Tabelle 23)

**Standard-Dateityp:** A86

**Parameter:** ¤Td wobei T = Typ und d = Gerät

Tabelle 23: Parametertypen und Geräte

| Typ: | A | H | P | S |
|---|---|---|---|---|
| Geräte: | | | | |
| A - P | x | x | x | x |
| X | | x | x | x |
| Y | | x | x | x |
| Z | | x | x | x |

x = zulässig

Zulässige Parameter:

Das aktuelle Laufwerk ist das Standardgerät.

Tabelle 24: Parametertypen

| Typ | Wirkung |
|---|---|
| A | steuert das Laufwerk der Assembler-Quelldatei |
| H | steuert das Laufwerk der Code-Datei |
| P | steuert das Laufwerk der Listendatei |
| S | steuert das Laufwerk der Symboldatei |

Tabelle 25: Gerätetypen

| Typ | Gerät |
|---|---|
| A - P | Laufwerk A - P |
| X | Bildschirm |
| Y | Drucker |
| Z | keine Ausgabe |

Tabelle 26: Aufrufbeispiele

| Beispiel | Wirkung |
|---|---|
| ASM86 I0 | assembliert Datei I0.A86, erzeugt I0.H86, I0.LST und I0.SYM |
| ASM86 I0.ASM ¤ AD SZ | assembliert Datei I0.ASM von Gerät D, erzeugt I0.LST und I0.H86, keine Symboldatei |
| ASM86 I0 ¤ PY SX | assembliert Datei I0.A86, erzeugt I0.H86, Listenausgabe auf dem Drucker, Symbolausgabe auf dem Bildschirm |

## Anlage 2 Mnemonikunterschiede zum BOS-1810-Assembler

Der Assembler ASM86 benutzt die gleichen Befehle wie der Assembler ASM86 des BOS 1810, ausgenommen Mnemoniks für Fern- und Kurzsprünge, Rufe und Rückkehr. Die folgende Tabelle zeigt die vier Unterschiede:

Tabelle 27: Mnemonikunterschiede

| Mnemonikfunktion | SCP 1700 | BOS 1810 |
|---|---|---|
| Kurzsprung innerhalb eines Segmentes: | JMPS | JMP |
| Sprung zwischen Segmenten: | JMPF | JMP |
| Rückkehr zwischen Segmenten: | RETF | RET |
| Ruf zwischen Segmenten: | CALLF | CALL |

## Anlage 3 – Hexadezimales Ausgabeformat

Tabelle 28 definiert die Folge und Inhalte der Bytes in einem Hexadezimalsatz.

Jeder hexadezimale Satz hat eine der vier Formate, die Tabelle 29 zeigt.

Beispiel:

Bytezahl = 0 1 2 3 4 5 6 7 8 9 ... n

Inhalte = : l l a a a a t t d d d ... c c (CR) (LF)

Tabelle 28: Hexadezimale Satzinhalte

| Byte | Inhalt | Symbol |
|---|---|---|
| 0 | Satzmarke | : |
| 1-2 | Satzlänge | l l |
| 3-6 | Ladeadresse | a a a a |
| 7-8 | Satztyp | |
| 9-(n-1) | Datenbytes | d d ... d |
| n-(n+1) | Prüfsumme | c c |
| n+2 | Wagenrücklauf | (CR) |
| n+3 | Zeilenschaltung | (LF) |

Tabelle 29: Hexadezimale Satzformate

| Satztyp | Inhalt | Format |
|---|---|---|
| 00 | Datensatz | : ll aaaa DT Daten... cc |
| 01 | Dateiende | : 00 0000 01 FF |
| 02 | erweiterte Zieladresse | : 02 0000 ST ssss cc |
| 03 | Anfangsadresse | : 04 0000 03 ssss iiii cc |

