ANWENDER-DOKUMENTATION	Sprachbeschreibung	MOS
11/87	Assembler as	MUTOS 1700

Programmtechnische Beschreibung Teil 2

Sprachbeschreibung

Assembler as

AC A 7100/7150

VEB Robotron-Projekt Dresden

C 8016-0001-1 M3030

Ausgabe: 11/87

Die Ausarbeitung dieser Dokumentation erfolgte durch ein Kollektiv der
TH Ilmenau im Auftrage des VEB Robotron-Elektronik Dresden
Stammbetrieb des VEB Kombinat Robotron.

Im Interesse einer ständigen Weiterentwicklung werden alle Leser gebeten, Hinweise zur Verbesserung der Dokumentation dem Herausgeber mitzuteilen.

Herausgeber:

VEB Robotron-Projekt Dresden
Leningrader Str. 9
Dresden 8010

Kurzreferat

As ist ein ganz einfacher Assembler ohne die Fähigkeit, Makros zu verarbeiten. Er erzeugt ein Ausgabe-File, das Verschiebeinformationen und eine vollständige Symboltabelle enthält. Die Ausgabe wird vom Lader ld des Systems MUTOS 1700 akzeptiert, der verwendet wird, um die Ausgabe verschiedener Assemblerläufe zu vereinigen und um Objektprogramme aus Bibliotheken zu holen. Das Format der Ausgabe wurde so gestaltet, daß ein Programm, welches keine Referenzen auf externe Symbole enthält, ohne weitere Bearbeitung ausführbar ist.

1. Einleitung

Der Assembler as erzeugt aus einem Assembler-Quellprogramm ein Zielprogramm, manchmal auch Objektprogramm genannt. Dieses Zielprogramm steht in einem Ausgabe-File, das im wesentlichen aus dem Maschinencode sowie Relocationinformationen und einer vollständigen Symboltabelle besteht. Dieses Ausgabe-File wird vom Lader ld des Systems MUTOS 1700 akzeptiert. Der Lader kann Zielprogramme verschiedener Assemblerläufe verbinden und Objektprogramme aus Bibliotheken holen. Das Format der Ausgabe wurde so gestaltet, daß ein Programm, welches keine Referenzen auf externe Symbole enthält, ohne weitere Bearbeitung abarbeitbar ist. Der Assembler as hat nicht die Fähigkeit, Macros zu verarbeiten. Falls der Anwender Macros wünscht, kann er dafür den Präprozessor des C-Compilers oder das Systemprogramm m4 verwenden.

Zur formalen Darstellung der Assemblersprache wird hier eine erweiterte Backus-Naur-Notation verwendet. Metavariable stehen in spitzen Klammern und werden durch

```
     <Metavariable> : <metalinguistische-Formel>
```

definiert. Dabei bedeuten

:	Definitionszeichen,
<x> | <y>	Auswahl einer der syntaktischen Konstruktionen <x> oder <y> (Alternative). Steht die Auswahl in geschweiften Klammern, dann ist genau eine Alternative auszuwählen.
[<x>]	Die Konstruktion <x> kann einmal wahlweise auftreten oder auch nicht.
<x> .	Die Konstruktion <x> kann mehrfach auftreten.

Morpheme (synonym auch Terminalsymbole oder Token genannt) werden fettgedruckt.

Einzelheiten des AC A 7100 werden hier nicht erklärt (siehe Technische Beschreibung).

Diese Dokumentation ist nicht für das Erlernen der Assemblerprogrammierung oder als Anleitung zum Schreiben von Assemblerprogrammen vorgesehen. Letzteres wäre ein Verstoß gegen die Philosophie des Systems MUTOS 1700, nämlich möglichst alle Programme in der Programmiersprache C oder einer anderen höheren Programmiersprache zu schreiben.

2. Benutzung

Der Assembler aaasss ist mit folgendem Kommando zu benutzen:

```
as [-LWDT][-o <output>] <file>.
```

Die wahlweise angebbaren Kommandoparameter haben folgende Bedeutungen:

Wenn -L angegeben wird, werden Marken, die mit dem Buchstaben L beginnen, in die vom Assembler ausgegebene Symboltabelle eingetragen. Wird kein -L angegeben, werden diese Marken nur intern vom Assembler geführt und sie erscheinen nicht in der Symboltabelle. Die vom C-Compiler erzeugten, für das File lokalen Marken, beginnen alle mit L.

Mit -W werden alle Fehlermeldungen des Assemblers ausgeschaltet.

Mit -D und -T können Debug- und Trace-Informationen nach einer entsprechenden C-Compilierung des Assemblers selbst verfügbar gemacht werden.

Die Eingabe in den Asembler erfolgt entweder über die Standardeingabe oder es werden die durch <file>. angegebenen Files miteinander verkettet und dann eingegeben. Damit besteht die Möglichkeit, Programme in mehreren getrennten Teilen zu schreiben.

Mit -o <output> wird veranlaßt, daß die Ausgabe in das File <output> erfolgt. Wird diese Option nicht angegeben, erfolgt die Ausgabe in das File a.out der aktuellen Directory. Wenn es keine unaufgelösten Referenzen nach der Assemblierung gibt und keine Fehler erkannt wurden, wird das ausgegebene File (<output> oder a.out) mit dem Attribut ausführbar (executable) markiert, anderenfalls als nicht ausführbar.

3. Lexikalische Festlegungen

Ein Assemblerprogramm besteht aus lexikalischen Einheiten. Die lexikalischen Einheiten des Assemblers, die kurz Token oder Morpheme genannt werden, sind Konstanten, Operatoren und Bezeichnungen. Die Bezeichnungen werden synonym auch Symbole oder Namen genannt.

3.1. Bezeichnungen

Eine Bezeichnung (Identifier) besteht aus einer Folge von alphanumerischen Zeichen (einschließlich der Zeichen Punkt (.) und Unterstreichung (_)), von denen das erste keine Ziffer sein darf. Nur die ersten acht Zeichen sind signifikant. Bezeichner, die mit dem Buchstaben L beginnen, werden als lokale Marken behandelt. Falls die Option -L im Kommando as angegeben wird, werden diese Bezeichnungen wie alle anderen Bezeichnungen behandelt.

3.2. Konstanten

Als Konstanten stehen Integerkonstanten, Realkonstanten und Stringkonstanten zur Verfügung.

Die Integerkonstanten sind Dezimalkonstanten, Oktalkonstanten, Hexakonstanten und Einzeichenkonstanten. Die Integerkonstanten werden in einem 16-Bit-Wort in Zweierkomplementnotation dargestellt. Bei den Realkonstanten werden Float- und Double-Konstanten unterschieden.

Eine Oktalkonstante besteht aus einer Folge von Ziffern, die durch die Ziffer 000 oder das Zeichen Tilde (~) eingeleitet wird. Die Dezimalziffern 8 und 9 werden durch die Oktalzahl 010 bzw. 011 dargestellt.

Eine Dezimalkonstante besteht aus einer Folge von Ziffern, die durch einen Dezimalpunkt (.) beendet werden kann. Sie darf nicht mit der Ziffer 0 beginnen. Der Wert der Konstanten muß mittels 15 Bits darstellbar sein, d.h. er muß kleiner als dezimal 32768 sein.

Eine Hexakonstante besteht aus einer Folge der Hexa-Ziffern 0 bis 9 und a bis f bzw. A bis F, die durch 0x oder / eingeleitet wird. Die Hexa-Ziffern A bis F bzw. äquivalent a bis f haben dezimal die Werte 10 bis 15.

Eine Einzeichenkonstante besteht aus einem Anführungsstrich (') und einem ASCII-Zeichen. Die bei C üblichen Ersatzzeichen, die mit inverser Schrägstrich (Backslash) beginnen, können nicht als Einzeichenkonstante verwendet werden.

Eine Float-Konstante beginnt mit den Zeichen 0f oder 0F, denen Zeichen folgen, die atof als Floating-Point-Zahl akzeptiert. Eine Double-Konstante beginnt im Unterschied zu einer Floating-Point-Zahl mit den Zeichen 0d bzw. 0D. Als Kennzeichen für den Exponenten werden bei Float-Konstanten e oder E und bei Double-Konstanten d oder D verwendet.

Eine Stringkonstante besteht aus einer Folge von ASCII-Zeichen, die in Anführungszeichen (") notiert wird. Ersatzzeichenfolgen, die mit Backslash eingeleitet werden und dieselben wie bei C sind, können verwendet werden, um folgende nichtgraphische Zeichen darzustellen:

```
\n   NL    (036)
\s   SP    (040)
\t   HT    (011)
\e   EOT   (004)
\0   NUL   (000)
\r   CR    (015)
\a   ACK   (006)
\p   PFX   (033)
\\   \
```

Das letzte Zeichen wird benötigt, um das Zeichen Backslash selbst angeben zu können. Der Wert der Stringkonstanten besteht aus dem Code der angegebenen Zeichen, die paarig byteweise in Worten abgespeichert werden. Das Ende der Stringkonstanten wird durch die voranstehende Pseudooperation bestimmt (siehe Abschnitt <Direktiven>, .ascii und .asciz).

3.3. Operatoren

Es gibt verschiedene Operatoren, die durch ein oder zwei Zeichen dargestellt werden, siehe Abschnitt <Ausdrücke>.

3.4. Abstände

Zwischen den Tokens dürfen `beliebig viele' Leerzeichen und/oder Tabulatorzeichen stehen. Ausgenommen dabei sind Token, die Zeichenkonstanten darstellen. Bezeichnungen oder Konstanten, die nebeneinander stehen, müssen durch Leer- oder Tabulatorzeichen getrennt werden, falls sie nicht durch andere Zeichen getrennt sind.

3.5. Kommentare

Es gibt verschiedene Möglickeiten für Kommentare.

Das Zeichen | leitet einen Kommentar ein, der sich bis zum Zeilenende erstreckt. Folgt dem Strich ein weiterer, also ||, wird diese lexikalische Einheit als Operationszeichen ODER analysiert. Kommentare werden vom Assembler ignoriert.

Kommentare können im C-Stil geschrieben werden, die sich über mehrere Zeilen erstrecken können. Sie bestehen aus Zeichenfolgen, die in /* und */ eingeschlossen werden. Diese Kommentare werden vom Assembler ignoriert.

