Inhaltsverzeichniss

1 DER 8086 / 8088 MIKROPROZESSOR

Der 8086 er Mikroprozessor und der 8088 er Mikroprozessor von Intel haben einen Adressbus von 20 Bit Breite und können daher 1MB (2^{20} Byte) Arbeitsspeicher adressieren. Man sagt, diese Prozessoren verfügen über einen 1MB großen Adressraum.
Die kleinste adressierbare Einheit dieses Adressraums ist ein Byte.
Jedes Byte des Arbeitsspeichers ist durch eine eindeutige Adresse ansprechbar, die zwischen 0 und 2^{20} - 1 liegen muß. Diese Adresse wird als physikalische Adresse bezeichnet. Mit einem Speicherzugriff kann auf ein Byte oder auf zwei benachbarte Bytes (Wort) zugegriffen werden.

Bemerkung:
Beim 8086 er Mikroprozessor hat der Datenbus eine Breite von 16 Bit, beim 8088 er Mikroprozessor hat der Datenbus eine Breite von 8 Bit.

1.1 logische und physikalische Adressen

1.1.1 Motivation

Der 8086 / 8088 er Mikroprozessor kann 1 MB (= 1024 KB = 2^{20} Byte) Arbeitsspeicher adressieren. Um diesen Speicherbereich adressieren zu können, ist ein 20 Bit breiter Adressbus nötig. Die physikalische Adresse ist eine 20 Bit Größe und wird über die Adressleitungen auf den Adressbus ausgegeben.
Die internen Register des 8086 / 8088 er Mikroprozessors sind jedoch nur 16 Bit breit. Das bedeuted, daß die physikalische 20 Bit Adresse aus zwei 16 - Bit -Teilkomponenten aufgebaut werden muß. Dies ist eine charakteristische Eigenschaft des 8086 / 8088 er Mikroprozessors, die von den sonstigen 16 - Bit Mikroprozessoren abweicht.
Die physikalische Adresse wird aus der Segmentadresse, (die in einem bestimmten Register steht) und der Offsetadresse, (die auch in einem bestimmten Register steht), aufgebaut.
Statt Segmentadresse und Offsetadresse sagt man kurz Segmet bzw. Offset.
Die Darstellung der physikalischen Adresse in der Form *Segment : Offset* nennt man logische Adresse.

Berechnung der physikalischen Adresse (in einem speziellen 20 Bit Register):

physikalische Adresse = Segment * 16 + Offset

Bemerkung:
1) Multiplikation mit 16 bedeuted, daß an das Bitmuster 4 Nullen angehängt werden; es wird also 20 Bit breit.

2) Falls nicht anders gekennzeichnet, sind in diesem Skript alle Adressen der Form Segment:Offset, hexadezimal !!

1.1.2 Zusammenfassung

Zum Programmieren des 8086 / 8088 benötigt man nicht die physikalische Adresse, sondern die (nicht eindeutige) Segment : Offset Darstellung.

Der Arbeitsspeicher wird (gedanklich) in 16 Byte große Abschnitte unterteilt.
Abschnitt 0 beginnt bei Adresse 0,
Abschnitt 1 beginnt bei Adresse 16,
Abschnitt 2 beginnt bei Adresse 32, usw.
In der Segment : Offset Darstellung wird
- das Segment (Abschnitt), auf das man sich bezieht, angegeben und
- der Offset, als die auf dieses Segment bezogene relative Adresse, angegeben.

Segment bzw. Offset sind jeweils 2 Byte groß.

Beispiele:

Segment : Offset
Physikalische Adresse (Segment * 16 + Offset)

0000 : 0000

$$\left[0\cdot16^3+0\cdot16^2+0\cdot16^1+0\cdot16^0\right]\cdot16+0\cdot16^3+0\cdot16^2+0\cdot16^1+0\cdot16^0=0$$

0000 : 0019

$$\left[0\cdot16^3+0\cdot16^2+0\cdot16^1+0\cdot16^0\right]\cdot16+0\cdot16^3+0\cdot16^2+1\cdot16^1+9\cdot16^0=25$$

0001 : 0009

$$\left[0\cdot16^3+0\cdot16^2+0\cdot16^1+1\cdot16^0\right]\cdot16+0\cdot16^3+0\cdot16^2+0\cdot16^1+9\cdot16^0=25$$

1234 : ABCD

$$\left[1\cdot16^3+2\cdot16^2+3\cdot16^1+4\cdot16^0\right]\cdot16+10\cdot16^3+11\cdot16^2+12\cdot16^1+13\cdot16^0=118541$$

Bemerkung:
1) Bei festgehaltener Segmentadresse läßt sich durch die Gesamtheit aller 16 - Bit Offset Adressen ein Speicherbereich von 64 KByte adressieren.
Diesen 64 KByte Speicherbereich nennt man auch 'Segment'
Innerhalb des Segments gibt die Offsetadresse den Abstand relativ zur Segmentadresse an.
Man nennt die Offsetadresse häufig die effektive Adresse EA.

2) Von der Adresse `F000:0000` bis zur Adresse `F000:FFFF` befindet sich der ROM (Read Only Memory d.h: dieser Bereich des Arbeitsspeichers kann nur gelesen aber nicht beschrieben werden). Hier befindet sich das Programm und die Daten für den Boot-Vorgang und die BIOS Funktionen.

1.2 Die Register des 8086 / 8088 er Mikroprozessors

Zusätzlich zum Arbeitsspeicher gibt es noch einige Speicherstellen, Register genannt, die 2 Byte groß sind.

Wenn der Mikroprozessor bestimmte Befehle abarbeitet, wird der Inhalt dieser Register verändert.

Bitnummer	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AX	AH								AL							
BX	BH								BL							
CX	CH								CL							
DX	DH								DL							
SI																
DI																
BP																
SP																
IP																
FLAGS																
CS																
DS																
ES																
SS																

1.3 Beschreibung einiger Register

AX, BX, CX, DX nennt man die allgemeinen Register. Mit jedem dieser Register können arithmetische und logische Operationen durchgeführt werden.
Der Akkumulator AX nimmt eine Sonderstellung ein. Bei vielen Befehlen führt die Verwendung des Akkumulators anstelle eines anderen Registers zu kürzerem Maschinenbefehl und zu kürzeren Befehlsausführungszeiten.

CS, DS, ES, SS nennt man die Segment Register. Die Segment Register dienen zur Adressierung des Arbeitsspeichers.

Die allgemeinen Register (und nur diese) lassen sich in zwei 8 Bit Register aufteilen und zwar in einen sogenannten High und einen Low Teil. Diese 8 Bit Register lassen sich ebenso ansprechen und verwenden wie ein 16 Bit Register. Sie können jedoch nur ein Byte statt ein Wort (2 Byte) aufnehmen. Wird eine 16 Bit Operation durchgeführt, so ist z.B. das DX Register zu verwenden, bei einer 8 Bit Operation kann dagegen das DH oder das DL Register verwendet werden.
Beispiele:

Maschinencode: | B8 | E1 | 47 | symbolisch: `MOV AX, 47E1`

Maschinencode: | B2 | F3 | symbolisch: `MOV DL, F3`

Der Befehlssatz des 8086 / 8088 er Mikroprozessors erlaubt, daß *nur* das Register BX als Adresse (besser Offset einer Adresse) verwendet werden darf.
Das BX Register kann also im Gegensatz zu AX, CX, DX als Adresse verwendet werden:

`MOV AX, [BX]` ist daher möglich (siehe später)
`MOV CX, [AX]` nicht möglich
`MOV DX, [CX]` nicht möglich
`MOV AX, [DX]` nicht möglich

Der Befehlszähler IP spielt eine wichtige Rolle bei der Ausführung eines Programms. Zusammen mit dem CS Register gibt *CS : IP* in der Segment : Offset Darstellung die *Adresse des nächsten auszuführenden Befehls* im Arbeitsspeicher an.

Die Reihenfolge der Ausführung der einzelnen Befehle geschieht also nach folgendem sich wiederholenden Schema:

1)
Befehl ausführen, dessen Adresse bei CS : IP beginnt.
Nach Ausführung dieses Befehls bekommt der Befehlszähler IP einen neuen Wert.

2)
weiter mit 1)

1.3.1 Wichtige Bemerkung

1.3.1.1 Bytes als Befehl interpretieren

Der Mikroprozessor versucht die Bitfolge an der Adresse **CS : IP** als einen Befehl aus dem Befehlssatz zu interpretieren. Er versucht dies, auch wenn dort Daten stehen wie z.B. 4711 oder ein Text wie "Hallo wie geht es dir ?".
Falls die Bitfolge kein Befehl aus dem Befehlssatz ist, wird ein Hardware Interrupt ausgelöst.

1.3.1.2 Op - Code

Der Op-Code eines Maschinenbefehls besteht aus ein oder zwei Bytes.
Der Mikroprozessor versucht den Maschinenbefehl anhand des ersten Bytes des Op-Codes zu erkennen (anschaulich: im Handbuch zu finden).
Gelingt dies nicht, weil es mehrere Maschinenbefehle mit diesem Op-Code gibt, benutzt der Mikroprozessor das zweite Byte im Op-Code um den Maschinenbefehl zu erkennen (identifizieren).

1.3.1.3 Zusammenbasteln der Adresse

Zugriff eines Maschinenbefehls auf eine Adresse im Arbeitsspeicher:
Wenn bei der Ausführung eines Maschinenbefehls auf einen Speicheroperanden (an einer Adresse im Arbeitsspeicher) zugegriffen werden muß ,
(z.B: Kopiere den Inhalt an der Adresse 6574:7843 ins Register AL),
muß dessen Adresse im Operandenteil des Maschinenbefehls stehen.
Leider ist dies nicht ganz richtig.
Im Operandenteil steht nur der Offset dieser Adresse.
Woher holt sich der Mikroprozessor aber das Segment dieser Adresse ?
Das Segment holt er sich standardmäßig aus dem Register DS, einem sogenannten Segmentregister. (CS, DS, ES, SS heißen Segmentregister).
Das heißt der Programmierer muß zu Beginn seines Programms das Segment der Adresse an der seine ganzen Daten (auf die er mit Maschinenbefehlen zugreifen kann) beginnen, in das Segmentregister DS kopieren !

Bem:
Adressen werden innerhalb von symbolischen Befehlen in eckigen Klammern [ ] geschrieben.

Beispiel 1:
```
MOV AX,[A234]
```
bedeutet, daß der Inhalt der Adresse DS:A234 in den Akkumulator AX kopiert wird.

Beispiel 2:
```
MOV AX,[BX]
```
bedeutet, daß der Inhalt der Adresse DS:BX in den Akkumulator AX kopiert wird.

Beispiel 3:
```
MOV AX, [AB12]
```
bedeutet, daß der Inhalt der Adresse DS:AB12 in den Akkumulator AX kopiert wird.