ll = Satzlänge -- Anzahl der Datenbytes

cc = Prüfsumme -- Summe aller Satzbytes

aaaa = 16-Bit-Adresse

ssss = 16-Bit-Segmentwert

iiii = Offsetwert von der Anfangsadresse

DT = Datensatztyp

ST = Segmentadressen-Satztyp

Tabelle 30: Segmentsatztypen

| Symbol | Wert | Bedeutung |
|---|---|---|
| DT | 81H | für Daten, die zum Code-Segment gehören |
| | 82H | für Daten, die zum Data-Segment gehören |
| | 83H | für Daten, die zum Stack-Segment gehören |
| | 84H | für Daten, die zum Extra-Segment gehören |
| ST | 85H | für eine absolute Code-Segmentadresse |
| | 86H | für eine absolute Data-Segmentadresse |
| | 87H | für eine absolute Stack-Segmentadresse |
| | 88H | für eine absolute Extra-Segmentadresse |

## Anlage 4 Reservierte Worte

Tabelle 31: Reservierte Worte

| | | | | |
|---|---|---|---|---|
| **Vordefinierte Zahlen** | | | | |
| BYTE | WORD | DWORD | | |
| **Operatoren** | | | | |
| EQ | GE | GT | LE | LT |
| NE | OR | AND | MOD | NOT |
| PTR | SEG | SHL | SHR | XOR |
| LAST | TYPE | LENGTH | OFFSET | |
| **Assembler-Direktiven** | | | | |
| DB | DD | DW | IF | RS |
| RB | RW | END | ENDM | EQU |
| ORG | CSEG | DSEG | ESEG | SSEG |
| EJECT | ENDIF | TITLE | LIST | NOLIST |
| INCLUDE | SIMFORM | PAGESIZE | CODEMACRO | PAGEWIDTH |
| **Code-Makro-Direktiven** | | | | |
| DB | DD | DW | DBIT | RELB |
| RELW | MODRM | SEGFIX | NOSEGFIX | |
| **Register** | | | | |
| AH | AL | AX | BH | BL |
| BP | BX | CH | CL | CS |
| CX | DH | DI | DL | DS |
| DX | ES | SI | SP | SS |
| **Befehlsmnemoniks -- siehe Anlage 7** | | | | |

Anlage 5 ASM86 - Zusammenfassung der Befehle

Tabelle 32: ASM86 - Zusammenfassung der Befehle

| Mnemonik | Beschreibung | Abschnitt |
|---|---|---|
| AAA | ASCII adjust for Addition | 1.4.2. |
| AAD | ASCII adjust for Division | 1.4.2. |
| AAM | ASCII adjust for Multiplikation | 1.4.2. |
| AAS | ASCII adjust for Subtraction | 1.4.2. |
| ADC | Add with Carry | 1.4.3. |
| ADD | Add | 1.4.2. |
| AND | And | 1.4.2. |
| CALL | Call (intra segment) | 1.4.4. |
| CALLF | Call (inter segment) | 1.4.4. |
| CBW | Convert Byte to Word | 1.4.2. |
| CLC | Clear Carry | 1.4.5. |
| CLD | Clear Direction | 1.4.5. |
| CLI | Clear Interrupt | 1.4.5. |
| CMC | Completment Carry | 1.4.5. |
| CMP | Compare | 1.4.2. |
| CMPS | Compare Byte or Word (of string) | 1.4.3. |
| CWD | Convert Word to Double Word | 1.4.2. |
| DAA | Decimal Adjust for Addition | 1.4.2. |
| DAS | Decimal Adjust for Subtraction | 1.4.2. |
| DEC | Decrement | 1.4.2. |
| DIV | Divide | 1.4.2. |
| ESC | Escape | 1.4.5. |
| HLT | Halt | 1.4.5. |
| IDIV | Integer Divide | 1.4.2. |
| IMUL | Integer Multiply | 1.4.2. |
| IN | Input Byte or Word | 1.4.1. |
| INC | Increment | 1.4.2. |
| INT | Interrupt | 1.4.4. |
| INTO | Interrupt on Overflow | 1.4.4. |
| IRET | Interrupt Return | 1.4.4. |
| JA | Jump on Above | 1.4.4. |
| JAE | Jump on Above or Equal | 1.4.4. |
| JB | Jump on Below | 1.4.5. |
| JBE | Jump on Below or Equal | 1.4.4. |
| JC | Jump on Carry | 1.4.4. |
| JCXZ | Jump on CX Zero | 1.4.4. |
| JE | Jump on Equal | 1.4.4. |
| JG | Jump on Greater | 1.4.4. |
| JGE | Jump on Greater or Equal | 1.4.4. |
| JL | Jump on Less | 1.4.4. |
| JLE | Jump on Less or Equal | 1.4.4. |
| JMP | Jump (intra segment) | 1.4.4. |
| JMPF | Jump (inter segment) | 1.4.4. |
| JMPS | Jump (8 bit displacement) | 1.4.4. |
| JNA | Jump on Not Above | 1.4.4. |
| JNAE | Jump on Not Above or Equal | 1.4.4. |
| JNB | Jump on Not Below | 1.4.4. |