Mit dem Zeichen # auf der ersten Position in einer Zeile kann ein spezieller Kommentar eingeleitet werden, der bis zum Zeilenende gilt. Dieser Kommentar hat das Format

```
        ###<Ausdruck> <String>
```

Vom Assembler wird dieser Kommentar so interpretiert, daß er sich gerade bei der Assemblierung in der Zeile <Ausdruck> des Files <String> befindet. Solche Zeilen werden z.B. vom C-Präprozessor erzeugt.

4. Segmente und Speicherplatzzähler - Zielprogrammstruktur

Das vom Assembler erzeugte Zielprogramm besteht aus Kopf, Segmenten, Relocationinformationen und Symboltabelle. Der Kopf hat die Structur exec, die im File a.out.h deklariert wird (siehe a.out(5)). Dieser Kopf enthält alle Längenangaben für die im Zielprogramm folgenden Informationen. Die Segmente sind das Textsegment und das Datensegment. Sie stehen in dieser Reihenfolge im Zielprogramm. Das Textsegment wird manchmal auch Codesegment genannt.

Wozu dienen Text- und Datensegment ? Das Operationssystem bietet die Möglichkeit, daß beim Mehrnutzerbetrieb eines Programms ein bestimmter Teil dieses Programms, nämlich das Textsegment, von allen gemeinsam genutzt wird. Diese Nutzungsart setzt voraus, daß das betreffende Programm mit der Option -n geladen wurde. Damit wird erreicht, daß das Textsegment schreibgeschützt ist. Das Datensegment eines derartigen Programms wird für jeden Nutzer getrennt angelegt. Beispiel: Wollen mehrere Nutzer unter Verwendung desselben Editors editieren und sei dieser Editor ein schreibgeschüztes Programm, dann wird das Textsegment des Editors nur einmal im Hauptspeicher stehen und jeder Nutzer hat sein eigenes Datensegment für den Editor, in dem seine zu editierenden Daten stehen. Aus der Sicht des Assemblers unterscheiden sich Text- und Datensegment nur dadurch, daß sie getrennt anzulegen sind. Es können Befehle und Daten in beiden Segmenten stehen. Im Textsegment befinden sich gewöhnlich die auszuführenden Befehle. Im Datensegment stehen zunächst die initialisierten Daten. Die nichtinitialisierten Daten werden in dem bss-Segment untergebracht. Zur Laufzeit des Programms wird das Datensegment um das bss-Segement erweitert. Die dazu erforderlichen Informationen stehen in der Symboltabelle und im Kopf des Zielprogramms.

Für jedes Segment führt der Assembler einen Speicherplatzzähler. Der Speicherplatzzähler (location counter) wird mit dem Zeichen Punkt (.) symbolisiert. Der Wert des Speicherplatzzählers ist zu jeder Zeit der Abstand vom Anfang des betreffenden Segments (synonym: Offset, Verschiebung, relative Adresse), wo der nächste Befehl oder das nächste Datum bei der Assemblierung platziert werden soll. Dem Speicherplatzzähler kann ein Wert zugewiesen werden. Hierbei darf sich der aktuelle Wert nicht verkleinern. Wenn durch die Zuweisung der Wert von dem Wert Null generiert. Typischerweise wird der Speicherplatzzähler für das bss-Segment durch z.B. folgende Anweisungen, die untereinander äquivalent sind, um 5 Bytes weitergezählt:

```
          .lcomm marke, 5
          .bss
           marke :.blkw 5
           marke :.blkb 10
           marke :. = . + 10
```

5. Quellprogramm- und Anweisungstruktur

Ein Quellprogramm besteht aus einer Folge von Anweisungen. Die Anweisungen werden entweder durch Newlines oder Semikolons getrennt. Es gibt verschiedene Arten von Anweisungen. Anweisungen, die durch <Mnemonik> gekennzeichnet sind, haben folgende Syntax:

[<Marke>].[<Präfix>]<Mnemonik>[<Operand>].[<Kommentar>]

Neben dieser Art von Anweisungen gibt es noch Zuweisungen mit folgender Syntax:

[<Marke>].<Bezeichnung> = <Ausdruck>

Jeder Anweisung können eine <Marke> oder auch mehrere <Marken> voranstehen. Die durch <Memonik> gekennzeichneten Anweisungen lassen sich in zwei Klassen einteilen: Befehle und Direktiven (Pseudo-Befehle). Nur bei Befehlen kann ein <Präfix> stehen. Ob in der Anweisung ein oder mehrer <Operanden> möglich sind, hängt vom <Mnemonik> ab. Die <Operanden> werden durch Komma getrennt.

5.1. Marken

Es gibt zwei Arten von <Marken>: Namenmarken und Zahlenmarken. Die Definition einer Namenmarke besteht aus einem <Namen>, dem ein Doppelpunkt : folgt. Der <Name> symbolisiert den aktuellen Wert des Speicherplatzzählers an der Stelle der Definition im Programm. Im Pass 1 wird ein Fehler gemeldet, wenn der Name schon an anderer Stelle als Namenmarke definiert wurde.

Die Definition einer Zahlenmarke besteht aus einer <Ziffer>, 0 bis 9, der ein Doppelpunkt folgt. In einem Programm dürfen an verschiedenen Stellen dieselben Ziffern als Zahlenmarken dienen. Sie werden als lokale <Marken> verwendet. Auf eine solche Zahlenmarke kann man sich mittels <Ziffer> b auf die davor stehende (backward) und mittels <Ziffer> f auf die folgende (forward) beziehen, und zwar auf die am nächstliegende Zahlenmarke <Ziffer>. Durch die Verwendung von Zahlenmarken spart man Platz in der Symboltabelle.

5.2. Präfixe und Mnemoniks

Die <Präfixe> und <Mnemoniks> für Befehle stehen in den Tabellen 1 und 2. Die <Mnemoniks> für Direktiven werden im nächsten Kapitel angegeben.

Die <Mnemoniks> sind für den Assembler reservierte <Bezeichnungen>. Sie kennzeichnen die Befehle bzw. Direktiven. Die Syntax der Anweisung hängt im einzelnen vom <Mnemonik> ab. Alle <Mnemoniks> werden weiter unten mit der geforderten Syntax angegeben.

<Präfixe> sind spezielle Befehle, die vor allem bei Stringbefehlen verwendet werden.

5.3. Leere Anweisung

Eine leere Anweisung ist eine Anweisung, in der es keine <Präfixe>, keine <Mnemoniks> und keine <Operanden> gibt. Die Leeranweisung wird vom Assembler ignoriert.übliche Leeranweisungen sind leere Zeilen oder Zeilen, die nur Kommentare enthalten.

5.4. Zuweisungen

Eine Zuweisung besteht aus einer <Bezeichnung>, einem Gleichheitszeichen = und einem <Ausdruck>. Der Wert und der Typ des <Ausdrucks> werden der <Bezeichnung> zugewiesen. Der Punkt, welcher den Speicherplatzzähler symbolisiert, wird als <Bezeichnung> betrachtet.

Die Eigenschaft extern eines <Ausdrucks> geht bei einer Zuweisung verloren. Es ist also nicht möglich, durch eine Zuweisung ein globales Symbol zu deklarieren. Ebenfalls ist es unmöglich, ein Symbol als Offset eines nicht lokal definierten globalen Symbols zu definieren.

Wie bereits erwähnt wurde, kann man dem Speicherplatzzähler etwas zuweisen. Dabei ist es erforderlich, daß der Typ des zugewiesenen <Ausdrucks> derselbe wie der von . ist, und daß es verboten ist, den Wert von . zu vermindern. Praktisch hat eine solche Zuweisung meist die Form

```
.== . +<ganze_Zahl>.
```

Die Wirkung dieser Zuweisung ist, daß soviele Bytes mit dem Wert Null generiert werden, wie die <ganze_Zahl> angibt.

6. Ausdrücke

Arithmetische <Ausdrücke> werden als <Operanden> in Anweisungen verwendet. Ein arithmetischer <Ausdruck> verknüpft mittels den nachfolgend angegebenen Operatoren Werte von Bezeichnungen und Zahlenkonstanten. (Mnemoniks von Befehlen ergeben den Operationscode, Register ihren internen Code im Assembler.) Ein <Ausdruck> ist eine Folge von Symbolen, die einen Wert darstellen. Seine Bestandteile sind Bezeichnungen, insbesondere Marken, ferner Konstanten, Operatoren und Klammern. Jeder <Ausdruck> hat einen Typ.

Alle Operatoren in <Ausdrücken> sind zweistellig. Desweiteren sind einstellige Operatoren: +, - und !. (Der Assembler verarbeitet diese Operatoren aber trotzdem als zweistellige Operatoren, indem der erste <Operand> als eine Null vom Typ absolut gesetzt wird. - Kann bei Typfortpflanzung und bestimmten Fehlermeldungen von Interesse sein !) Die Arithmetik ist eine Zweierkomplementarithmetik und hat eine Genauigkeit von 16 Bit. Alle Operatoren besitzen den gleichen Vorrang. Die <Ausdrücke> werden von links nach rechts ausgewertet, außer beim Vorhandensein von eckigen Klammern.

6.1. Ausdrucks-Operatoren

Die Operatoren sind:

+ Addition

- Subtraktion

* Multiplikation

/ Division (beachte, dass `/' eine Hexakonstante einleitet)

& bitweises UND

|| bitweises ODER

> (oder >>)logische Rechtsverschiebung
< (oder <<)logische Linksverschiebung

% Modulo

! a ! b bedeutet a oder (nicht b); d.h., der erste <Operand> wird mit dem Einerkomplement des zweiten <Operanden> verknüpft; am häufigsten unär gebraucht (ergibt Einerkomplement bzw. bitweise Negation).

^ bitweise Antivalenz (synonym: ausschliessendes bzw. exklusives ODER)

<Ausdrücke> können durch Verwendung von eckigen Klammern `[]' gruppiert werden (runde Klammern sind für Adressierungszwecke reserviert).

6.2. Typen

Den Morphemen Bezeichnung und Konstante, ohne Stringkonstante, wird vom Assembler eine Eigenschaft zugeordnet, die als sogenannter Typ kodiert wird. Bei der Auswertung von arithmetischen <Ausdrücken> während der Assemblierung überträgt sich der Typ der <Operanden> nach bestimmten Regeln auf den Wert des <Ausdrucks>. Der Typ dient einerseits im Assembler der Fehlererkennung und andererseits zur übermittlung von Informationen an den Lader. Der Assembler trägt den Typ in der Symboltabelle und bei bestimmten Relocationinformationen ein. Anmerkung: Stringkonstante und Bezeichner von Pseudobefehlsmnemoniks erhalten keinen Typ.