Beispiel 4:
```
MOV AX, AB12
```
bedeutet, daß die Konstante AB12 in den Akkumulator AX kopiert wird.

Zusammenfassung:

Adressen, die der Mikroprozessor benötigt:
a) Die Adresse des nächsten auszuführenden Befehls beginnt bei: **CS : IP**
b) Die Adresse auf die in einem Maschinenbefehl (im Operandenteil) zugegriffen wird beginnt bei: **DS : Adressangabe im Operandenteil** des Maschinenbefehl.
Ausnahme siehe unten)

1.3.1.4 Ausnahme

Wenn bei der Ausführung eines Maschinenbefehls auf einen Speicheroperanden (an einer Adresse im Arbeitsspeicher) zugegriffen werden muß, holt sich der Mikroprozessor das Segment dieser Adresse standardmäßig aus dem Register DS (siehe oben).
Wenn er sich das Segment aus einem anderen Segmentregister holen soll, muß dieses Segmentregister im Assemblerbefehl angegeben werden.

Beispiele:

```
MOV AX, [CS:A234]
MOV AX, [DS:A234]
MOV AX, [ES:A234]
MOV AX, [SS:A234]
```

1.4 Beschreibung einiger Flags

Das Flag Register enthält alle relevanten Informationen über den Status des Mikroprozessors und die Resultate der letzten Befehle.

Bitnummer	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FLAGS					O	D	I	T	S	Z		A		P		C

Carry Flag	CF	Übertragsflag
Parity Flag	PF	Paritätsflag
Auxiliary Carry Flag	AF	Hilfsübertragsflag
Zero Flag	ZF	Nullflag
Sign Flag	SF	Vorzeichenflag
Overflow Flag	OF	Überlaufflag
Trap Flag	TF	Einzelschrittflag
Interrupt Enable Flag	IF	Interruptflag
Direction Flag	DF	Richtungsflag

1.4.1 Das Carry Flag CF

wird gesetzt (auf 1) , wenn bei einer Operation ein Übertrag aus dem höchstwertigen Bit hinaus erfolgt (z.B. bei der Addition) oder wenn ein "Borgen" in das höchstwertige Bit hinein erfolgt (z.B. bei der Subtraktion), andernfalls wird dieses Flag gelöscht (auf 0 gesetzt).
Das CF wird bei allen arithmetischen (Addition, Subtraktion, Multiplikation, Division) Operationen verändert.

Beispiele:

a) Addition zweier Dualzahlen

```
     10101110      (174 Dez)            CF = 1
+    01110100      (116 Dez)
------------
     00100010       (34 Dez)
```

b) Subtraktion zweier Dualzahlen

```
     00001100       (12 Dez)            CF = 1
-    00001111       (15 Dez)
------------
     11111101      (253 Dez)
```

Beispiel 1

```
MOV AX, 3
SUB AX, 5
```

(CF = 1)

Beispiel 2

```
MOV AH, FE
ADD AH, 5
```

(CF = 1)

1.4.2 Das Zero Flag ZF

wird gesetzt, wenn das Ergebnis nach einer Operation 0 ist, andernfalls wird es gelöscht.

Beispiel:

```
      01101101       (109 Dez)             ZF = 1
  -   01101101       (109 Dez)
  ------------
      00000000         (0 Dez)
```

1.4.3 Das Sign Flag SF

wird gesetzt, wenn das höchstwertige Bit des Ergebnisses nach einer Operation gesetzt ist, andernfalls wird es gelöscht.

Beispiel:

```
      01100000        (96 Dez)             SF = 1
  +   01000000        (64 Dez)
  ------------
      10100000       (160 Dez)
```

1.5 Die Maschinenbefehle des 8086 / 8088 er Mikroprozessors

Der kürzeste Maschinenbefehl ist ein Byte lang, der längste Befehl ist sechs Byte lang.
Zwingend enthalten in jedem Maschinenbefehl ist nur das erste Byte mit dem
Op - Code und der Information (Wort Bit), ob der Operand (falls vorhanden) im Maschinenbefehl 1 Byte oder 2 Byte groß ist.
Nur einer der Operanden eines Maschinenbefehls kann ein Speicheroperand sein !!
Wenn der Maschinenbefehl Register oder Operanden verwendet, wird diese Information im 2. Byte gespeichert, das zum Operandenteil gehört.

1.5.1 Der Befehl CMP (Vergleich)

Beispiel: `CMP AX, BX`
führt eine Subtraktion durch (linker Operand minus rechter Operand), ohne das Ergebnis abzulegen. Es wird (nicht nur) das Carry Flag folgendermaßen verändert: Wenn bei der Subtraktion ein Borgen auftritt (d.h. der linke Operand ist kleiner als der rechte, falls man sie vorzeichenlos auffaßt), wird das Carry Flag gesetzt. Andernfalls wird es gelöscht.
Der Befehl CMP unterscheidet sich vom Befehl SUB dadurch, daß bei SUB das Ergebnis im linken Operand abgespeichert wird.

1.5.2 Der Befehl JMP (Unbedingter Sprung)

Befehlsformat:

Maschinencode :

EB	d

Symbolisch : `JMP`

verbal:
1) Der Befehlszähler wird um die Befehlslänge, also um 2, erhöht, kurz:
`IP := IP + 2`
2) Das dem Op-Code folgende Byte (d) wird vom MP als eine Zahl zwischen -128 und 127 interpretiert, auf zwei Byte (vorzeichen)erweitert und anschließend zum Inhalt des Befehlszählers hinzuaddiert.

Kurzschreibweise:
`IP := IP + 2`
`IP := IP + d`

1.5.2.1 Beispiel eines JMP Befehls

1) Annahmen
`CS=7312, IP=0003` (sämtliche Angaben hexadezimal)

Die folgende Tabelle stellt einen Auszug aus dem Arbeitsspeicher dar.

Adresse (Hex)	Inhalt (Hex)	
. . .	. .	
7312 : 0003	EB	
7312 : 0004	04	
7312 : 0005	. .	
7312 : 0006	. .	
7312 : 0007	. .	
7312 : 0008	. .	
7312 : 0009	. .	← Der JB folgende Befehl beginnt hier, Begründung folgt
. . .	. .	

2) Was macht der Mikroprozessor ?
Der Mikroprozessor führt den Befehl aus, dessen Adresse bei CS:IP, also bei 7312:0003 beginnt. Anhand des Bytes des Op-Codes schaut er im Befehlssatz des Mikroprozessors nach und findet heraus, daß es sich um einen JB Befehl handelt:

EB	04

oder symbolisch `JMP`

Es geschieht nun (laut Befehlsbeschreibung oben) folgendes:
1) IP wird um die Befehlslänge (also 2) erhöht, d.h:
`IP = 0003+ 2 = 0005`

2) Das dem Op-Code (EB) des Befehls JMP folgende Byte (04) wird als positive Distanz 4 interpretiert (Distanz ist vorzeichenbehaftet).
Diese Distanz 4 wird als 2 Byte dargestellt (0004) und zum aktuellen Wert (0005) des Befehlszählers hinzuaddiert:

```
      0   0   0   5            ( 5 Dez)
  +   0   0   0   4            ( 4 Dez)
 ----------------
      0   0   0   9            ( 9 Dez)
```

Der Befehlszähler hat nun also den Wert 0009
Das heißt der nächste Befehl, den der MP ausführt, beginnt an der Adresse 7312:0009

Bem:
Es ist üblich, daß bei der symbolischen Darstellung des JMP Befehls der neue Wert von IP dem Wort JMP folgend, dargestellt wird, hier:
`JMP 0009`

1.5.2.2 Beispiel eines JMP Befehls

1) Annahmen
`CS=0815, IP=0040` (sämtliche Angaben hexadezimal)

Die folgende Tabelle stellt einen Auszug aus dem Arbeitsspeicher dar.

Adresse (Hex)	Inhalt (Hex)
...	..
0815 : 003B	..
0815 : 003C	..
0815 : 003D	..
0815 : 003E	..
0815 : 003F	..
0815 : 0040	EB
0815 : 0041	FB
...	..

← Der JMP folgende Befehl beginnt hier, Begründung folgt

2) Was macht der Mikroprozessor ?
Der Mikroprozessor führt den Befehl aus, dessen Adresse bei CS:IP, also bei 0815:0040 beginnt. Anhand des Bytes des Op-Codes schaut er im Befehlssatz des Mikroprozessors nach und findet heraus, daß es sich um einen JB Befehl handelt:

EB	FB

oder symbolisch `JMP`

Es geschieht nun (laut Befehlsbeschreibung oben) folgendes:
1) IP wird um die Befehlslänge (also 2) erhöht, d.h:
`IP = 0040 + 2 = 0042`

2) Das dem Op-Code (EB) des Befehls JMP folgende Byte (FB) wird als negative Distanz -5 interpretiert (Distanz ist vorzeichenbehaftet).
Diese Distanz -5 wird als 2 Byte dargestellt (FFFB) und zum aktuellen Wert (0042) des Befehlszählers hinzuaddiert:

```
     0   0   4   2          ( 66 Dez)
 +  1F  1F   F   B           (-5 Dez)
 ----------------
     0   0   3   D          ( 61 Dez)
```

Der Befehlszähler hat nun also den Wert 003D
Das heißt der nächste Befehl, den der MP ausführt, beginnt an der Adresse 0815:003D
Bem:
Es ist üblich, daß bei der symbolischen Darstellung des JMP Befehls der neue Wert von IP dem Wort JMP folgend, dargestellt wird, hier:
`JMP 003D`

1.5.3 Der Befehl JB (Bedingter Sprung)

Befehlsformat:

Maschinencode :

72	d

symbolisch : `JB`

verbal:
1) Der Befehlszähler wird um die Befehlslänge, also um 2, erhöht, kurz:
`IP := IP + 2`
2) Wenn das Carry Flag gesetzt ist (1), wird das dem Op-Code folgende Byte d vom MP als eine Zahl zwischen -128 und 127 interpretiert, auf zwei Byte (vorzeichen)erweitert und anschließend zum Inhalt des Befehlszählers hinzuaddiert.
Wenn das Carry Flag nicht gesetzt ist (0), passiert nichts.

Kurzschreibweise:

```
IP := IP + 2
IF CF = 1 THEN
  IP := IP + d
```

1.5.3.1 Beispiel eines JB Befehls

1) Annahmen
`CS=4711, IP=0011, CF = 1` (sämtliche Angaben hexadezimal)

Die folgende Tabelle stellt einen Auszug aus dem Arbeitsspeicher dar.