Tabelle 32: (Fortsetzung)

| Mnemonik | Beschreibung | Abschnitt |
|---|---|---|
| JNBE | Jump on Not Below or Equal | 1.4.4. |
| JNC | Jump on Not Carry | 1.4.4. |
| JNE | Jump on Not Equal | 1.4.4. |
| JNG | Jump on Not Greater | 1.4.4. |
| JNGE | Jump on Not Greater or Equal | 1.4.4. |
| JNL | Jump on Not Less | 1.4.4. |
| JNLE | Jump on Not Less or Equal | 1.4.4. |
| JNO | Jump on Not Overflow | 1.4.4. |
| JNP | Jump on Not Parity | 1.4.4. |
| JNS | Jump on Not Sing | 1.4.4. |
| JNZ | Jump on Not Zero | 1.4.4. |
| JO | Jump on Overflow | 1.4.4. |
| JP | Jump on Parity | 1.4.4. |
| JPE | Jump on Parity Even | 1.4.4. |
| JPO | Jump on Parity Odd | 1.4.4. |
| JS | Jump on Sign | 1.4.4. |
| JZ | Jump on Zero | 1.4.4. |
| LAHF | Load AH with Flags | 1.4.1. |
| LDS | Load Pointer into DS | 1.4.1. |
| LEA | Load Effective Address | 1.4.1. |
| LES | Load pointer into ES | 1.4.1. |
| LOCK | Lock bus | 1.4.5. |
| LODS | Load Byte or Word (of string) | 1.4.3. |
| LOOP | Loop | 1.4.4. |
| LOOPE | Loop While Equal | 1.4.4. |
| LOOPNE | Loop While Not Equal | 1.4.4. |
| LOOPNZ | Loop While Not Zero | 1.4.4. |
| LOOPZ | Loop While Zero | 1.4.4. |
| MOV | Move | 1.4.1. |
| MOVS | Move Byte or Word (of sring) | 1.4.3. |
| MUL | Multiply | 1.4.2. |
| NEG | Negate | 1.4.2. |
| NOT | Not | 1.4.2. |
| OR | Or | 1.4.2. |
| OUT | Output Byte or Word | 1.4.1. |
| POP | Pop | 1.4.1. |
| POPF | Pop Flags | 1.4.1. |
| PUSH | Push | 1.4.1. |
| PUSHF | Push Flags | 1.4.1. |
| RCL | Rotate through Carry Left | 1.4.2. |
| RCR | Rotate through Carry Right | 1.4.2. |
| REP | Repeat | 1.4.3. |
| RET | Return (intra segment) | 1.4.4. |
| RETF | Return (inter segment) | 1.4.4. |
| ROL | Rotate Left | 1.4.2. |
| ROR | Rotate Right | 1.4.2. |

Tabelle 32: (Fortsetzung)

| Mnemonik | Beschreibung | Abschnitt |
|---|---|---|
| SAHF | Store AH into Flags | 1.4.1. |
| SAL | Shift Arithmetic Left | 1.4.2. |
| SAR | Shift Arithmetic Right | 1.4.2. |
| SBB | Shift with Borrow | 1.4.2. |
| SCAS | Scan Byte or Word (of string) | 1.4.3. |
| SHL | Shift Left | 1.4.2. |
| SHR | Shift Right | 1.4.2. |
| STC | Set Carry | 1.4.5. |
| STD | Set Direction | 1.4.5. |
| STI | Set Interrupt | 1.4.5. |
| STOS | Store Byte or Word (of string) | 1.4.3. |
| SUB | Subtract | 1.4.2. |
| TEST | Test | 1.4.2. |
| WAIT | Wait | 1.4.5. |
| XCHG | Exchange | 1.4.1. |
| XLAT | Translate | 1.4.1. |
| XOR | Exclusive Or | 1.4.2. |