Die Typen sind: absolut, undefiniert, text, data, bss, extern, extern-undefiniert, extern-text, extern-data und extern-bss. Sie haben folgende Bedeutungen:

absolut
: Die Konstanten ohne Stringkonstanten, Mnemoniks für Befehle, Direktiven und Register haben immer den Typ absolut. Dieser Typ bedeutet, daß das betreffende Morphem im Zielprogramm durch einen während der Assemblierung bekannten Wert, z.B. Maschinencode, dargestellt wird. Der Wert bleibt vom Lader unverändert.

text | data | bss
: Die Sorten der Segmente sind: text, data und bss. Bezeichner von Marken erhalten als Typ die Sorte des Segmentes, in dem sie definert werden. (Zahlenmarken werden im Assembler in lokale Namenmarken umgewandelt.) Marken haben an der Stelle der Definition im Segment den Wert des Speicherplatzzählers. Für Segmente einer Segmentsorte wird ein zur Segmentsorte gehörender Speicherplatzzähler geführt, der mit Null beginnt. Der Wert einer Marke kann sich beim Laden mehrer Files zu einem Programm verändern (durch Addition der jeweiligen Anfangsadresse).

undefiniert
: Marken, die im gerade zu assemblierenden File nicht definiert werden, erhalten als Typ undefiniert. Während des ersten Assemblerdurchlaufs (Pass) sind bei Sprüngen zu einer nachfolgenden Marke diese Marken beim erstmaligen Auftreten undefinert. Erst nach dem Passieren der Markendefinition oder einer .globl-Deklaration, ist der Typ der Marke bekannt. Während des zweiten Assemblerdurchlaufs darf ein auszuwertender <Ausdruck> nicht den Typ undefiniert haben.

undefiniert-extern
: Mit .globl wird einer Bezeichnung der Typ extern zugewiesen. Die Bezeichnung hat den Typ undefiniert-extern, wenn die Bezeichnung in dem zu assemblierenden File keinen Wert erhält, entweder als definerte Marke oder durch Zuweisung. Gibt es in einem Zielprogramm Morpheme vom Typ undefiniert, muß dieses Zielprogramm mit anderen mittels llldddd geladen werden, in denen letztlich die betreffenden Morpheme definiert werden.

extern-text | extern-data | extern-bss
: Wenn einer Bezeichnung der Typ extern mittels .globl zugewiesen wurde und die betreffende Bezeichnung als Marke in dem vorliegenden File definiert wird oder durch Zuweisung einen Wert erhält, erhält sie desweiteren auch noch den Typ infolge der Segmentsorte, siehe oben: text | data | bss.

Die Kodierungen für die Typen sind im Programmierhandbuch bei a.out(5) bzw. im File a.out.h angegeben.

6.3. Typ-Fortpflanzung in Ausdrücken

Verknüpft man <Operanden> durch Ausdrucksoperatoren, so hat das Ergebnis einen Typ, der von den Typen der <Operanden> und Operatoren abhängt. Die Verknüpfungsregeln sind hierfür:

- Wenn einer der <Operanden> den Typ undefiniert hat, so hat das Ergebnis den Typ undefiniert.

- Wenn beide <Operanden> den Typ absolut haben, so hat das Ergebnis den Typ absolut.

Weitere Regeln, die bei den Operatoren + und - wirksam werden, sind:

+ Wenn ein <Operand> den Typ text, data, bss oder undefiniert-extern hat und der andere den Typ absolut, so hat das Ergebnis den Typ text, data, bss bzw. undefiniert-extern.
- Wenn der erste <Operand> den Typ text, data oder bss hat und der zweite <Operand> hat denselben Typ wie der erste, hat das Ergebnis den Typ absolut. Wenn der erste <Operand> undefiniert-extern ist, muß der zweite absolut sein. Alle übrigen Kombinationen von Typen sind unzulässig.

7. Direktiven

In diesem Abschnitt werden die Direktiven mit ihren Parametern angegeben. Der Unterschied zwischen Befehlen und Direktiven besteht darin, daß Befehle im Zielprogramm durch einen Befehlscode und den codierten Operanden bzw. deren Adressen dargestellt werden, während Direktiven, die auch Pseudo-Befehle oder Pseudo-Operationen genannt werden, den Assembler steuern und nicht in jedem Falle eine unmittelbare Entsprechung im Zielprogramm erfahren. Die Direktiven sind nachfolgend nach Funktionsgruppen gegliedert.

7.1. Speicherplatzzählersteuerung

7.1.1. Segmentwahl

```
.text
.data
.bss
```

Das jeweilige <Mnemonik> leitet ein Text-, Data- oder bss-Segment ein. Die diesen <Mnemoniks> folgenden Anweisungen werden in das betreffende Segment assembliert. Abschnitte im Assembler-Quellprogramm, die mit derselben Segmentwahl-Direktive eingeleitet werden und mit einer anderen Segmentwahl-Direktive beendet werden, werden bei der Assemblierung in der Reihenfolge verkettet, wie sie im Assembler-Quellprogramm stehen.

7.1.2. .org Ausdruck

Der Speicherplatzzähler wird gleich dem vom <Ausdruck> gelieferten Wert gesetzt. Der Wert des Speicherplatzzählers darf nur vergrössert werden. Eine äquivalente Anweisung ist

```
. = . +<Ausdruck>.
```

7.1.3. .even

Wenn der Speicherplatzzähler ungerade ist, wird er um Eins weitergezählt und die nächste Anweisung kann ab einer Wortgrenze assembliert werden.

7.2. Initialisierte Daten

7.2.1. Durch Ausdruck initialisierte Daten

```
.byte <Ausdruck> [ , <Ausdruck> ] ...
.word <Ausdruck> [ , <Ausdruck> ] ...
.int <Ausdruck> [ , <Ausdruck> ] ...
.long <Ausdruck> [ , <Ausdruck> ] ...
.float <Ausdruck> [ , <Ausdruck> ] ...
```

```
     .double <Ausdruck>     [ , <Ausdruck> ] ...
```

Die <Ausdrücke>, die in der durch Kommata getrennten Liste stehen, werden auf den angegebenen Umfang beschnitten, d.h. also bei .byte = 8 Bits, bei .word und anderfolgende Plätze assembliert. Die Ausdrücke müssen bei Abarbeitung des zweiten Passes absolut sein.

7.2.2. Mit Zeichenketten initialisierte Daten

```
     .asciz <String> [ , <String> ] ...
     .ascii <String> [ , <String> ] ...
```

Die Zeichen der Zeichenkette <String> werden in (den Adressen nach) aufeinanderfolgende Bytes gespeichert. Bei .asciz wird die Zeichenette mit Null abgeschlossen und bei .ascii nicht. Werden mehrere <Strings> angegeben, werden diese verkettet. Bei .asciz wird erst nach der Verkettung eine Null angefügt. Die Ersatzzeichen, die von C bekannt sind, werden akzeptiert (siehe Lexikalische Festlegungen).

7.2.3. Mit Null initialisierte Daten

```
     .space [ <Ausdruck> ]
     .blkb [ <Ausdruck> ]
     .blkw [ <Ausdruck> ]
```

Bei .space und .blkb bzw. .blkw werden soviele Bytes bzw. Worte mit Null assembliert, wie der <Ausdruck> angibt. Wird kein <Ausdruck> angegeben, wird ein Byte bzw. ein Wort mit Null assembliert.

7.3. Symboldefinition

7.3.1. .globl Name

Dieses Statement kennzeichnet den <Namen> als externes Symbol. Gibt es mehrere Files, in denen ein bestimmter <Name> (oder auch mehrere) mittels .globl als externe Symbole gekennzeichnet werden und wird in einem dieser Files dieser <Name> (oder diese Namen, nicht notwendig im selben File) definiert, d.h. er tritt als Marke oder mit dem Pseudo .set auf, dann sorgt der Lader dafür, daß über den gleichen <Namen> die eine Stelle der Definition addressiert wird.

<Namen>, die in der laufenden Assemblierung nicht definiert wurden, werden als undefiniert-extern gekennzeichnet.

7.3.2. .set Name, Ausdruck

Das Paar <Name>, <Ausdruck> wird in die Symboltabelle eingetragen. Dasselbe kann man mit einer Zuweisung erreichen:

```
        <Name> = <Ausdruck>.
```

7.3.3. .comm Name, Ausdruck

<Name> bekommt den Typ undefiniert-extern und den Wert des <Ausdrucks>. Der Lader ld ist so realisiert, daß alle undefiniert-externen Symbole, deren Wert ungleich Null ist, im bss-Segment plaziert werden. Hinter dem Symbol wird für soviele Bytes Platz gelassen, wie durch den <Ausdruck> angegeben werden.

7.3.4. .lcomm Name, Ausdruck

Soviele Bytes wie der <Ausdruck> angibt, werden im bss-Segment plaziert. Das erste Byte ist über den <Namen> adressierbar.

8. Befehle

Die Befehle werden in dieser Dokumentation nur aufgezählt und die Adressierungsarten werden kurz behandelt. Beschreibung der einzelnen Befehle sind in z.B. /1/ zu finden.

8.1. Operandenadressierung

Allgemein betrachtet befindet sich ein <Operand> in einem der Register oder auf einem Speicherplatz.