Adresse (Hex)	Inhalt (Hex)	
...	..	
4711 : 0011	72	
4711 : 0012	05	
4711 : 0013	..	
4711 : 0014	..	
4711 : 0015	..	
4711 : 0016	..	
4711 : 0017	..	
4711 : 0018	..	← Der JB folgende Befehl beginnt hier, Begründung folgt
...	..	

2) Was macht der Mikroprozessor ?
Der Mikroprozessor führt den Befehl aus, dessen Adresse bei CS:IP, also bei 4711:0011 beginnt. Anhand des Bytes des Op-Codes schaut er im Befehlssatz des Mikroprozessors nach und findet heraus, daß es sich um einen JB Befehl handelt:

72	05	oder symbolisch	JB

Es geschieht nun (laut Befehlsbeschreibung oben) folgendes:
1) IP wird um die Befehlslänge (also 2) erhöht, d.h:
`IP = 0011 + 2 = 0013`

2) Da CF = 1, wird zu IP das dem Op-Code (72) des Befehls JB folgende Byte (05) als positive Distanz 5 interpretiert (Distanz ist vorzeichenbehaftet).
Diese Distanz 5 wird als 2 Byte dargestellt (0005) und zum aktuellen Wert (0013) des Befehlszählers hinzuaddiert:

```
    0   0   1   3          (19 Dez)
+   0   0   0   5          ( 5 Dez)
----------------
    0   0   1   8          (24 Dez)
```

Der Befehlszähler hat nun also den Wert 0018
Das heißt der nächste Befehl, den der MP ausführt, beginnt an der Adresse 4711:0018

Bem:
Es ist üblich, daß bei der symbolischen Darstellung des JB Befehls der neue Wert von IP für den Fall daß CF = 1 ist, JB folgend, dargestellt wird, hier:
`JB 0018`

1.5.3.2 Beispiel eines JB Befehls

1) Annahmen
CS=8134, IP=0007, CF = 1 (sämtliche Angaben hexadezimal)

Die folgende Tabelle stellt einen Auszug aus dem Arbeitsspeicher dar.

Adresse (Hex)	Inhalt (Hex)	
...		
8134 : 0003	..	← Der JB folgende Befehl beginnt hier,
8134 : 0004	..	Begründung folgt
8134 : 0005	..	
8134 : 0006	..	
8134 : 0007	72	
8134 : 0008	FA	
...	..	

2) Was macht der Mikroprozessor ?
Der Mikroprozessor führt den Befehl aus, dessen Adresse bei CS:IP, also bei 8134:0007 beginnt. Anhand des Bytes des Op-Codes schaut er im Befehlssatz des Mikroprozessors nach und findet heraus, daß es sich um einen JB Befehl handelt:

72	FA

oder symbolisch JB

Es geschieht nun (laut Befehlsbeschreibung oben) folgendes:
1) IP wird um die Befehlslänge (also 2) erhöht, d.h:
IP = 0007 + 2 = 0009

2) Da CF = 1, wird zu IP das dem Op-Code (72) des Befehls JB folgende Byte (FA) als negative Distanz -6 interpretiert (Distanz ist vorzeichenbehaftet).
Diese Distanz -6 wird als 2 Byte dargestellt (FFFA) und zum aktuellen Wert (0009) des Befehlszählers hinzuaddiert:

```
     0   0   0   9          ( 9 Dez)
  +  1F  1F  1F  A          (-6 Dez)
 ----------------
     0   0   0   3          ( 3 Dez)
```

Der Befehlszähler hat nun also den Wert 0003
Das heißt der nächste Befehl, den der MP ausführt, beginnt an der Adresse 8134:0003

Bem:
Es ist üblich, daß bei der symbolischen Darstellung des JB Befehls der neue Wert von IP für den Fall daß CF = 1 ist, JB folgend, dargestellt wird, hier:
JB 0003

1.5.3.3 Bemerkungen

1) Vorzeichen
Ob eine Bitfolge als vorzeichenlose oder vorzeichenbehaftete Dualzahl interpretiert wird, hängt nur vom Anwenderprogramm ab.
Der Prozessor "weiß" zum Beispiel nichts davon, ob im folgenden Befehl die Bitfolge 11111111 als -1 (Dez) oder 255 (Dez) angesehen wird.

```
ADD AX, 11111111
```

Es gibt Befehle, die für vorzeichenlose und vorzeichenbehaftete Zahlen gelten, z.B:
`ADD, SUB`
Es gibt Befehle, die nur für vorzeichenlose Zahlen gelten, z.B:
`MUL , DIV`
Es gibt Befehle, die nur für vorzeichenbehaftete Zahlen gelten, z.B:
`IMUL, IDIV`

Beispiel:

```
...
CMP AX, BX
JB M1
...
```

Programmbeschreibung:
a) Für vorzeichenlose Zahlen AX, BX gilt:
„Wenn AX < BX, springe zu M1“
Wenn nämlich AX < BX ist, wird durch CMP AX, BX das CF = 1.

b) Für vorzeichenbehaftete Zahlen AX, BX gilt **NICHT**:
„Wenn AX < BX, springe zu M1“
Man betrachte dazu folgende Gegenbeispiele:

```
CMP AX, BX
```

AX	BX	Ergebnis	CF
0001 (+1)	FFFE (-2)	0003 (+3)	1
FFFE (-2)	0001 (+1)	FFFD (-3)	0

c) Damit für vorzeichenbehaftete Zahlen AX, BX gilt:
„Wenn AX < BX, springe zu M1“
muß der Befehl JB zu JL (Jump if Less) abgeändert werden, der für vorzeichenbehaftete Zahlen gilt. Also:

```
...
CMP AX, BX
JL M1
...
```

3) Anfangszustand
Nach dem Einschalten (bzw nach einem Hardware-Reset) des Mikroprozessors 8086 stellt sich automatisch (hardwaremäßig) folgender Anfangszustand ein:

```
DS = 0000
SS = 0000
ES = 0000
CS = FFFF
IP = 0000
```

1.6 Die Intelkonvention

1.6.1 Motivation:

Im Operandenteil eines Maschinenbefehls können sich 2 Bytes befinden (A und B genannt) , die eine Adresse oder eine Konstante darstellen.

Annahme:
A = 00000000,
B = 11111111
A sei das erste Byte (d.h. mit der niederen Adresse im Arbeitsspeicher bzw. das dem Op-Code nähere Byte von den beiden).
B sei das zweite Byte (d.h. mit der höheren Adresse im Arbeitsspeicher bzw. das dem Op-Code fernere Byte von den beiden).

Frage:
Wie bastelt sich dann der Mikroprozessor aus diesen zwei Bytes eine Adresse bzw. eine Konstante ?
Es gibt 2 Möglichkeiten:
Die Adresse oder Konstante könnte
AB = 0000000011111111 oder
BA = 1111111100000000 heißen.

1.6.2 Bemerkung

In den 2 Byte großen Registern (z.B. AX) des Mikroprozessors ist das Byte mit der niedrigeren Adresse das sogenannte Low Byte, d.h. das Byte, das mit der Bitnummer 0 beginnt.
Das Byte mit der höheren Adresse ist das sogenannte High Byte, d.h. das Byte, das mit der Bitnummer 8 beginnt.

1.6.3 Lesen

Wenn bei der Abarbeitung eines Befehls zwei Bytes aus dem Speicher (Arbeitsspeicher oder Register) gelesen werden und daraus vom 8086 / 8088 er Mikroprozessor eine Adresse (besser Adressoffset) oder eine Konstante (also eine 2 Byte große Zahl) gebastelt wird, dann besteht der niederwertigere Teil dieser Zahl aus dem Byte im Speicher mit der niedrigeren Adresse und der höherwertigere Teil der Zahl aus dem Byte im Speicher mit der höheren Adresse.

1.6.4 Schreiben

Die Intelkonvention besagt folgendes:
Wenn eine 2 Byte große Zahl durch den 8086 / 8088 er Mikroprozessor in den Speicher (Arbeitsspeicher oder Register) geschrieben wird, dann wird das niederwertigere Byte dieser Zahl in den Speicher mit der niederen Adresse geschrieben und das höherwertigere Byte dieser Zahl in den Speicher mit der höheren Adresse geschrieben.

1.6.4.1.1 Beispiel

eines MOV Befehls (der Register BX und Speicheroperand 1234 verwendet)

1) Annahmen

```
DS=7312H, CS=7441H, IP=0000
```

Die folgende Tabelle stellt einen Auszug aus dem Arbeitsspeicher dar.

Adresse (Hex)	Inhalt (Hex)
. . .	
7312 : 1234	E1
7312 : 1235	F2
. . .	
. . .	
7441 : 0000	8B
7441 : 0001	1E
7441 : 0002	34
7441 : 0003	12
. . .	

2) Was macht der Mikroprozessor ?

Der Mikroprozessor führt den Befehl aus, dessen Adresse bei CS:IP, also bei 7441:0000 beginnt. Anhand der zwei Bytes des Op-Codes (Ein Byte reicht nicht, warum ?) schaut er im Befehlssatz des Mikroprozessors nach und findet heraus, daß es sich um einen MOV Befehl handelt, genauer: (siehe Befehlssatz)

8B	1E	34	12

oder symbolisch `MOV BX, [1234]`

34 ist das Low Byte (weil es an der niedereren Adresse steht als das Byte 12) des Offsets des Speicheroperanden und 12 ist das High Byte des Offsets des Speicheroperanden, den der MOV Befehl in das Register BX kopiert.

MOV kopiert nun das Low Byte des Worts, das an der Adresse 7312:1234 beginnt (E1) in das Low Byte von BX (also BL) und das High Byte (F2) in das High Byte von BX (also BH). Nach dem Ende der Befehlsabarbeitung steht dann in BX:

BX	F2	E1

Ein Ausschnitt aus dem Arbeitsspeicher

1234 : 0006	A1	`mov  ax,[0010]`
1234 : 0007	10	
1234 : 0008	00	
1234 : 0009	05	`add  ax,0001`
1234 : 000A	01	
1234 : 000B	00	
1234 : 000C	B9	`mov  cx,0001`
1234 : 000D	01	
1234 : 000E	00	
1234 : 000F	BA	`mov dx,0000`
1234 : 0010	00	
1234 : 0011	00	
1234 : 0012	3B	`cmp cx,ax`
1234 : 0013	C8	
1234 : 0014	72	`jb 0018`
1234 : 0015	02	
1234 : 0016	EB	`jmp 0025`
1234 : 0017	0D	
1234 : 0018	83	`add cx,0001`
1234 : 0019	C1	
1234 : 001A	01	
1234 : 001B	03	`add dx,[0012]`
1234 : 001C	16	
1234 : 001D	12	
1234 : 001E	00	
1234 : 001F	3B	`cmp cx,ax`
1234 : 0020	C8	
1234 : 0021	72	`jb 0018`
1234 : 0022	F5	
1234 : 0023	EB	`jmp 0025`
1234 : 0024	00	
1234 : 0025	89	`mov [0014],dx`
1234 : 0026	16	
1234 : 0027	14	
1234 : 0028	00	
1234 : 0029	B4	`mov ah,4C`
1234 : 002A	4C	
1234 : 002B	CD	`int 21`
1234 : 002C	21	
. . . . .	. .	. . . .
. . . . .	. .	. . . .
4711 : 000F	90	
4711 : 0010	03	
4711 : 0011	00	
4711 : 0012	06	
4711 : 0013	00	
4711 : 0014	11	
4711 : 0015	78	

Der Anfangswert (Inhalt) folgender Register ist:

CS = 1234
DS = 4711
IP = 0006

Der Anfangswert (Inhalt) der anderen Register ist irgend ein beliebiger Wert.