## Anlage 6 Code-Makro-Definitionssyntax

```
codemacro ::= CODEMACRO name [formal-list]
              [list-of-macro-directives]
              ENDM

name ::= IDENTIFIER

formal-list ::= parameter-description[{,parameter-description}]

parameter-description ::= formal-name :specifier-letter
                          modifier-letter[(range)]

specifier-letter ::= A | C | D | E | M | R | S | X

modifier-letter ::= b | w | d | sb

range ::= single-range|double-range

single-range::=REGISTER | NUMBERB

double-range ::= NUMBERB,NUMBERB | NUMBERB,REGISTER |
                 REGISTER,NUMBERB | REGISTER,REGISTER

list-of-macro-directives ::= macro-directive
                             {macro-directive}

macro-directive ::= db | dw | dd | segfix |
                    nosegfix | modrm | relb | relw | dbit

db ::= DB NUMBERB | DB formal-name

dw ::= DW NUMBERW | DW formal-name

dd ::= DD formal-name

segfix ::= SEGFIX formal-name
```

```
nosegfix ::= NOSEGFIX formal-name

modrm::= MODRM NUMBER7, formal-name
         MODRM formal-name , formal-name

relb ::= RELB formal-name

relw ::= RELW formal-name

dbit ::= DBIT field-description{,field-description}

field-desciption ::= NUMBER15 ( NUMBERB) |
                     NUMBER15 ( formal-name ( NUMBERB ))

formal-name ::= IDENTIFIER
```

NUMBERB ist ein 8-Bit-Wert.
NUMBERW ist ein 16-Bit-Wert.
NUMBER7 sind Werte von 0, 1, ..., 7.
NUMBER15 sind Werte von 0, 1, ..., 15.

## Anlage 7 ASM86-Fehlermitteilungen

ASM86 liefert zwei Fehlertypen:

1. Fehler mit nachfolgendem Abbruch der Übersetzung:

   NO FILE
   DISK FULL
   DIRECTORY FULL
   DISK READ ERROR
   CANNOT CLOSE
   SYMBOL TABLE OVERFLOW
   PARAMETER ERROR

2. Fehler, die während der Analyse der Assemblerzeile auftreten. Diese Fehler werden durch Ausgabe einer Nummer vor der Assemblerzeile gekennzeichnet. Wenn eine Zeile mehr als einen Fehler enthält, so wird nur der erste Fehler angezeigt. Tabelle 33 führt die ASM86-Fehlermitteilungen auf.

Tabelle 33: ASM86-Fehlermitteilungen

| Nummer | Bedeutung |
|---|---|
| 0 | ILLEGAL FIRST ITEM<br>(ungültiges erstes Element auf einer Quellzeile) |
| 1 | MISSING PSEUDO INSTRUKTION<br>(Pseudobefehl fehlt) |
| 2 | ILLEGAL PSEUDO INSTRUCTION<br>(unerlaubter Pseudobefehl) |
| 3 | DOUBLE DEFINED VARIABLE<br>(mehrfach definierte Variable) |
| 4 | DOUBLE DEFINED LABEL<br>(mehrfach definierte Marke) |
| 5 | UNDEFINED INSTRUCTION<br>(undefinierter Befehl) |
| 6 | GARBAGE AT END OF LEINE - IGNORED<br>(unklares Zeilenende - wird ignoriert) |
| 7 | OPERAND(S) MISMATCH INSTRUCTION<br>(Operand paßt nicht zum Befehl) |
| 8 | ILLEGAL INSTRUCTION OPERANDS<br>(unerlaubte Befehlsoperanden) |
| 9 | MISSING INSTRUCTION<br>(Befehl nicht vorhanden) |