Ein Register wird mit seinem <Namen> adressiert. Man spricht von Registeradressierung.

```
<Operand> : <Registername>
```

Die <Namen> für Register sind reserviert. Folgende <Registernamen> sind verwendbar:

```
ax  bx  cx  dx  sp  bp  si  di
al  bl  cl  dl  ah  bh  ch  dh
cs  ds  ss  es
```

In der nachfolgenden Erklärung werden mit X, Y und Z Hexaziffern symbolisiert. Bei der Speicheradressierung wird die physische Adresse eines Speicherplatzes, die auch absolute Adresse genannt wird (X Z Z Z Y), vom Prozessor durch Addition der Segmentbasisadresse plus der effektiven Adresse (Y Y Y Y) gebildet. Die Segmentbasisadresse (X X X X 0) ist der mit 16 multiplizierte Inhalt des zutreffenden Segmentregisters (X X X X). (Segmente können nur an durch 16 dividierbaren absoluten Adressen beginnen, sogenannte Paragraph-Adressen.) Schematisch kann dieser Sachverhalt in einer Hexazahlendarstellung wie folgt veranschaulicht werden:

```
Segmentbasisadresse     X X X X 0
effektive Adresse    +  0 Y Y Y Y
                        ------------
absolute Adresse        X Z Z Z Y
```

Für die Bildung der effektiven Adresse stehen folgende Methoden zur Verfügung:

- unmittelbare Adressierung
- direkte Adressierung
- indexierte Adressierung mit oder ohne Displacement
- basisrelative Adressierung zum BX- bzw. BP-Register
- basisrelative und indexierte Adressierung

Bei der unmittelbaren Adressierung wird der <Operand> mit seinem Wert unmittelbar selbst, in Form eines <Ausdrucks>, der im einfachsten Fall eine Konstante ist, angegeben. (Im Zielprogramm wird hinter der Einheit, bestehend aus Operationscodebyte, Modrm-Byte und den Bytes für das <Displacement>, der Wert

unmittelbar gespeichert.) Die Kennzeichnung des <Ausdrucks> mit * und # bedeutet, daß der Wert des <Ausdrucks> in einem Byte bzw. in einem Wort (16 Bits = 2 Bytes) gespeichert werden soll.

```
<Operand> : { * | # } <Ausdruck>
```

Das Befehlsmnemonik bestimmt letztlich, welche Speichereinheit gewählt wird. Beispiel:

```
mov dx, *9
```

bedeutet, daß der unmittelbare Operand "9" in einem Wort gespeichert wird. Hingegen

```
movb dx, *9
```

bewirkt die Speicherung von "9" in einem Byte.

Bei der direkten Adressierung wird im allgemeinen die effektive Adresse in Form eines <Ausdrucks> angegeben. Der Wert dieses <Ausdrucks> gibt die Anzahl der Bytes an, die zur aktuellen Segmentanfangsadresse addiert werden. Man bezeichnet diesen Byteanzahlwert als <Displacement>. Dieser Wert ist gleich dem des Speicherplatzzählers, der ab den Direktiven .text, .data bzw. .bss während der Assemblierung gezählt wird. Wenn der Programmierer durch Befehle den Inhalt der Segmentregister und damit also die Segmentanfangsadresse verändert, wird eine entsprechende absolute Adresse gebildet, die verschieden ist zu der absoluten Adresse, wenn die Segmentanfangsadresse nicht verändert wurde.

```
<Displacement> : <Ausdruck>
<Operand> : <Displacement>
```

Bei der indexierten Adressierung wird die efektive Adresse aus dem Inhalt eines der Indexregister SI oder DI plus einem möglicherweise angegebenen <Displacement> gebildet.

```
<Operand> : [ <Displacement> ] { ( DI ) | ( SI ) }
```

Bei der basisrelativen Adressierung zum BX- oder BP-Register wird die effektive Adresse als Summe aus dem Inhalt eines der angebenen Basisregister (BX oder BP) und den möglicherweise angegebenen <Displacement> und Indexregisterinhalt gebildet. Bei der BP-basisrelativen Adressierung muß ein <Displacement> angegeben werden.

```
<Operand> : [ <Displacement> ] ( BX )
<Operand> : <Displacement> ( BP )
```

Zur Unterscheidung zwischen der Registeradressierung und der indexierten oder basisrelativen Adressierung werden die Register bei den beiden letzten Adressierungsarten in runden Klammern geschrieben.

Bei der basisrelativen und indexierten Adressierung mit oder ohne Displacement werden die Operanden wie folgt gebildet, wobei Basisregister und Indexregister in beliebiger Reihenfolge stehen können:

```
<Operand> : [ <Displacement> ] ( BX ) { ( SI ) | ( DI ) }
<Operand> : [ <Displacement> ] { ( SI ) | ( DI ) } ( BX )
<Operand> : <Displacement> ( BP ) { ( SI ) | ( DI ) }
<Operand> : <Displacement> { ( SI ) | ( DI ) } ( BP )
```

Es besteht durch die Hardware folgende Standardzuordnung zwischen Segment-, Basis- und Indexregistern, falls diese durch den Programmierer nicht mittels des Befehls Segment- Override

seg { CS | DS | SS | ES }

verändert wurde. (Segment-Override sorgt dafür, daß der nachfolgende Befehl zur Adreßbildung das bei sssseeeggg angegebene Segmentregister verwendet.)

Adressierungsbestandteil	Segmentregister
(BX)	DS
(BP)	SS
(DI) | (SI)	DS
(BX)(SI) | (BX)(DI)	DS
(BP)(SI) | (BP)(DI)	SS

Ausnahmen bestehen für die gerade angebenen Standardzuordnungen bei den Befehlen push, pop, call, ret, int und iret, die immer SS als Segmentregister verwenden. Die Stringbefehle verwenden immer ES als Segmentregister für Operanden, die mittels DI adressiert werden.

Die Adressierungsmöglichkeiten sind nicht uniform auf alle Befehle anwendbar. Bei der Auflistung der Befehle werden die zulässigen Adressierungsarten angegeben. Dazu werden folgende Abkürzungen verwendet:

A	allgemeine Adresse (Speicheradressierung oder Registeradressierung)
D	direkte Adressierung
E	irgendein Ausdruck
M	Speicheradressierung ohne unmittelbare Adressierung
P	Portadresse
R	Registeradressierung (ohne Segmentregister)
S	Segmentregister
U	unmittelbare Adressierung

Der Umfang wird wie folgt angegeben:

Bei der unmittelbaren Adressierung:

* U	Byte
# U	Wort

und bei allen anderen Adressierungsarten:

+B Byte,
+W Wort
und die entsprechenden Formen des Akkumulators
AL Byte
AX Wort.

Wenn der Operand nur ein bestimmtes Register oder eine bestimmte anzugebende Zahl sein darf, wird er explizit angegeben, z.B. CL, 1 oder 3. Darf ein Wert bei der unmittelbaren Adressierung nur in einem bestimmten Bereich liegen, wird dies durch das Intervall angegeben, z.B. U(0,63). Bei der Angabe von

Segmentregistern mittels S wird durch {NC} das CS-Register von den anderen ausgenommen, d.h. nur DS, SS oder ES sind zugelassen. Mit @ wird eine indirekte Adressierung gekennzeichnet. d : s bedeutet, daß der Offset (neuer PC-Inhalt; d) und Code-Segmentregisterinhalt (s), getrennt durch Doppelpunkt :, anzugeben sind. Diese Werte können 16-Bit-Werte sein.

Bei den Befehlsmnemoniks kennzeichnet ein angefügtes b einen Byte-Befehl. Die Kennzeichnung der unmittelbaren Operanden wird bei derartigen Befehlen unwirksam. Umgekehrt gilt bei Wort-Befehlen, daß der unmittelbare Operand immer in ein Wort assembliert wird.

Ein dem Befehlsmnemonik voranstehendes i kennzeichnet Intersegment-Befehle. Die Kennzeichnungen b bzw. i sind nur bei den in der Befehlstabelle damit angegebenen Befehlen verwendbar.

8.2. Verzweigebefehle

Der Prozessor kennt keine Branch-Befehle. Der vorliegende Assembler simuliert sie. Die Ursache dafür ist folgende:
Bei bedingten Sprüngen muß die Entfernung zwischen Sprungbefehl und dem Sprungziel folgender Bedingung genügen:

```
-257 < Entfernung < +256.
```

Der Assembler stellt Branch-Befehle bereit, bei denen die Entfernung in einem Wort untergebracht wird (Tabelle 2).

Der jeweilige Branch-Befehl wird durch einen passenden bedingten Sprung und einen unbedingten Sprung nachgebildet. Die Bedingung des Branch-Befehls erscheint bei dem bedingten Sprung negiert. Beispiel:

```
beq  lab;   =~  jne  *3; jmp  lab;
```

9. Fehlermeldungen

Die Fehlermeldungen erklären sich selbst. Die Zeilennummer steht dem Fehlermeldungstext voran.

Anlage A Befehlsliste

Befehls-Beschreibung	Befehls-Mnemonik	Anz.	Operanden-Beschreibung	
ascii adjust for addition	aaa	0		
ascii adjust for division	aad	0		
ascii adjust for multiply	aam	0		
ascii adjust for subtraction	aas	0		
addition with cary	adc	2	A+W	* U
	adc	2	A+W	# U
	adc	2	AX	# U
	adc	2	A+W	R+W
	adc	2	R+W	A+W
	adcb	2	A+B	* U
	adcb	2	AL	* U
	adcb	2	A+B	R+B
	adcb	2	R+B	A+B
addition	add	2	A+W	* U
	add	2	A+W	# U
	add	2	AX	# U
	add	2	A+W	R+W
	add	2	R+W	A+W
	addb	2	A+B	* U
	addb	2	AL	* U
	addb	2	A+B	R+B
	addb	2	R+B	A+B
logical and	and	2	A+W	# U
	and	2	AX	# U
	and	2	A+W	R+W
	and	2	R+W	A+W
	andb	2	A+B	* U
	andb	2	AL	* U
	andb	2	A+B	R+B
	andb	2	R+B	A+B
intra segment call indirect	call	1	@ A+W	
intra segment call	call	1	C+W	
inter segment call indirect	calli	1	@ A+W	
inter segment call	calli	1	d : s	
convert byte to word	cbw	0		
clear carry flag	clc	0		
clear direction flag	cld	0		
clear interrupt flag	cli	0		
complement carry flag	cmc	0		
compare	cmp	2	A+W	* U
	cmp	2	A+W	# U
	cmp	2	AX	# U
	cmp	2	A+W	R+W
	cmp	2	R+W	A+W
	cmpb	2	A+B	* U
	cmpb	2	AL	* U
	cmpb	2	A+B	R+B
	cmpb	2	R+B	A+B
compare string	cmps	0		
	cmpsb	0		
convert word to double word	cwd	0		