Was macht der Prozessor ?

Bem:
Alle Inhalte (und die Adressen sowieso) von hier bis zum Ende dieses Kapitels sind hexadezimal !

Aufgabenblatt (Bitte Bleistift und Radiergummi verwenden !)

cs : 0006	A11000			mov	ax,[0010]				
cs : 0009	050100			add	ax,0001				
cs : 000C	B90100			mov	cx,0001				
cs : 000F	BA0000			mov	dx,0000				
cs : 0012	3BC8			cmp	cx,ax				
cs : 0014	7202			jb	0018				
cs : 0016	EB0D			jmp	0025				
cs : 0018	83C101			add	cx,0001				
cs : 001B	03161200			add	dx,[0012]				
cs : 001F	3BC8			cmp	cx,ax				
cs : 0021	72F5			jb	0018				
cs : 0023	EB00			jmp	0025				
cs : 0025	89161400			mov	[0014],dx				
cs : 0029	B44C			mov	ah,4C				
cs : 002B	CD21			int	21				
ds : 0000	A1	B2	C3	D4	E5	F6	07	18	
ds : 0008	29	3A	4B	5C	6D	7E	8F	90	
ds : 0010	03	00	06	00	11	78	13	13	
ds : 0018	13	24	35	46	57	68	79	8A	
ds : 0020	9B	AC	BD	CE	DF	E0	F1	02	

Zur Tabelle links:
Der vorige Ausschnitt aus dem Arbeitsspeicher wird hier anders dargestellt.
Für die Anfangswerte (Inhalte) der Register gilt das gleiche wie in der vorigen Darstellung.

Geben Sie nach Ausführung jedes einzelnen Befehls in der Tabelle unten die (noch fehlenden) Inhalte der Register und Speicheradressen an.
Was macht das Programm im Endeffekt ?
(Ein • bezeichnet das durch den Befehl betroffene Register oder Adresse)

Dynamisches Verhalten der Registerinhalte und der Speicherinhalte:

IP (alt)	IP (neu)	AX	CX	DX	CF	DS: 0010	DS: 0011	DS: 0012	DS: 0013	DS: 0014	DS: 0015	Befehl
0006	•0009	•0003	?	?	?	03	00	06	00	11	78	MOV AX,[0010]
0009	•000C	•0004	?	?	0	03	00	06	00	11	78	ADD AX,0001
000C												

Lösung:

Dynamisches Verhalten der Registerinhalte und der Speicherinhalte:

IP (alt)	IP (neu)	AX	CX	DX	CF	DS: 0010	DS: 0011	DS: 0012	DS: 0013	DS: 0014	DS: 0015	Befehl
0006	•0009	•0003	?	?	?	03	00	06	00	11	78	MOV AX,[0010]
0009	•000C	•0004	?	?	•0	03	00	06	00	11	78	ADD AX,0001
000C	•000F	0004	•0001	?	0	03	00	06	00	11	78	MOV CX,0001
000F	•0012	0004	0001	•0000	0	03	00	06	00	11	78	MOV DX,0000
0012	•0014	0004	0001	0000	•1	03	00	06	00	11	78	CMP CX,AX
0014	•0018	0004	0001	0000	1	03	00	06	00	11	78	JB 0018
0018	•001B	0004	•0002	0000	•0	03	00	06	00	11	78	ADD CX,0001
001B	•001F	0004	0002	•0006	•0	03	00	06	00	11	78	ADD DX,[0012]
001F	•0021	0004	0002	0006	•1	03	00	06	00	11	78	CMP CX,AX
0021	•0018	0004	0002	0006	1	03	00	06	00	11	78	JB 0018
0018	•001B	0004	•0003	0006	•0	03	00	06	00	11	78	ADD CX,0001
001B	•001F	0004	0003	•000C	•0	03	00	06	00	11	78	ADD DX,[0012]
001F	•0021	0004	0003	000C	•1	03	00	06	00	11	78	CMP CX,AX
0021	•0018	0004	0003	000C	1	03	00	06	00	11	78	JB 0018
0018	•001B	0004	•0004	000C	•0	03	00	06	00	11	78	ADD CX,0001
001B	•001F	0004	0004	•0012	•0	03	00	06	00	11	78	ADD DX,[0012]
001F	•0021	0004	0004	0012	•0	03	00	06	00	11	78	CMP CX,AX
0021	•0023	0004	0004	0012	0	03	00	06	00	11	78	JB 0018
0023	•0025	0004	0004	0012	0	03	00	06	00	11	78	JMP 0025
0025	•0029	0004	0004	0012	0	03	00	06	00	**•12**	**•00**	MOV [0014],DX

Programmbeschreibung:
Die Zahl (der Inhalt der 2 Byte) , die an den Adressen DS:0010 und DS:0011 steht, wird mit der Zahl (der Inhalt der 2 Byte), die an den Adressen DS:0012 und DS:0013 steht, multipliziert und an die Adressen DS:0014 und DS:0015 kopiert (abgelegt).
Der alte Inhalt der Adressen DS:0014 und DS:0015 wird überschrieben.

Kurz:

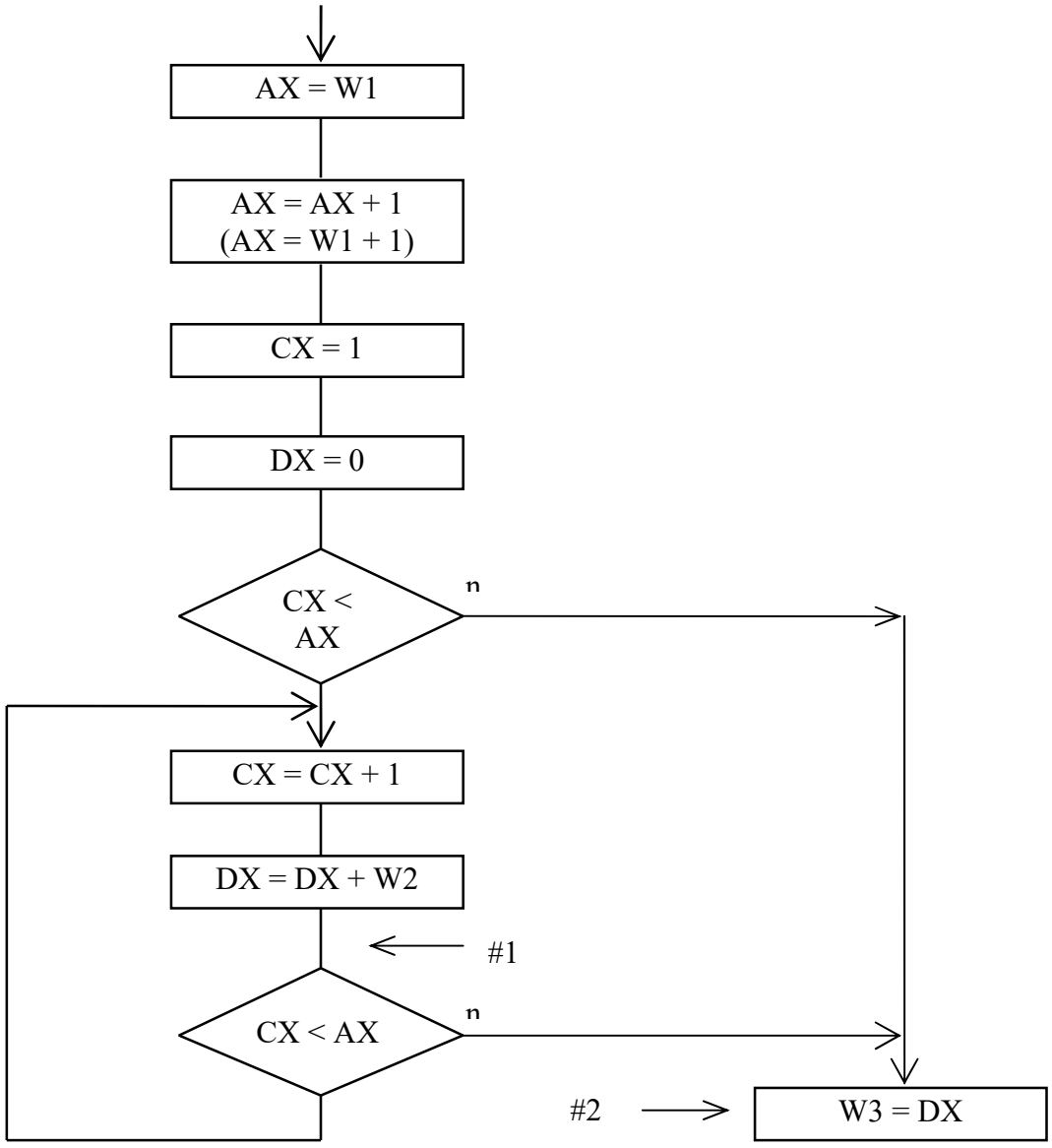
[DS:0010, DS:0011] * [DS:0012, DS:0013] → [DS:0014, DS:0015]

Auf die obige Speicherbelegung bezogen:

3 (Hex) * 6 (Hex) → 12 (Hex)

Bemerkung:
Der alte Inhalt der Adressen DS:0014 und DS:0015 , also 1178
wird überschrieben durch 1200

Darstellung des Programmablaufs in einem Flußdiagramm:



Bemerkungen zum Flußdiagramm:
1) der Inhalt der 2 Bytes ab Adresse 4711:0010 wird W1 genannt,
der Inhalt der 2 Bytes ab Adresse 4711:0012 wird W2 genannt,
der Inhalt der 2 Bytes ab Adresse 4711:0014 wird W3 genannt

W1, W2, W3 sollen vorzeichenlose Zahlen sein.