Tabelle 33: (Fortsetzung)

| Nummer | Bedeutung |
|---|---|
| 10 | UNDEFINED ELEMENT OF EXPRESSION<br>(undefiniertes Ausdruckselement) |
| 11 | ILLEGAL PSEUDO OPERAND<br>(unerlaubter Pseudooperand) |
| 12 | NESTED IF ILLEGAL - IF IGNORED<br>(maximale Schachtelung mit IF überschritten) |
| 13 | ILLEGAL IF OPERAND - IF IGNORED<br>(unerlaubter IF-Operand) |
| 14 | NO MATCHING IF FOR ENDIF<br>(keine Übereinstimmung zwischen IF und ENDIF) |
| 15 | SYMBOL ILLEGALED FORWARD REFERENCED - NEGLECTED<br>(unerlaubte Vorwärtsreferenz in ORG, RS, EQU oder IF) |
| 16 | DOUBLE DEFINED SYMBOL - TREATED AS UNDEFINED<br>(mehrfach definiertes Symbol) |
| 17 | INSTRUCTION NOT IN CODESEGMENT<br>(Befehl nicht im Code-Segment) |
| 18 | FILE NAME SYNTAX ERROR<br>(Dateiname in INCLUDE-Direktive syntaktisch falsch) |
| 19 | NESTED INCLUDE NOT ALLOWED<br>(geschachtelte Einbindung nicht erlaubt) |
| 20 | ILLEGAL EXPRESSION ELEMENT<br>(unerlaubtes Ausdruckselement) |
| 21 | MISSING TYPE INFORMATION IN OPERAND(S)<br>(Typangabe im Operanden fehlt) |
| 22 | LABEL OUT OF RANGE<br>(Marke außerhalb der zulässigen Grenzen) |
| 23 | MISSING SEGMENT INFORMATION IN OPERAND<br>(Segmentangabe im Operanden fehlt) |
| 24 | ERROR IN CODEMACROBUILDING<br>(Fehler bei der Code-Makro-Bildung) |

## Anlage 8 DDT86-Fehlermitteilungen

Tabelle 34: DDT86-Fehlermitteilungen

| Fehlermitteilung | Bedeutung |
|---|---|
| AMBIGOUS OPERAND | Mehrdeutiger Operand. Zu einem A-Kommando wurde ein mehrdeutiger Operand angegeben. Der Operand muß durch BYTE oder WORD spezifiziert werden. |
| CANNOT CLOSE | Datei kann nicht geschlossen werden. Eine Plattendatei, die durch ein W-Kommando geschrieben wurde, kann nicht geschlossen werden. |
| DISK READ ERROR | Plattenfehler. Eine Plattendatei, die in einem R-Kommando spezifiziert wurde, kann nicht gelesen werden. |
| DISK WRITE ERROR | Plattenschreibfehler. Eine Schreiboperation auf die Platte während eines W-Kommandos kann nicht richtig ausgeführt werden. Vermutlich ist die Platte voll. |
| INSUFFICIENT MEMORY | Unzureichender Speicher. Es ist nicht genügend zusammenhängender Speicher vorhanden, um eine Datei durch ein R- oder E-Kommando zu laden. |
| MEMORY REQUEST DENIED | Speicheranforderung abgelehnt. Eine Speicheranforderung während eines R-Kommandos wurde abgelehnt. Es können nur bis zu sieben Speicherbereiche unter DDT86 gleichzeitig verwaltet werden. |
| NO FILE | Keine Datei. Die in einem R- oder E-Kommando spezifizierte Datei kann nicht auf der Platte gefunden werden. |
| NO SPACE | Kein Speicherbereich. Ein W-Kommando kann nicht ausgeführt werden, weil das Dateiverzeichnis voll ist. |
| VERIFY ERROR s:o | Der durch ein F-, S-, M- oder A-Kommando eingegebene Wert kann nicht korrekt rückgelesen werden. Das kann von einem fehlerhaften RAM herrühren oder es wurde versucht auf einen ROM zu schreiben oder der Speicher existiert nicht im angegebenen Speicherbereich. |

Sachwortverzeichnis