Befehls-Beschreibung	Befehls-Mnemonik	Anz.	Operanden-Beschreibung	
decimal adjust for addition	daa	0		
decimal adjust f. subtraction	das	0		
decrement by one	dec	1	A+W	
	dec	1	R+W	
decrement byte by one	decb	1	A+B	
division unsigned	div	1	A+W	
	divb	1	A+B	
escape	esc	2	U(063)	A+W
	escb	2	U(063)	A+B
halt	hlt	0		
integer division	idiv	1	A+W	
	idivb	1	A+B	
integer multiplication	imul	1	A+W	
	imulb	1	A+B	
input	in	2	AX	* P+B
	in	2	AX	DX
	inb	2	AL	* P+B
	inb	2	AL	DX
increment by one	inc	1	A+W	
	inc	1	R+W	
	incb	1	A+B	
interrupt	int	1	* U	
	int	1	3	
interrupt if overflow	into	0		
interrupt return	iret	0		
short jump if above	ja	1	C+B	
short jump if above or equal	jae	1	C+B	
short jump if below	jb	1	C+B	
short jump if below or equal	jbe	1	C+B	
short jump if CX is zero	jcxz	1	C+B	
short jump on equal	je	1	C+B	
short jump on greater than	jg	1	C+B	
short jump on gr. than or equal	jge	1	C+B	
short jump on less than	jl	1	C+B	
short jump on ls.than or equal	jle	1	C+B	
intra segment jump indirect	jmp	1	@ A+W	
intra segment jump	jmp	1	C+W	
inter segment jump indirect	jmpi	1	@ M+W	
inter segment jump	jmpi	1	d : s	
short jump	j	1	C+B	
short jump not above	jna	1	C+B	
short jump not above or equal	jnae	1	C+B	
short jump not below	jnb	1	C+B	
short jump not below or equal	jnbe	1	C+B	
short jump not equal	jne	1	C+B	
short jump not greater	jng	1	C+B	
short jump not gr. or equal	jnge	1	C+B	
short jump not less	jnl	1	C+B	
short jump not less or equal	jnle	1	C+B	
short jump not overflow	jno	1	C+B	
short jump not parity	jnp	1	C+B	
short jump not sign	jns	1	C+B	
short jump not zero	jnz	1	C+B	
short jump on overflow	jo	1	C+B	
short jump if parity	jp	1	C+B	

Befehls-Beschreibung	Befehls-Mnemonik	Anz.	Operanden-Beschreibung	
short jump if parity even	jpe	1	C+B	
short jump if parity odd	jpo	1	C+B	
short jump if signed	js	1	C+B	
short jump if zero	jz	1	C+B	
load AH from flags	lahf	0		
load pointer using DS	lds	2	R+W	A+W
load effective address	lea	2	R+W	M
load pointer using ES	les	2	R+W	A+W
lock bus	lock	0		
load string	lods	0		
	lodsb	0		
loop short label	loop	1	C+B	
loop if equal	loope	1	C+B	
loop if not equal	loopne	1	C+B	
loop if not zero	loopnz	1	C+B	
loop if zero	loopz	1	C+B	
move	mov	2	A+W	# U
	mov	2	R+W	# U
	mov	2	A+W	R+W
	mov	2	R+W	A+W
	mov	2	A+W	S
	mov	2	S{NC}	A+W
	mov	2	AX	D+W
	mov	2	D+W	AX
	movb	2	A+B	* U
	movb	2	R+B	* U
	movb	2	A+B	R+B
	movb	2	R+B	A+B
	movb	2	AL	D+B
	movb	2	D+B	AL
move string	movs	0		
	movsb	0		
multiplation unsigned	mul	1	A+W	
	mulb	1	A+B	
negate	neg	1	A+W	
	negb	1	A+B	
no operation	nop	0		
logical not	not	1	A+W	
	notb	1	A+B	
logical or	or	2	A+W	# U
	or	2	AX	# U
	or	2	A+W	R+W
	or	2	R+W	A+W
	orb	2	A+B	* U
	orb	2	AL	* U
	orb	2	A+B	R+B
	orb	2	R+B	A+B
output	out	2	* P+B	AX
	out	2	DX	AX
	outb	2	* P+B	AL
	outb	2	DX	AL

Befehls-Beschreibung	Befehls-Mnemonik	Anz.	Beschreibung	
pop from stack	pop	1	A+W	
	pop	1	S	
	pop	1	R+W	
pop flag from stack	popf	0		
push onto stack	push	1	A+W	
	push	1	S	
	push	1	R+W	
push flag onto stack	pushf	0		
rotate left through carry	rcl	2	A+W	1
	rcl	2	A+W	CL
	rclb	2	A+B	1
	rclb	2	A+B	CL
rotate right through carry	rcr	2	A+W	1
	rcr	2	A+W	CL
	rcrb	2	A+B	1
	rcrb	2	A+B	CL
repeate string operation	rep	0		
repeat string	repe	0		
repeat str. op. not equal	repne	0		
repeat str. op. not zero	repnz	0		
repeat str. op. while zero	repz	0		
return from procedure	ret	1	# U	
	ret	0		
return f. intersegment proc.	reti	1	# U	
	reti	0		
rotate left	rol	2	A+W	1
	rol	2	A+W	CL
	rolb	2	A+B	1
	rolb	2	A+B	CL
rotate right	ror	2	A+W	1
	ror	2	A+W	CL
	rorb	2	A+B	1
	rorb	2	A+B	CL
store AH into flags	sahf	0		
shift arithmetic left	sal	2	A+W	1
	sal	2	A+W	CL
	salb	2	A+B	1
	salb	2	A+B	CL
shift arithmetic right	sar	2	A+W	1
	sar	2	A+W	CL
	sarb	2	A+B	1
	sarb	2	A+B	CL
subtract with borrow	sbb	2	A+W	* U
	sbb	2	A+W	# U
	sbb	2	AX	# U
	sbb	2	A+W	R+W
	sbb	2	R+W	A+W
	sbbb	2	A+B	* U
	sbbb	2	AL	* U
	sbbb	2	A+B	R+B
	sbbb	2	R+B	A+B
scan string	scas	0		
	scasb	0		
segment override	seg	1	S	
shift logical left	shl	2	A+W	1

Befehls-Beschreibung	Befehls-Mnemonik	Anz.	Beschreibung	
	shl	2	A+W	CL
	shlb	2	A+B	1
	shlb	2	A+B	CL
shift logical right	shr	2	A+W	1
	shr	2	A+W	CL
	shrb	2	A+B	1
	shrb	2	A+B	CL
set carry flag	stc	0		
set direction flag	std	0		
set interrupt enable flag	sti	0		
store string	stos	0		
	stosb	0		
subtraction	sub	2	A+W	* U
	sub	2	A+W	# U
	sub	2	AX	# U
	sub	2	A+W	R+W
	sub	2	R+W	A+W
	subb	2	A+B	* U
	subb	2	AL	* U
	subb	2	A+B	R+B
	subb	2	R+B	A+B
test	test	2	A+W	# U
	test	2	AX	# U
	test	2	A+W	R+W
	test	2	R+W	A+W
	testb	2	A+B	* U
	testb	2	AL	* U
	testb	2	A+B	R+B
	testb	2	R+B	A+B
wait while TEST pin	wait	0		
echange	xchg	2	A+W	R+W
	xchg	2	R+W	A+W
	xchg	2	R+W	AX
	xchg	2	AX	R+W
	xchgb	2	A+B	R+B
	xchgb	2	R+B	A+B
translate	xlat	0		
exclusiv or	xor	2	A+W	# U
	xor	2	AX	# U
	xor	2	A+W	R+W
	xor	2	R+W	A+W
	xorb	2	A+B	* U
	xorb	2	AL	* U
	xorb	2	A+B	R+B
	xorb	2	R+B	A+B

Anlage B Verzweigebefehle

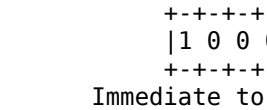
Branch-Mnemonik	Code des bed. Sprungs	Bedeutung
beq	0x75	long branch equal
bge	0x7C	long branch greater or equal
bgt	0x7E	long branch greater
bhi	0x76	long branch on high
bhis	0x00	long branch high or same
ble	0x7F	long branch less or equal
blo	0x00	long branch on low
bloss	0x00	long branch low or same
blt	0x7D	long branch less than
bne	0x74	long branch not equal
br	0x00	long branch

Anlage C Befehlssatz des K 1810 WM 86

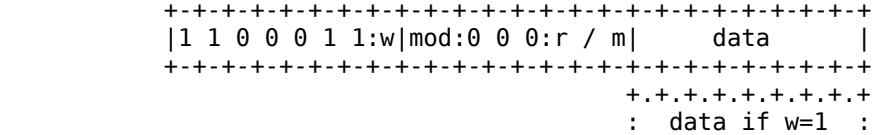
C.1. D A T A T R A N S F E R

M O V = Move

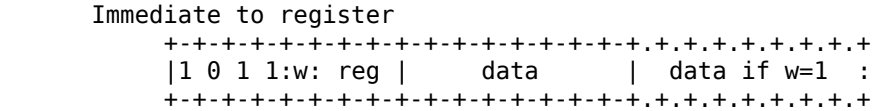
Register/memory to/from register



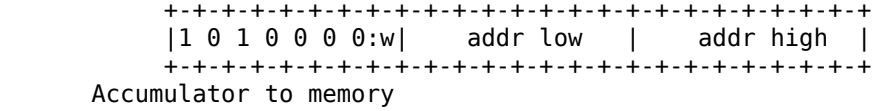
Immediate to register/memory



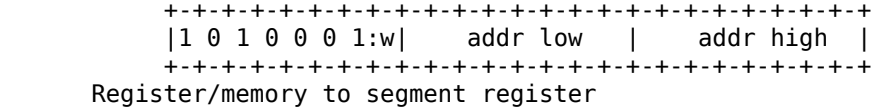
Immediate to register



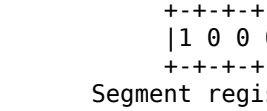
Memory to accumulator



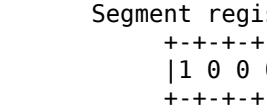
Accumulator to memory



Register/memory to segment register



Segment register to register/memory



PUSH = Push

Register/memory

```
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |1 1 1 1 1 1 1 1|mod:1 1 0:r / m|
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Register

```
     +-+-+-+-+-+-+-+-+
     |0 1 0 1 0: reg |
     +-+-+-+-+-+-+-+-+
```

Segment register

```
     +-+-+-+-+-+-+-+-+
     |0 0 0:reg:1 1 0|
     +-+-+-+-+-+-+-+-+
```

P O P = Pop

Register/memory

```
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |1 0 0 0 1 1 1 1|mod:0 0 0:r / m|
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Register

```
     +-+-+-+-+-+-+-+-+
     |0 1 0 1 1: reg |
     +-+-+-+-+-+-+-+-+
```