Dynamisches Verhalten der Registerinhalte CX und DX (an den Stellen #1 und #2)

Kommentar	#1	#1	#1	(vorletzter #1 Schleifen-durchgang)	(letzter #2 Schleifen-durchgang bzw. nach Verlassen der Schleife)
CX	2	3	....	AX - 1 (= W1)	AX (= W2)
DX	W2	2 * W2	....	(W1-1)*W2	W1 * W2

weitere Bemerkungen (siehe Befehlssatz):
1) MOV verändert keine Flags
2) ADD verändert bestimmte Flags (z.B. wird CF verändert)

Fragen:
1) Welche Berechnung braucht mehr Rechenzeit:
3 * 6 oder 6 * 3

Braucht das Programm unabhängig von der Größe der 2 zu multiplizierenden Zahlen immer gleich viel Rechnezeit ?

2) Was hat es für Konsequenzen, wenn man für die Daten jeweils nur 1 Byte reserviert ?

3) Was ist der Geltungsbereich der Programmbeschreibung:
Gilt die Programmbeschreibung z.B. nur für vorzeichenlose Zahlen W1 und W2, oder dürfen sie vorzeichenbehaftet sein ?
Wie groß dürfen die Zahlen W1 und W2 sein ?

4) Welche Busse (Adressbus, Datenbus) werden in den einzelnen Befehlen verwendet ?

1.7 Das Zeitverhalten

Bei einem Mikroprozessor werden alle Elemente synchron geschaltet, die logischen Operationen des Systems zu bestimmten Zeitpunkten vollzogen. Dies geschieht durch den Taktgeber (Clock generator).

Die Zeit, die ein Mikroprozessor braucht, um einen Befehl auszuführen, hängt ab von der

(1) Taktfrequenz
(2) Der Anzahl der Taktzyklen, die für den Befehl benötigt werden.

Beispiel:
Der 8086 / 8088 er Mikroprozessor im IBM - PC arbeitet mit einer Taktfrequenz von 4,77 MHz.
Ein Taktzyklus benötigt somit:

$$T = \frac{1}{f} = \frac{1}{4,77\,\text{MHz}} = 210\,\text{ns}$$

Der Befehl
`MOV AX, a`
benötigt 10 Takte, braucht also 10 * 210 ns,
wobei a einen Direktwert (Konstante) bedeutet.

Die Anzahl der Taktzyklen, die zusätzlich benötigt werden, um falls erforderlich die effektive Adresse (EA) eines Speicheroperanden zu bilden, hängt vom Befehl ab.
Die Anzahl der Taktzyklen eines Befehls sind dem Prozessorhandbuch zu entnehmen.

weiteres Beispiel:
`MOV BX, CX`
benötigt 2 Takte

Ein Auszug des 8086 / 8088 Befehlssatzes

Erklärungen zu den folgenden Assembler-Befehlen

Symbolische Adressen und ihre Inhalte in den folgenden Assemblerbefehlen:

Address-Offset	symbolische Adrese	Daten - größe	Inhalt
0000	BYTE_VAR	DB	?
0001	WORT_VAR	DW	?
0003	WORT	DW	?
0005	BYTE_TAB	DB	?
0069	WORT_TAB	DW	?
01F9	VARIABLE	DB	?
0299	START_ADR	DW	?
029B	VEC	DD	?
042B	B_STR1	DB	?
047B	W_STR1	DW	?
0583	DWORT_VAR	DD	?
0587	Q_STRING_ADR	DD	B_STR1

```
EQUATE_WERT EQU 5
```

Codierung der Register

W = 1	w = 0	reg
AX	AL	000
CX	CL	001
DX	DL	010
BX	BL	011
SP	AH	100
BP	CH	101
SI	DH	110
DI	BH	111

Segmentregister

sreg	Segmentregister
00	ES
01	CS
10	SS
11	DS

Zustand der Flags nach Abarbeitung des Befehls

unverändert:	-
undefiniert	U
verändert	X

1) Das zweite Byte des Op - Codes
Falls der Op - Code aus 2 Bytes besteht, wird das 2. Byte im Befehlssatz wie folgt bezeichnet:

mod (2 Bit)	reg (3 Bit)	r/m (3 Bit)

Dieses Byte legt die Art der Operanden fest (Adresse, Direktwer, Register)
Genauerses dazu steht im Assembler-Handbuch.

2) Gestrichelte Kästchen
Maschinenbefehlsbytes, die bei jeder Form eines Befehls vorhanden sind, sind in Kästchen mit durchgehenden Linien dargestellt.
Maschinenbefelsbytes, die z.B. in Abhängigkeit von der Adressierungsart vorhanden sein können oder auch nicht, sind in Kästchen mit gepunkteten Linien eingezeichnet.

JAE

JAE (jump if above or equal)
Sprung, wenn größer oder gleich

ist gleich mit JNB (jump if not below) und verzweigt zur angegebenen Adresse (Marke), wenn CF = 0 ist. Die angegebene Adresse (Marke) darf nicht weiter entfernt sein als -128 bis +127 Bytes. Above und below beziehen sich auf Zahlen ohne Vorzeichen.

Operation:
Wenn CF = 0 ist, dann wird die im Befehl enthaltene Distanzadresse auf 16 Bit vorzeichenerweitert und zum IP addiert.

```
IF (CF) = 0 then
  (IP) ← (IP) + distanz
```

Maschinencode:

0111 0011	distanz

Zeit:
Sprung wird ausgeführt: 16 Takte
Sprung wird nicht ausgeführt: 4 Takte

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Beispiele:
```
CMP AL, 0FH    ; vorzeichenlos
JAE Marke      ; Sprung nach Marke, falls
                 AL >= 0FH
SUB AL, BYTE_VAR
JAE WEITER
```

JBE

JBE (jump if below or equal)
Sprung, wenn kleiner oder gleich

ist gleich mit JNA (jump if not above) und verzweigt zur angegebenen Adresse (Marke), wenn CF = 1 oder ZF = 1ist. Die angegebene Adresse (Marke) darf nicht weiter entfernt sein als -128 bis +127 Bytes. Above und below beziehen sich auf Zahlen ohne Vorzeichen.

Operation:
Wenn CF = 1 oder ZF = 1 ist, dann wird die im Befehl enthaltene Distanzadresse auf 16 Bit vorzeichenerweitert und zum IP addiert.

```
IF (CF) = 1 or (ZF) = 1 then
  (IP) ← (IP) + distanz
```

Maschinencode:

0111 0110	distanz

Zeit:
Sprung wird ausgeführt: 16 Takte
Sprung wird nicht ausgeführt: 4 Takte

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Beispiele:
```
CMP AL, 39H
JBE Ziffer
SUB AL, BYTE_VAR
JBE WEITER
```

JA

JA (jump if above)
Sprung, wenn größer

ist gleich mit JNBE (jump if not below or equal) und verzweigt zur angegebenen Adresse (Marke), wenn CF = 0 und ZF = 0sind. Das Sprungziel darf nicht weiter entfernt sein als -128 bis +127 Bytes. Above und below beziehen sich auf Zahlen ohne Vorzeichen.

Operation:
Wenn CF = 0 und ZF = 0 ist, dann wird die im Befehl enthaltene Distanzadresse auf 16 Bit vorzeichenerweitert und zum IP addiert.

```
IF (CF) = 0 and (ZF) = 0 then
  (IP) ← (IP) + distanz
```

Maschinencode:

0111 0111	distanz

Zeit:
Sprung wird ausgeführt: 16 Takte
Sprung wird nicht ausgeführt: 4 Takte

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Beispiele:
```
CMP AL, 0      ; vorzeichenlos
JA Marke       ; Sprung nach Marke, falls
                 AL > 0
SUB AL, Direkt_Wert
JA Weiter
```

JB

JB (jump if below)
Sprung, wenn kleiner

ist gleich mit JNAE (jump if not above or equal) und verzweigt zur angegebenen Adresse (Marke), wenn CF = 1 ist. Die angegebene Adresse (Marke) darf nicht weiter entfernt sein als -128 bis +127 Bytes. Above und below beziehen sich auf Zahlen ohne Vorzeichen.

Operation:
Wenn CF = 1 ist, dann wird die im Befehl enthaltene Distanzadresse auf 16 Bit vorzeichenerweitert und zum IP addiert.

```
IF (CF) = 1 then
  (IP) ← (IP) + distanz
```

Maschinencode:

0111 0010	distanz

Zeit:
Sprung wird ausgeführt: 16 Takte
Sprung wird nicht ausgeführt: 4 Takte

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Beispiele:
```
CMP AL, 3AH
JB Ziffer
SUB AL, BYTE_VAR
JB WEITER
```

JE

JE (jump if equal)
Sprung, wenn gleich

ist gleich mit JZ (jump if zero) und verzweigt zur angegebenen Adresse (Marke), wenn ZF =1 ist. Die angegebene Adresse (Marke) darf nicht weiter entfernt sein als -128 bis +127 Bytes.

Operation:
Wenn ZF = 1 ist, dann wird die im Befehl enthaltene Distanzadresse auf 16 Bit vorzeichenerweitert und zum IP addiert.

```
IF (ZF) = 1 then
  (IP) ← (IP) + distanz
```

Maschinencode:

0111 0100	distanz

Zeit:
Sprung wird ausgeführt: 16 Takte
Sprung wird nicht ausgeführt: 4 Takte

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Beispiele:
```
CMP AL, 0
JE Marke
SUB AL, BYTE_VAR
JE Weiter
```

JNE

JNE (jump if not equal)
Sprung, wenn nicht gleich

ist gleich mit JNZ (jump if not zero) und verzweigt zur angegebenen Adresse (Marke), wenn ZF =0 ist. Die angegebene Adresse (Marke) darf nicht weiter entfernt sein als -128 bis +127 Bytes.

Operation:
Wenn ZF = 0 ist, dann wird die im Befehl enthaltene Distanzadresse auf 16 Bit vorzeichenerweitert und zum IP addiert.

```
IF (ZF) = 0 then
  (IP) ← (IP) + distanz
```

Maschinencode:

0111 0101	distanz

Zeit:
Sprung wird ausgeführt: 16 Takte
Sprung wird nicht ausgeführt: 4 Takte

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Beispiele:
```
CMP AL, 0
JNE Marke
SUB AL, BYTE_VAR
JNE Weiter
```

JMP

JMP (jump)
Unbedingter Sprung

bewirkt einen unbedingten Sprung auf eine Adresse (Marke). Es gibt sechs unterschiedliche Sprungmöglichkeiten:

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Direkter kurzer Intrasegment-Sprung

Maschinencode:

1110 1011	distanz

distanz wird auf 16 Bit erweitert und muß zwischen -128 und +127 liegen. Diese Distanz wird dann auf den IP addiert.

```
(IP) ← (IP) + distanz
```

Zeit: 15 Takte

Beispiele:
```
0000    EB 14   JMP SHORT MARKE
0016    90      MARKE: NOP
```

Direkter langer Intrasegment-Sprung

Maschinencode:

1110 1001	distanz - low	distanz - low

Die Distanz steht als 16-Bit Größe im zweiten und dritten Befehlsbyte. Damit sind beliebig weite Sprünge innerhalb eines Segments möglich.

Zeit: 15 Takte

Beispiele:
```
0017    E9 0146 R       JMP Weit
0146    90              Weit: NOP
```

Indirekter Intrasegment-Sprung (über Register)

Maschinencode:

1111 1111	11100 reg

Sprung innerhalb eines Codesegments zu einem Befehl, dessen Offset im angegebenen Register steht.

Zeit: 11 Takte

Beispiele:
```
FF E0   JMP AX
FF E6   JMP SI
FF E5   JMP BP
```

Indirekter Intrasegment-Sprung (über Speicher)

Maschinencode:

1111 1111	mod 100 r/m	adr-low	adr-high

Sprung innerhalb eines Codesegments zu einem Befehl, dessen Offset in dem adressierten Speicherwort steht.

Zeit: 18 + EA Takte

Beispiele:
```
FF A7 0069 R            JMP WORT_TAB[BX]
FF 24                   JMP WORD PTR [SI]
FF 24                   JMP [SI]
```

JMP

MOV

MOV (move)
Übertragen

überträgt den Inhalt des Quelloperanden (der Rechte) in den Zieloperanden (der Linke). Der Befehl ist sehr leistungsfähig, da die Operanden Register, Speicherstellen oder Direktwerte sein können. Folgende Operandenkombinationen sind möglich:

LIOP \ REOP	Akku	Reg	Sreg	Speicher	Direktw.
Akku	ja	ja	ja	ja	ja
Reg	ja	ja	ja	ja	ja
Sreg (<>CS)	ja	ja	nein	ja	nein
Speicher	ja	ja	ja	nein	ja

Operation:
Der Inhalt des Quelloperanden (der Rechte) wird in den Zieloperanden (der Linke) übertragen.

```
(LIOP) ← (REOP)
```

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Übertragung von Register nach Register

Maschinencode:

1000 101w	11 reg reg

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 2 Takte

Beispiele:
```
8B C3           MOV AX, BX
8A E0           MOV AH, AL
8A CF           MOV CL, BH
8B EC           MOV BP, SP
8B FF           MOV DI, DI
```

Übertragung von Speicheroperand nach Register

Maschinencode:

1000 101w	mod reg r/m	adr - low	adr - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 8 + EA Takte

Beispiele:
```
8B 1E 0001 R    MOV BX, WORT_VAR
8A 99 0005 R    MOV BL, BYTE_TAB[BX + DI]
8B 4E 08        MOV CX, [BP + 8]
8A 26 0000 R    MOV AH,BYTE_VAR
```

Übertragung von Direktwert nach Speicheroperand

Maschinencode:

1100 011w	mod 000 r/m	adr - low	adr - high	data - low
				data - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 10 + EA Takte

Beispiele:
```
C6 06 0000 R FF    MOV BYTE_VAR, 0FFH
C7 06 0001 R 0005 MOV WORT_VAR, EQUATE_WERT
C7 81 01FD R C420       MOV VARIABLE [BX+DI+4], -
1532
C7 06 0003 R 01F9 R     MOV WORT, OFFSET VARIABLE
```

Übertragung von Akkumulator nach Speicheroperand

Maschinencode:

1010 001w	adr - low	adr - high

w = 0 für 8 Bit und AL und w = 1 für 16 Bit und AX

Zeit: 10 Takte

Beispiele:
```
A3 0001 R       MOV WORT_VAR, AX
A2 0000 R       MOV BYTE_VAR, AL
```

Übertragung von Speicheroperand nach Akkumulator

Maschinencode:

1010 000w	adr - low	adr - high

w = 0 für 8 Bit und AL und w = 1 für 16 Bit und AX

Zeit: 10 Takte

Beispiele:
```
A1 0001 R       MOV AX, WORT_VAR
A0 0000 R       MOV AL, BYTE_VAR
```

Übertragung von Register nach Speicheroperand

Maschinencode:

1000 100w	mod reg r/m	adr - low	adr - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 9 + EA Takte

Beispiele:
```
88 36 0000 R    MOV BYTE_VAR, DH
89 0E 0003 R    MOV WORT, CX
88 90 0005 R    MOV BYTE_TAB[BX+SI], DL
89 50 02        MOV [BX+SI+2], DX
```

Übertragung von Direktwert nach Register

Maschinencode:

1011 w reg	daten - low	daten - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 4 Takte

Beispiele:
```
B8 00A3         MOV AX, 0A3H
B3 05           MOV BL, EQUATE_WERT
B9 FF80         MOV CX, -128
B2 02           MOV DL, TYPE VARIABLE
```

Übertragung von Register nach Segmentregister

Maschinencode:

1000 1110	110 sreg reg	sreg <> 1

Zeit: 2 Takte

Beispiele:
```
8E D8           MOV DS, AX
8E C3           MOV ES, BX
8E D5           MOV SS, BP
8E CA           MOV CS, DX ;(unzulässig)
```

ADD

ADD (Addition)
Addiere

addiert die zwei Operanden und speichert das Ergebnis im Bestimmungsoperanden (dem linken Operanden) ab. Erlaubte Operandenpaare sind:

LIOP / REOP

	Akku	Reg	Sreg	Speicher	Direktw.
Akku	ja	ja	nein	ja	ja
Reg	ja	ja	nein	ja	ja
Sreg	nein	nein	nein	nein	nein
Speicher	ja	ja	nein	nein	ja

Operation:
Die Summe der beiden Operanden wird im Bestimmungsoperanden (LIOP) abgelegt.

```
(LIOP) ← (LIOP) + (REOP)
```

Flags:	O	D	I	T	S	Z	A	P	C
	X	-	-	-	X	X	X	X	X

Register nach Speicher

Maschinencode:

0000 000w	mod reg r/m	adr - low	adr - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 16 + EA Takte

Beispiele:
```
01 16 0003 R   ADD WORT, DX
01 84 0069 R   ADD WORT_TAB[SI], AX
00 10          ADD [SI][BX], DL
01 1F          ADD [BX], BX
```

Direktwert nach Akkumulator

Maschinencode:

0000 010w	data - low	data - high

w = 0 für 8 Bit und AL und w = 1 für 16 Bit und AX

Zeit: 4 Takte

Beispiele:
```
04 03          ADD AL, 3
05 1234        ADD AX, 1234H
05 0005        ADD AX, EQUATE_WERT
04 05          ADD AL, EQUATE_WERT
```

Register nach Register

Maschinencode:

0000 001w	11 reg reg

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 3 Takte

Beispiele:
```
03 C6          ADD AX, SI
03 FB          ADD DI, BX
02 EB          ADD CH, BL
02 E0          ADD AH, AL
```

Speicher nach Register

Maschinencode:

0000 001w	mod reg r/m	adr - low	adr - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 9 + EA Takte

Beispiele:
```
03 16 0001 R   ADD DX, WORT_VAR
03 85 0069 R   ADD AX, WORT_TAB[DI]
02 80 0005 R   ADD AL, BYTE_TAB[BX+SI]
02 2F          ADD CH, [BX]
```

Direktwert nach Register

Maschinencode:

1000 00sw	11 000 reg	data - low	data - high

w = 0 für 8 Bit s = 0 falls keine data-Erweiterung erforderlich
w = 1 für 16 Bit s = 1 falls data-Erweiterung erforderlich

Zeit: 4 Takte

Beispiele:
```
83 C2 03       ADD DX, 3
81 C3 EDCC     ADD BX, -1234H
80 C1 05       ADD CL, EQUATE_WERT
81 C7 00BC     ADD DI, BC00
```

Direktwert nach Speicher

Maschinencode:

1000 00sw	mod 000 r/m	adr - low	adr - high	data - low
				data - high

w = 0 für 8 Bit s = 0 falls keine data-Erweiterung erforderlich
w = 1 für 16 Bit s = 1 falls data-Erweiterung erforderlich

Zeit: 17 + EA Takte
Beispiele:
```
83 06 0001 R 03 ADD WORT_VAR, 3
83 84 0069 R 01 ADD WORT_TAB[SI], 1
80 07 05      ADD BYTE PTR [BX], EQUATE_WERT
83 04 05      ADD WORD PTR [SI], EQUATE_WERT
```

CMP

CMP (compare two operands)
zwei Operanden vergleichen

führt eine Subtraktion der beiden Operanden durch, ohne das Ergebnis abzulegen. Es werden nur die Flags verändert. Beide Operanden müssen vom selben Typ sein (Byte oder Wort). Direktwerte in Bytegröße werden vorzeichenerweitert, falls sie mit einem Wort verglichen werden. Erlaubte Operandenwerte sind:

LIOP / REOP

	Akku	Reg	Sreg	Speicher	Direktw.
Akku	ja	ja	nein	ja	ja
Reg	ja	ja	nein	ja	ja
Sreg	nein	nein	nein	nein	nein
Speicher	ja	ja	nein	nein	ja

Operation:
Der rechte Operand wird vom linken Operanden subtrahiert. Das Ergebnis wird nicht abgelegt. Die Flags werden verändert.

```
(FLAGS) ← (LIOP) - (REOP)
```

Flags:	O	D	I	T	S	Z	A	P	C
	X	-	-	-	X	X	X	X	X

Register mit Register

Maschinencode:

0011 101w	11 reg reg

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 3 Takte

Beispiele:
```
3B C2          CMP AX, DX
3B F5          CMP SI, BP
3A F9          CMP BH, CL
```

Speicher mit Register

Maschinencode:

0011 100w	mod reg r/m	adr - low	adr - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 9 + EA Takte

Beispiele:
```
39 36 0003 R   CMP WORT, SI
38 06 0000 R   CMP BYTE_VAR, AL
39 17          CMP [BX], DX
39 80 0069 R   CMP WORT_TAB[BX+SI], AX
```

Register mit Speicher

Maschinencode:

0011 101w	mod reg r/m	adr - low	adr - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 9 + EA Takte

Beispiele:
```
3B 36 0001 R       CMP SI, WORT_VAR
3B 97 0069 R       CMP DX, WORT_TAB[BX]
3A 2E 0000 R       CMP CH, BYTE_VAR
3B 1F              CMP BX, [BX]
```

Akkumulator mit Direktwert

Maschinencode:

0011 110w	data - low	data - high

w = 0 für 8 Bit und w = 1 für 16 Bit

Zeit: 4 Takte

Beispiele:
```
3C 06          CMP AL, 6
3C 05          CMP AL, EQUATE_WERT
3D FFFF        CMP AX, -1
3D FFFF        CMP AX, 0FFFFH
```

Register mit Direktwert

Maschinencode:

1000 00sw	11 111 reg	data - low	data - high

w = 0 für 8 Bit s = 0 falls keine data-Erweiterung erforderlich
w = 1 für 16 Bit s = 1 falls data-Erweiterung erforderlich

Zeit: 4 Takte

Beispiele:
```
80 FF 0F       CMP BH, 0FH
80 F9 05       CMP CL, EQUATE_WERT
83 FA FF       CMP DX, -1
81 FE 0C00     CMP SI, 3 * 1024
```

Speicher mit Direktwert

Maschinencode:

1000 00sw	mod 111 r/m	adr - low	adr - high	data - low
				data - high

w = 0 für 8 Bit s = 0 falls keine data-Erweiterung erforderlich
w = 1 für 16 Bit s = 1 falls data-Erweiterung erforderlich

Zeit: 10 + EA Takte

Beispiele:
```
81 3E 0001 R 00FF    CMP WORT_VAR, 0FFH
80 BC 0005 R 11      CMP BYTE_TAB[SI], 17
83 3F FF             CMP WORD PTR[BX], -1H
80 3C FF             CMP BYTE_PTR[SI], 0FFH
```

MUL

MUL multiply accumulator by register or memory; nsigned)
Vorzeichenlose Multiplikation des Akkumulators mit einem Speicheroperand oder Register)

führt eine Multiplikation mit folgenden Operandenkombinationen aus:

Multiplikant	Multiplikator	→	Produkt
8 Bit AL	8 Bit Operand		16 Bit AX
16 Bit AX	16 Bit Operand		32 Bit DX : AX

CF und OF werden gesetzt, wenn der HIGH - Teil des Ergebnisses nicht 0 ist.