Segment register

```
     +-+-+-+-+-+-+-+-+
     |0 0 0:reg:1 1 1|
     +-+-+-+-+-+-+-+-+
```

X C H G = Exchange

Register/memory with register

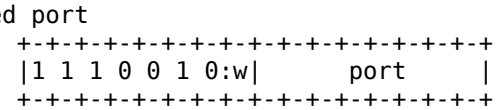
```
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |1 0 0 0 0 1 1:w|mod: reg :r / m|
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Register with accumulator

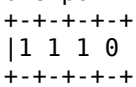
```
     +-+-+-+-+-+-+-+-+
     |1 0 0 1 0: reg |
     +-+-+-+-+-+-+-+-+
```

I N = Input to AL/AX from

Fixed port

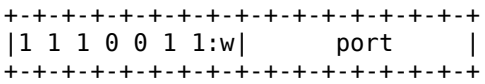


Variable port

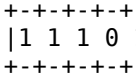


O U T = Output from AL/AX to

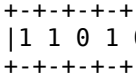
Fixed port



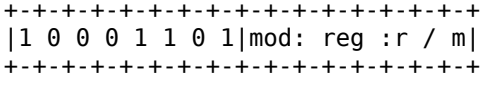
Variable port



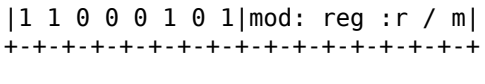
X L A T = Translate byte to AL



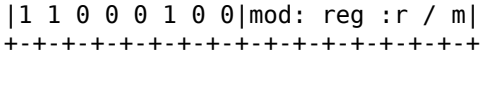
L E A = Load EA to register



L D S = Load pointer to DS



L E S = Load pointer to ES



L A H F = Load AH with flags

S A H F = Store AH into flags

P U S H F = Push flags

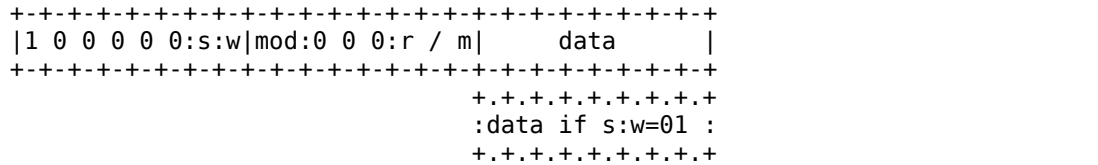
P O P F = Pop flags

C.2. A R I T H M E T I C

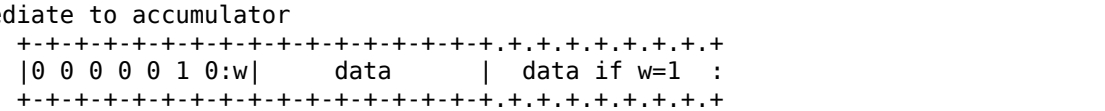
A D D = Add

Register/memory with register to either

Immediate to register/memory

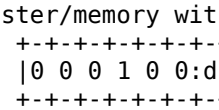


Immediate to accumulator

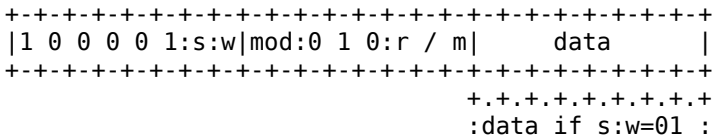


A D C = Add with carry

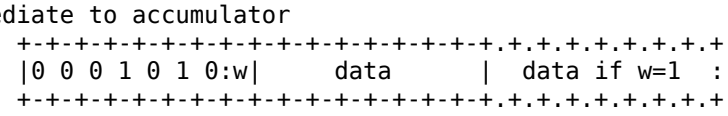
Register/memory with register to either



Immediate to register/memory

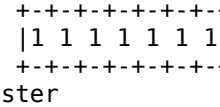


Immediate to accumulator

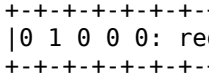


I N C = Increment

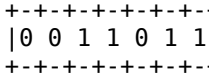
Register/memory



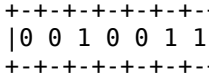
Register



A A A = ASCII adjust for add

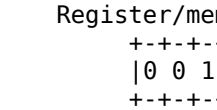


D A A = Decimal adjust for add

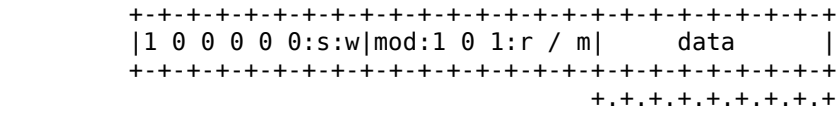


S U B = Subtract

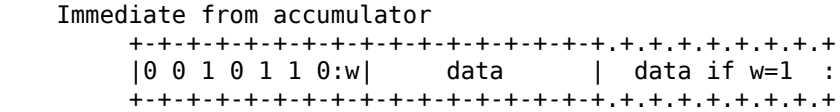
Register/memory and register to either



Immediate from register/memory

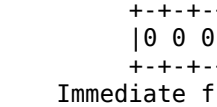


Immediate from accumulator

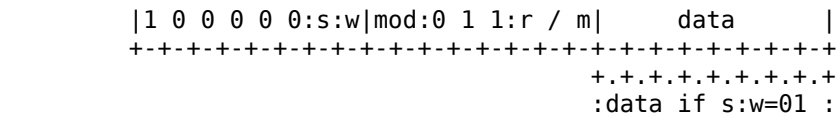


S B B = Subtrac with borrow

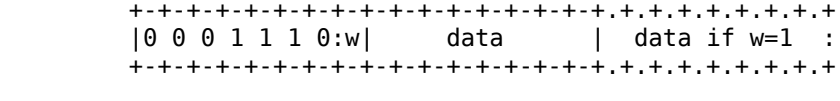
Register/memory and register to either



Immediate from register/memory

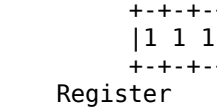


Immediate from accumulator

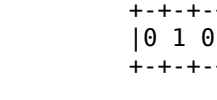


D E C = Decrement

Register/memory



Register

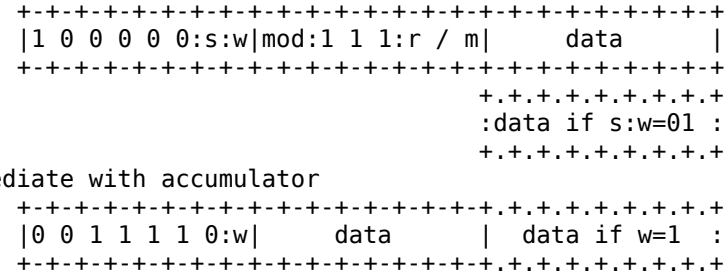


N E G = Change sign

C M P = Compare

Register/memory and register

Immmediate with register/memory



Immediate with accumulator

A A S = ASCII adjust for subtract

D A S = Decimal adjust for subtract

M U L = Multiply (unsigned)

I M U L = Integer multiply (signed)

A A M = ASCII adjust for multiply

D I V = Divide (unsigned)

I D I V = Integer divide (signed)

A A D = ASCII adjust for divide

C B W = Convert byte to word

C W D = Convert word to double word

C.3. L O G I C

N O T = Invert

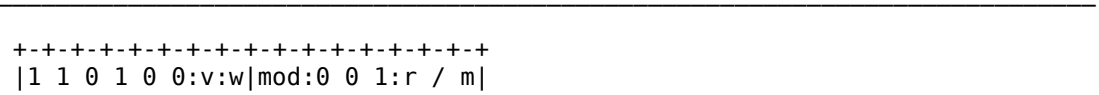
S H L / S A L = Shift logica/arithmetic left

S H R = Shift logical right

S A R = Shift arithmetic right

R O L = Rotate left

R O R = Rotate right

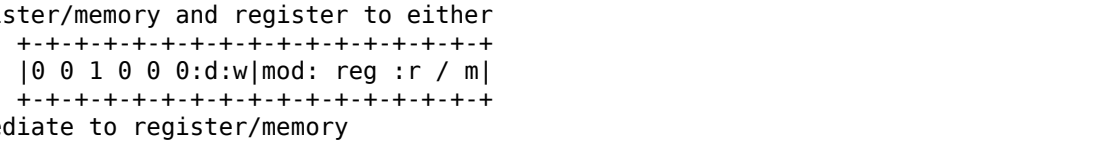


R C L = Rotate through carry left

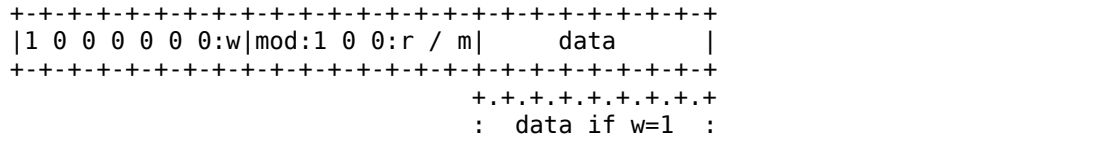
R C R = Rotate through carry right

A N D = And

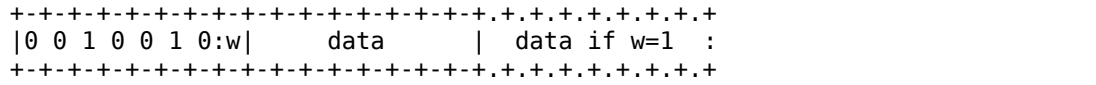
Register/memory and register to either



Immediate to register/memory



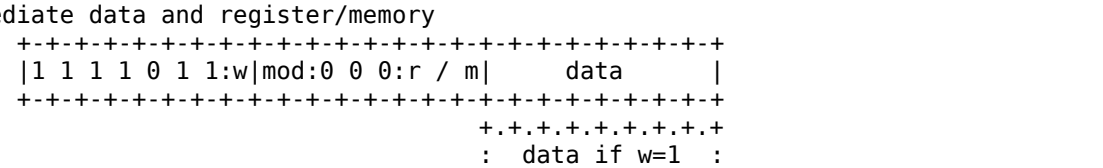
Immediate to accumulator



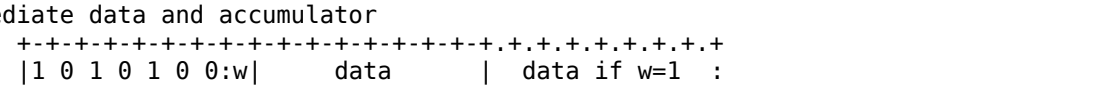
T E S T = And function to flags, no result