Operation:

```
(Produkt) ← (Multiplikant) * (Multiplikator)
if HIGH - Teil = 0 then
  (CF) ← 0 und (OF) ← 0
else
  (CF) ← 1 und (OF) ← 1
```

Maschinencode:

1111 011w	mod 100 r/m	adr - low	adr - high

w = 0 für Multiplikant = AL
Multiplikator = 8 Bit-Register oder 8 Bit-

Speicheroperand
Produkt = AX

w = 1 für Multiplikant = AX
Multiplikator = 16 Bit-Register oder 16 Bit-

Speicheroperand

Produkt = DX + AX

Zeit: 8 Bit 70 bis 83 + EA Takte
16 Bit 118 bis 139 + EA Takte

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

Beispiele:

a) Multiplikation von Byte mit Byte

```
A0 0000 R       MOV AL, BYTE_VAR
F6 A4 0005 R    MUL BYTE_TAB[SI]
                          ; Ergebnis in AX
A0 0000 R       MOV AL, BYTE_VAR
F6 E3           MUL BL      ; Ergebnis in AX
```

b1) Multiplikation von Byte mit einem Wort

```
A0 0000 R       MOV AL, BYTE_VAR
B4 00           MOV AH, 0
F7 26 0001 R    MUL WORT_VAR
; Ergebnis HIGH-Teil in DX, LOW-Teil in AX

A0 0000 R       MOV AL, BYTE_VAR
B4 00           MOV AH, 0
F7 E7           MUL DI
; Ergebnis HIGH-Teil in DX, LOW-Teil in AX
```

b2) Multiplikation von Wort mit einem Wort

```
A1 0001 R       MOV AX, WORT_VAR
F7 25           MUL WORD PTR[DI]
; Ergebnis HIGH-Teil in DX, LOW-Teil in AX

A1 0001 R       MOV AX, WORT_VAR
F7 E3           MUL BX
; Ergebnis HIGH-Teil in DX, LOW-Teil in AX
```

DIV

DIV (division; unsigned)
Division vorzeichenloser Operanden

In Abhängigkeit vom Operandentyp wird entweder ein Wort in AX durch einen Bytedivisor oder ein Doppelwort in DX:AX durch einen Wortdivisor geteilt. Quotient und Rest werden bei der Wort:Byte Division in AL und AH, bei der Doppelwort:Wort Division in AX und DX zur Verfügung gestellt.

Dividend /	Divisor	→	Quotient	Rest
16 Bit AX	8 Bit Operand		8 Bit AL	8 Bit AH
32 Bit DX : AX	16 Bit Operand		16 Bit AX	16 Bit DX

Wenn das Ergebnis überläuft (Ergebnis > High Value z.B. bei Division durch Null oder Division durch 1), wird ein Interrupt 0 generiert.

Operation:

```
if (Divident)/(Divisor)> HIGH VALUE then
  INT 0
else
  (Quotient)← (Divident) / (Divisor)
  (Rest) ← (Divident) MOD (Divisor)
```

Maschinencode:

1111 011w	mod 110 r/m	adr - low	adr - high

w = 0 für Dividend = AH + AL (16 Bit)
Divisor = 8 Bit-Register oder 8 Bit-Speicher
Quotient = AL Rest = AH

w = 1 für Dividend = DX + AX (32 Bit)
Divisor = 16 Bit-Register oder 16 Bit-Speicher
Quotient = AX Rest = DX

Zeit: 8 Bit 80 bis 96 + EA Takte
16 Bit 144 bis 168 + EA Takte

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	U	U	U

Beispiele:

a1) Division von Wort durch Byte

```
A1 0001 R       MOV AX, WORT_VAR
F6 36 0000 R    DIV Byte_VAR

8B 84 0069 R    MOV AX, WORT_TAB[SI]
F6 F3           DIV BL ; Ergebnis in AL,
                       ; Rest in AH
```

a2) Division von Byte durch Byte

```
8A 87 0005 R    MOV AL, BYTE_TAB[BX]
B4 00           MOV AH, 0
F6 F1           DIV CL ; Ergebnis in AL,
                       ; Rest in AH
```

b1) Division von Doppelwort durch Wort

```
8B 16 0585 R    MOV DX,WORD PTR DWORT_VAR+2
A1 0583 R       MOV AX, WORD PTR DWORT_VAR
F7 F3           DIV BX ; Ergebnis in AX,
                       ; Rest in DX
```

b2) Division von Wort durch Wort

```
A1 0001 R       MOV AX, WORT_VAR
BA 0000         MOV DX, 0
F7 37           DIV WORD_PTR[BX]
                          ; Ergebnis in AX,
                          ; Rest in DX
```

IN

IN (input byte and input word)
Einlesen eines Bytes oder eines Worts

überträgt ein Byte bzw Wort von einem Eingabe-Port nach AL bzw. AX. Der Port kann durch einen Direktwert spezifiziert werden, wenn die Portadresse zwischen 0 und 255 liegt. Ist die Adresse größer als 255, muß sie vorher in das DX Register geladen werden. Dadurch ist es möglich, Ein / Ausgabeports mit einem Adressbereich von 64 K zu erreichen.

Operation:
Der Wert, der an einem Input-Port anliegt, wird in den Akkumulator übertragen.

```
(LIOP) ← REOP(REOP)
```

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Portadresse ist ein Direktwert zwischen 0 und 255

Maschinencode:

1110 010w	portadresse

w = 0 für LIOP = AL und w = 1 für LIOP = AX

Zeit: 10 Takte

Beispiele:
```
E4 19          IN AL, 25
E5 FF          IN AX, 255
```

Portadresse steht in DX

Maschinencode:

1110 110w

w = 0 für LIOP = AL und w = 1 für LIOP = AX

Zeit: 8 Takte

Beispiele:
```
BA FEA0        MOV DX, 0FEA0H
EC             IN AL, DX

BA EA88        MOV DX, 0EA88H
ED             IN AL, DX
```

OUT

OUT (output byte and output word)
Ausgabe eines Bytes bzw. eines Worts

überträgt ein Byte bzw Wort von AL bzw. AX zu einem Output-Port. Der Port kann durch einen Direktwert spezifiziert werden, wenn die Portadresse zwischen 0 und 255 liegt. Ist die Adresse größer als 255, muß sie vorher in das DX Register geladen werden. Dadurch ist es möglich, Ein / Ausgabeports mit einem Adressbereich von 64 K zu erreichen.

Operation:
Der Inhalt des Akkumulators wird an einen Output-Port übertragen.

```
(PORT) ← (REOP)
```

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

Portadresse ist ein Direktwert zwischen 0 und 255

Maschinencode:

1110 011w	portadresse

w = 0 für AL und w = 1 für AX

Zeit: 10 Takte

Beispiele:
```
E6 19          OUT 25, AL
E7 E0          OUT 0E0H, AX
```

Portadresse steht in DX

Maschinencode:

1110 111w

w = 0 für AL und w = 1 für AX

Zeit: 8 Takte

Beispiele:
```
BA EA88        MOV DX, 0EA88H
EE             OUT DX, AL

BA F800        MOV DX, 0F800H
EF             OUT DX, AX
```

2 Die Programmierung eines Mikroprozessors

Um einen Mikroprozessor zu programmieren, sind folgende Voraussetzungen notwendig:
- Assembler (Anwender - Software)
- Debugger (Anwender - Software)
- System - Software
- Programmieren bestimmter Hardware- Komponenten des Mikroprozessors

2.1 Der Assembler

Eine Maschinensprache besteht aus Maschinenbefehlen.
Maschinenbefehle sind durch Befehlsformate charakterisiert und bestehen aus 0 en und 1 en.
Eine Assemblersprache besteht aus symbolischen Befehlen wie z.B. ADD AX, BX
Jedem Maschinenbefehl ist ein symbolischer Befehl zugeordnet.
Für den Menschen als Benutzer ist die Maschinensprache schwer lesbar.
Deshalb bevorzugt er die bequeme Schreibweise der Assemblersprache.
Ein Assembler ist ein Übersetzungsprogramm, das ein Programm in der Assemblersprache (kurz: Assemblerprogramm) in ein Programm in der Maschinensprache (kurz: Maschinenprogramm) übersetzt.

Der Assembler besorgt zusätzlich noch einige andere Aufgaben:
(1) Man kann symbolisch adressieren und Sprünge zu symbolischen Adressen (z.B.: JMP Ziel) angeben. Wenn ein Benutzer während der Testphase Befehle entfernt oder hinzufügt, dann muß nicht das ganze Programm geändert werden, wenn ein zusätzlicher Befehl zwischen einem Sprungbefehl und der Stelle eingefügt wird, auf die der Sprung zielt, solange man symbolische Marken verwendet. Der Assembler berücksichtigt das automatisch, wenn er bei der Übersetzung die symbolischen Marken in Adressen umwandelt.

(2) Der Assembler legt die definierten Daten (z.B. Var DB 3) an von ihm bestimmten Adressen im Arbeitsspeicher ab.

(3) Der Assembler untersucht das Assemblerprogramm auf Fehler in der symbolischen Darstellung der Befehle und meldet sie. Wenn z.B. bei ADD AX, BX das D weggelassen wurde, (AD AX, BX) meldet der Assembler einen Fehler.