Register/memory and register

Immediate data and register/memory

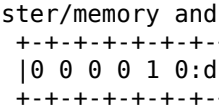


Immediate data and accumulator

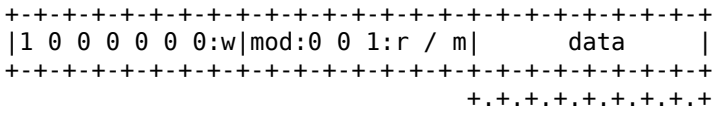


O R = Or

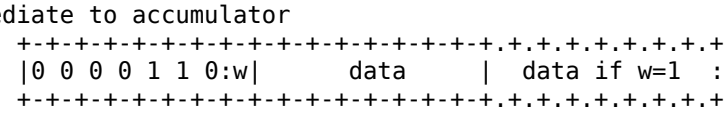
Register/memory and register to either



Immediate to register/memory

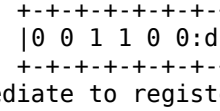


Immediate to accumulator

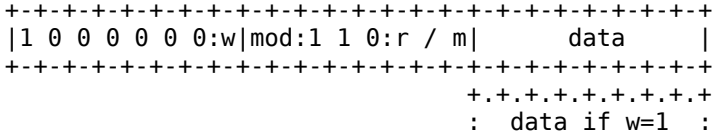


X O R = Exclusive or

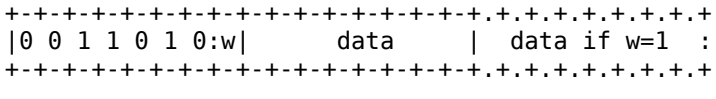
Register/memory and register to either



Immediate to register/memory

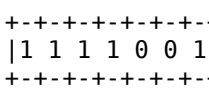


Immediate to accumulator



C.4. S T R I N G M A N I P U L A T I O N

R E P = Repeat prefix

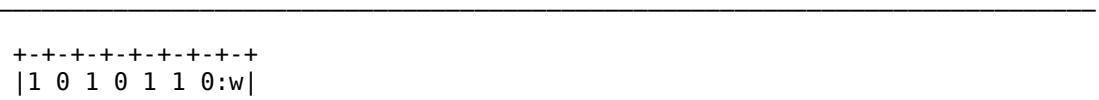


M O V E S = Move string byte/word

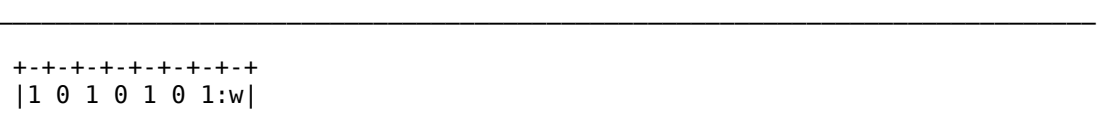
C M P S = Compare string byte/word

S C A S = Scan string byte/word

L O D S = Load string byte/word



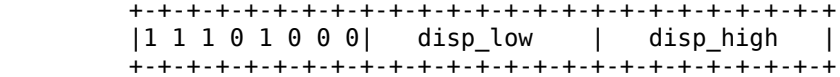
S T O S = Store string byte/word from AX|AL



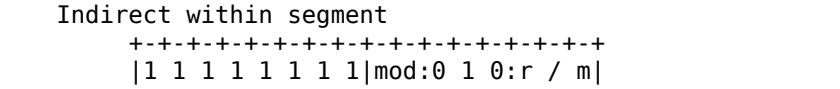
C.5. C O N T R O L T R A N S F E R

C A L L = Call

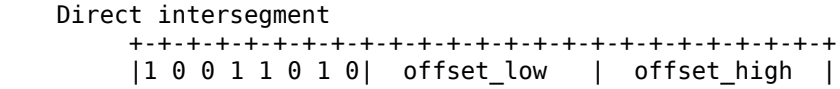
Direct within segment

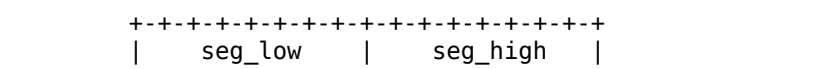


Indirect within segment

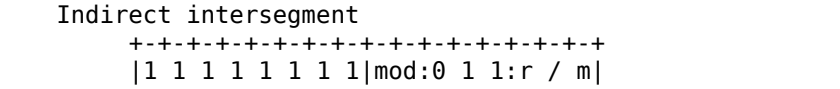


Direct intersegment



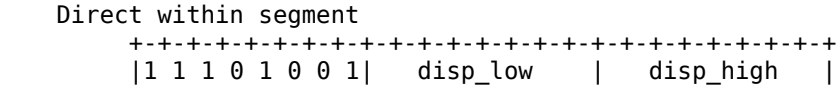


Indirect intersegment

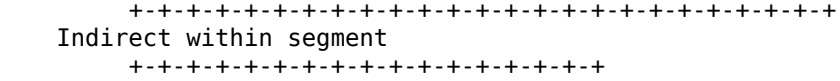


J M P = Unconditional jump

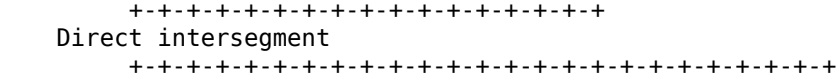
Direct within segment

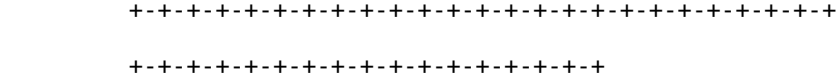


Indirect within segment

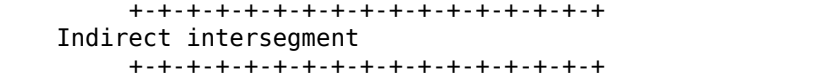


Direct intersegment

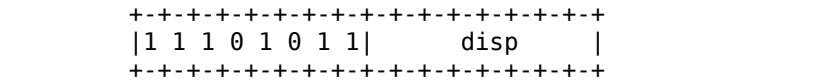




Indirect intersegment

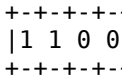


J = Direct within segment-short

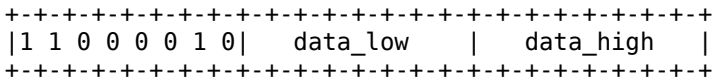


R E T = Return from CALL

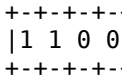
Whithin segment



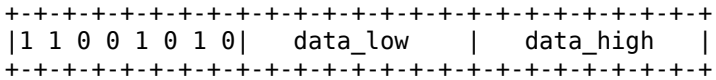
Direct within segment adding immediately to SP



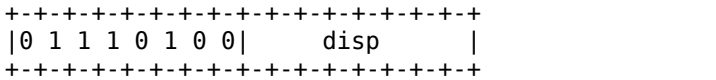
Intersegment



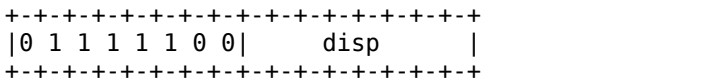
Intersegment adding immediately to SP



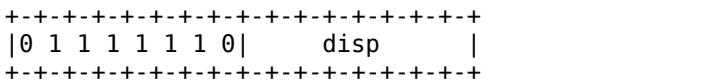
J E / J Z = Jump on equal/zero



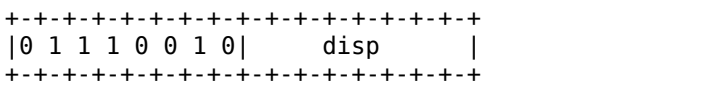
J L / J N G E = Jump on less/not greater



J L E / J N G - Jump on less or equal



J B / J N A E = Jump on below/not above or equal



J B E / J N A = Jump on below or equal/not above

J P / J P E = Jump on parity/parity even

J O = Jump on overflow

J S = Jump on sign

J N E / J N Z = Jump on not equal/not zero

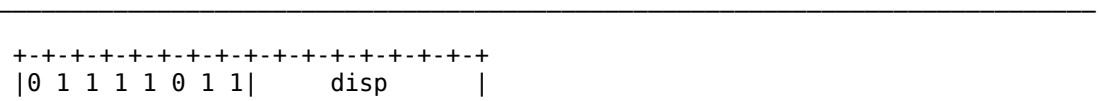
J N L / J G E = Jump on not less/greater or equal

J N L E / J G = Jump on not less or equal/greater

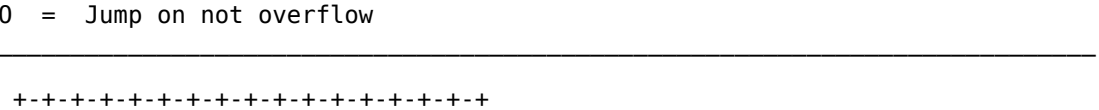
J N B / J A E = Jump on not below/above or equal

J N B E / J A = Jump on not below or equal/above

J N P / J P O = Jump on not parity/parity odd

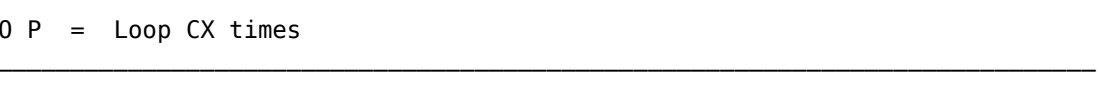


J N O = Jump on not overflow

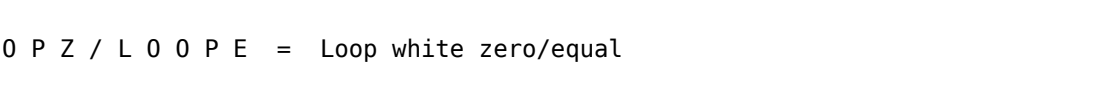


J N S = Jump on not sign

L O O P = Loop CX times



L O O P Z / L O O P E = Loop white zero/equal



L O O P N Z / L O O P N E = Loop white not zero/equal

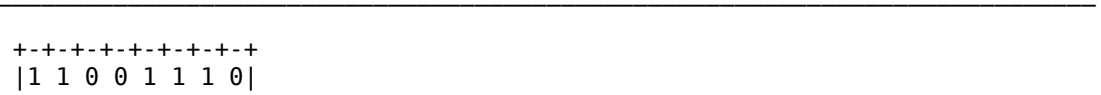
J C X Z = Jump on CX zero

I N T = Interrupt

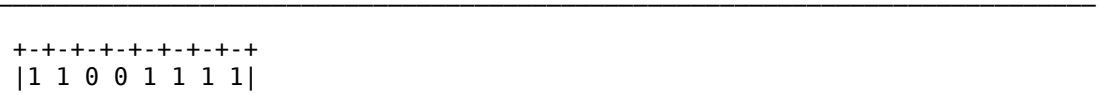
Type specified

Type3

I N T O = Interrupt on overflow



I R E T = Interrupt return



C.6. P R O C E S S O R C O N T R O L

C L C = Clear carry

C M C = Complement carry

S T C = Set carry

C L D = Clear direction

S T D = Set direction

C L I = Clear interrupt

S T I = Set interrupt

H L T = Halt

W A I T = Wait

```
     +-+-+-+-+-+-+-+-+
     |1 0 0 1 1 0 1 1|
     +-+-+-+-+-+-+-+-+
```

E S C = Escape (to external device)

```
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |1 1 0 1 1:  x  |mod:  x  :r / m|
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

L O C K = Bus lock prefix

```
     +-+-+-+-+-+-+-+-+
     |1 1 1 1 0 0 0 0|
     +-+-+-+-+-+-+-+-+
```

S E G = Segment override prefix

```
     +-+-+-+-+-+-+-+-+
     |0 0 1:reg:1 1 0|
     +-+-+-+-+-+-+-+-+
```

C.7. Bemerkungen zum Befehlssatz K 1810 WM 86

AL = 8-Bit Accumulator
AX = 16-Bit Accumulator
CX = Count-Register
DS = Data-Segment
ES = Extra-Segment
Above/below bezieht sich auf unsigned-Werte
Greater = positiver
Less = weniger positiv (negativer) signed-Wert

Wenn d = 1 dann "zu"("to").
Wenn d = 0 dann "aus"("from").
Wenn w = 1 dann Wort-Befehl.
Wenn w = 0 dann Byte-Befehl.
Wenn s:w = 01 dann bilden 16 Bits des
unmittelbaren Datum den Operanden.
Ist s:w = 11 dann bildet das vorzeichenerweiterte
unmittelbare Datumsbyte den 16 Bit-Operanden.

x = Achtung!, geht zum externen Gert.
Wenn v = 0 dann "count" = 1.
Wenn v = 1 dann "count" in (CL) Register.
z wird bei Strings zum Vergleich mit dem ZFLAG verwendet.

Befehle, die sich auf das Flag-Register-File (16 Bit-Objekt)
beziehen, verwenden die hier mit FLAGS angegebenen Symbole:

```
FLAGS =
 X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)
 1 2 3 4  5    6    7    8    9    10  11 12  13 14  15 16
```

Wenn mod = 11 dann wird r/m als a reg-Feld behandelt.
Wenn mod = 00 dann DISP = 0*
(disp_low and disp_high sind nicht vorhanden).
Wenn mod = 01 dann DISP = disp_low vorzeichenerweitert auf
16 Bits, disp_high gibt es nicht.
Wenn mod = 10 dann DISP = disp_high : disp_low

```
                          Standard seg reg
Wenn r/m = 000 dann EA = (BX)+(SI)+DISP            DS
Wenn r/m = 001 dann EA = (BX)+(DI)+DISP            DS
Wenn r/m = 010 dann EA = (BP)+(SI)+DISP            SS
Wenn r/m = 011 dann EA = (BP)+(DI)+DISP            SS
Wenn r/m = 100 dann EA =      (SI)+DISP            DS
Wenn r/m = 101 dann EA =      (DI)+DISP            SS
Wenn r/m = 110 dann EA = (BP)+     DISP*           DS
```

C.8. Typisches Format der K 1810 WM 86-ein-Operanden-Operation

```
   +.+.+.+.+.+.+.+.+   +-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
   :     :seg:     :   |   opcode    |w|   |mod|opcod| r/m |
   +.+.+.+.+.+.+.+.+   +-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
      (optional)
                       +.+.+.+.+.+.+.+.+   +.+.+.+.+.+.+.+.+
                       :    disp_lo    :   :    disp_hi    :
                       +.+.+.+.+.+.+.+.+   +.+.+.+.+.+.+.+.+
                          (optional)          (optional)
```

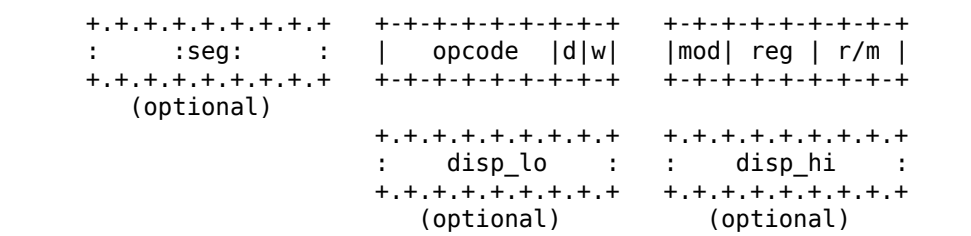
Operand ist Register oder Memory, der durch
seg, mod, r/m, disp_lo, disp_hi spezifiziert wird.

mod = 00,01,10: Operand ist Memory (siehe Tafel 2)
11: Operand ist Register (siehe Tafel 3)

seg gibt Overriding-Segment-Register an

w = 0 : Operand besteht aus 8 Bits
1 : Operand besteht aus 16 Bits

C.9. Typisches Format der K 1810 WM 86-zwei-Operanden-Operation, wenn einer der Operanden ein Register ist



Ein Operand ist Register oder Memory, der durch
seg, mod, r/m, disp_lo, disp_hi spezifiziert wird.

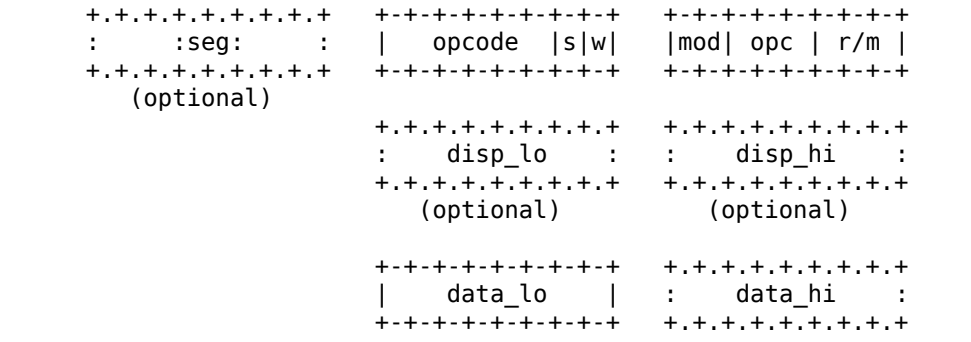
mod = 00,01,10:ein Operand ist Memory (siehe Tafel2)
11: ein Operand ist Register (siehe Tafel3)

seg gibt Overriding-Segment-Register an

Der andere Operand ist ein durch reg spezifiziertes Register,

w = 0 : Operanden bestehen je aus 8 Bits
1 : Operanden bestehen je aus 16 Bits
d = 0 : Destination ist der erste Operand
1 : Destination ist der zweite Operand

C.10. Typisches Format der K 1810 WM 86-zwei-Operanden-Operation, wenn der zweite Operand eine Konstante (immediate data) ist



Der erste Operand ist Register oder Memory, der durch
seg, mod, r/m, disp_lo, disp_hi spezifiziert wird.

mod = 00,01,10: erster Operand ist Memory (siehe Tafel 2)
11: erster Operand ist Register (siehe Tafel3)

seg gibt Overriding-Segment-Register an

Der zweite Operand ist eine Konstante (immediate data;
unmittelbare Adressierung), der durch
s, w, data_lo, data_hi spezifiziert wird.

```
w = 0  : DATA = data_lo (data_hi gibt es nicht)
    1  : s = 0  : DATA = data_hi, data_lo
             1  : DATA = data_lo sign extended
                       (data_hi gibt es bei s = 1 nicht)
```

Anlage D Bestimmung des K 1810 WM 86-Offset bei der Adressierung eines Memory-Operanden (zu verwenden, wenn mod != 11; anderenfalls Anlage E)

Diese Tabelle ist nur auf einen Operanden anzuwenden;
der andere kann niemals ein Memory-Operand sein.

mod spezifiziert wie disp_lo und disp_hi bei der Definition
des displacement verwendet werden:

```
         00 : DISP = 0 (disp_lo und disp_hi gibt es nicht)
   mod = 01 : DISP = disp_lo sign extended
                     (disp_hi gibt es nicht)
         10 : DISP = disp_hi, disp_lo
```

r/m spezifiziert, welche Basis- and Index-Registerinhalte
zum Displacement addiert werden, um damit den Offset der
Operanden-Adresse zu bilden:

```
         000 : OFFSET = (BX)+(SI)+DISP \
         001 : OFFSET = (BX)+(DI)+DISP |
         010 : OFFSET = (BP)+(SI)+DISP |
   r/m = 011 : OFFSET = (BP)+(DI)+DISP |  Adressierungs-
         100 : OFFSET =      (SI)+DISP |  art
         101 : OFFSET =      (DI)+DISP |
         110 : OFFSET = (BP)     +DISP |
         111 : OFFSET = (BX)     +DISP /
```

( ) bedeutet "Inhalt von"

Das folgende ist eine Ausnahme zu den obigen Regeln:

```
   wenn mod = 00   und   r/m = 110  \   direkte
   dann OFFSET = disp_hi, disp_lo   /   Adressierung
```

Anlage E Bestimmung des K 1810 WM 86-Register-Operanden
(zu verwenden, wenn mod = 11;
anderenfalls Anlage D)

erster Operand			zweiter Operand		
r/m	8-bit	16-bit	reg	8-bit	16-bit
000	AL	AX	000	AL	AX
001	CL	CX	001	CL	CX
010	DL	DX	010	DL	DX
011	BL	BX	011	BL	BX
100	AH	SP	100	AH	SP
101	CH	BP	101	CH	BP
110	DH	SI	110	DH	SI
111	BH	DI	111	BH	DI