2.1.1 Der Turbo Assembler TASM

Ein Beispiel eines Assemblers ist der Turbo Assembler TASM.
Die symbolischen Befehle der Assemblersprache werden durch den Assembler in die Maschinensprache übersetzt. Wir benutzten den Turbo Assembler (TASM) der Firma Borland und gehen wie folgt vor:

(1) Editieren und Abspeichern des Quell-Codes in einer Datei z.B. mit dem Editor EDIT:
EDIT assbsp1.asm
(2) Dann wird diese Datei assembliert:
TASM assbsp1
Dadurch wird die Objektdatei assbsp1.obj erzeugt.
(3) Dann wird diese Datei gelinkt:
TLINK assbsp1
Dadurch wird die ausführbare Datei assbsp1.exe erzeugt
(4) Jetzt kann das Programm gestartet werden mit:
assbsp1

Bem:
(1) Alle Quell-Codes in Assemblersprache bekommen die Endung .asm
(2) Mit der Option /zi bei TASM (z.B: TASM /zi assbsp1) und /v bei TLINK (z.B: TLINK /v assbsp1) bekommt man vom TD volle Unterstützung. Ausprobieren !

2.2 Der Turbo Debugger TD

Wenn man in einem erstellten Programm Fehler suchen will, oder nur einfach genau beobachten will, was im Einzelenen (Schritt für Schritt) geschieht, benutzt man einen Debugger. Wir benutzen den *Turbo Debugger* der Firma Borland.
Wenn man ein Programm debuggen will (z.B. assbsp1.exe), dann geht man wie folgt vor:
Der Debuggger wird aufgerufen mit

```
TD assbsp1
```

Danach kommt man direkt ins CPU Fenster oder mit der Tastenkombination Alt F10 View Cpu. Im Folgenden werden die Kombination verschiedener Tasten angegeben um in verschiedene Fenster oder Menüs zu gelangen, man kann aber auch die Maus verwenden.

2.2.1 Das CPU Fenster

Die oberste Zeile des CPU Fensters zeigt den Prozessor Typ des Systems an.
Das CPU Fenster besteht aus 5 Abschnitten:
Um von einem Abschnitt zum anderen zu gelangen drückt man die Tab Taste , die Shift Tab Taste oder benutzt die Maus.

(1) Der Code Auschnitt (linke obere Ausschnitt) zeigt den Programmcode.
Ein Pfeil zeigt die aktuelle Programm Position an (CS:IP)

(2) Der Register Ausschnitt zeigt den Inhalt der CPU Register

(3) Der Status Ausschnitt (rechte Ausschnitt) zeigt den Status der Flags an.

(4) Der Daten Ausschnitt zeigt einen Ausschnitt des Arbeitsspeichers.

(5) Der Stack Ausschnitt zeigt den Inhalt des Stacks an.
Im Stack Ausschnitt zeigt ein Pfeil auf den aktuellen Stackzeiger (SS:SP)

2.2.1.1 Daten Ausschnitt

(1) Auswählen des Speicherausschnitts:
Mit Alt F10, dann Goto (kurz: Ctrl g) kann man in einem Fenster eine bestimmte Adresse angeben, ab der der Speicherausschnitt gezeigt wird.
Man hat verschiedene Eingabe Möglichkeiten, zum Beispiel:
04F23H : 04362H (direkte Angabe von Segment:Offset)
DS : 02524H (Angabe eines aktuellen Segmentregisters und Offsets)

(2) Ändern eines Werts im Speicher:
Mit Alt F10, dann Change (kurz Ctrl c) kann man an einer Speicherstelle einen Wert verändern.
Dies geht auch, wenn man an der Speicherstelle, auf der der Cursor steht, zu schreiben beginnt. Dann geht ein Fenster auf und man gibt den neuen Wert ein.

2.2.1.2 Der Code Ausschnitt

(1) Ändern der Adresse
Mit Alt F10, dann Goto (kurz Ctrl g) kann man in einem Fenster eine bestimmte Adresse angeben, zu der der Cursor sich dann hinbewegt.
Mit Alt F10, dann Previous (kurz Ctrl p) kann man dann wieder dort hin, wo man vor dem Aufruf von Goto gewesen ist.

(2) Zurück zur aktuellen Programmposition
Mit Alt F10, dann Origin (kurz Ctrl o) kommt man zur aktuellen Programmposition, wie sie in CS:IP steht (also wohin der Dreieckspfeil zeigt)
Mit Alt F10, dann Previous (kurz Ctrl p) kann man dann wieder dort hin, wo man vor dem Aufruf von Origin gewesen ist.

(3) Folge der Zieladresse eines Sprungs
Mit Alt F10, dann Follow (kurz Ctrl f) kommt man zur Zieladresse eines Sprungbefehls, wie z.B. JMP
(Geht genauso für folg. Befehle: CALL, JNZ, JO, LOOP, JCXZ, INT)
Mit Alt F10, dann Previous (kurz Ctrl p) kann man dann wieder dort hin, wo man vor dem Aufruf von Follow gewesen ist.

Bem:
Durch diese Tasten werden die Befehle des Programms NICHT abgearbeitet.
Die Benutzung dieser Tasten erlauben dem Anwender nur, sich im Code Auschnitt umzusehen.

2.2.2 Abarbeitung des Programmes:

Taste
F7 : Befehl ausführen,
F8 : Befehl ausführen, Unterprogrammaufruf wird übersprungen
F9 : Programm laufen lassen
F4 : Programm wird bis zur Cursorposition ausgeführt.
Alt-F7 : Führt Befehle eines einzelnen Interrupts durch ! (für den Spezialisten!)

Das Hilfesystem:
Taste:
Alt F10, dann Hilfe (kurz: Ctrl h): ruft allgemeines Hilfe Menü auf.
F1 : ruft lokales (kontextbezogenes) Hilfe Menü auf. D.h. je nach aktuellem Fenster wird eine andere Hilfesetllung gegeben.
F1, F1 : Gibt Index des Hifssystems an

Wenn man die Taste Alt 2 Sekunden drückt, werden in der untersten Zeile die Menüs angezeigt, die in Kombination von Alt und einer anderen Taste aufgerufen werden. Analoges gilt für die Ctrl Taste.
Der Turbo Debugger hat lokale kontextbezogene Menüs, d.h. der Inhalt des Menüs hängt davon ab, wo sich der Cursor gerade befindet und von der Textauswahl.

Wichtige Vereinbarungen:
1) Innerhalb des Turbo Debuggers muß das Segment und der Offset einer Adresse mit einer Null beginnen und mit H enden:
Beispiele: 0ABC1H : 03BDBH, 01234H : 0A234H, 01234H : 04567H

2) Innerhalb von Assemblerprogrammen muß eine hexadezimale Zahl immer mit einer Null beginnen und mit H enden:

Beispiele:
```
MOV AX, 0A1B2H
MOV DX, DS:[01232H]
```

Beginnt in einem Assemblerprogramm eine Zahl ohne Null und endet ohne H, dann wird sie vom Assembler als Dezimalzahl aufgefaßt:

Beispiele:
```
MOV AX, 1234
MOV DX, DS:[1232]
```

3) Von hier ab gelten für das Skript folgende Vereinbarungen:
- Eine Hexadezimalzahl in einem Assemblerprogrammen beginnt mit einer 0 und endet mit H.
- Eine Dezimalzahl in einem Assemblerprogrammen hat die uns bekannte übliche Schreibweise.
- Wenn nichts anderes vereinbart wurde, erkennt man eine Hexadezimalzahl außerhalb eines Assemblerprogramms (im Skript) an der Endung H.

2.3 Programm - Beispiele

Beispiel 1:

```
; PROGRAMMBESCHREIBUNG
; Das Maximum der 2 Zahlen an den 2 Adressen Z1, Z2 wird an der  Adresse Max
; gespeichert (abgelegt).

.MODEL SMALL
.STACK 100H
.DATA
; Durch folgende Anweisungen wird durch den Turbo Assembler  im Arbeitsspeicher
; an der symbolischen Adresse @DATA:Z1 der Speicherinhalt 4 und an der symbolischen
; Adresse @DATA:Z2 der Speicherinhalt 8 und an der symbolischen Adresse @DATA:Max
; der Speicherinhalt 13 gespeichert (angelegt) und zwar jeweils 2 Byte für eine Zahl.
; DW bedeutet Define Word

Z1 DW 4
Z2 DW 8
Max DW 13
;  weil der Wert 13 egal ist, ist folgendes schöner:    MAX DW ?
; .CODE bedeutet, daß hier das eigentliche Programm, das aus Befehlen
; besteht,beginnt.

.CODE
; Die folgenden 2 Befehle sind obligat für jedes  Assemblerprogramm. Dadurch wird
; das Segment, (in dem die Daten Z1, Z2, Max stehen), in DS geladen.
MOV AX, @DATA
MOV DS, AX

; Jetzt geht's los !
MOV AX, [Z1]
MOV BX, [Z2]

CMP AX, BX
JB M1
MOV [Max],AX   ; AX ist Maximum
JMP Schluss

M1:
MOV [Max],BX   ; BX ist Maximum
JMP Schluss


Schluss:
; Die 2 folgenden Befehle sind notwendig, um das Programm zu  beenden  und
; wieder ins Betriebssystem zurückzukehren.
MOV AH, 04CH
INT 021H
END
```

Bemerkungen:
1) Der Entwickler eines Programms sollte sein Programm kommentieren. Kommentare sind für das Verständnis eines Programms sehr wichtig, weil hier einzelne Befehle oder die ganze Programmidee beschrieben werden können. Zu Beginn einer jeden Kommentarzeile muß ein Semikolon stehen. Der Assembler ignoriert Kommentare.

2) Den ganzen Programmcode kann man sich im Datenabschnitt (mit Alt F10 g) des Fensters ansehen. Man könnte sogar diesen Programcode ändern (mit Alt F10 c).

Fragen:
1) Was ist der Geltungsbereich der Programmbeschreibung:
Gilt die Programmbeschreibung z.B. nur für vorzeichenlose Zahlen Z1 und Z2, oder dürfen sie vorzeichenbehaftet sein ?
Wie groß dürfen die Zahlen Z1 und Z2 sein ?

Antwort:
Z1 und Z2 dürfen nicht vorzeichenbehaftet sein.
(Wenn Z1 und Z2 verschiedene Vorzeichen haben, wird nicht das Maximum berechnet !)

2) Warum macht man nicht das Programm durch den Befehl CMP [Z1], [Z2] viel kürzer ?

Aufgaben (zum Ausprobieren):
1) Man kann im Turbo Debugger im Arbeitsspeicher für Z1 und Z2 andere Zahlen eingeben und das Programm dann nochmals mit F7 durchgehen

2) Es müssen nicht alle 2 Zahlen in Register kopiert werden. Man kann mit
MOV AX, [Z1] und CMP AX, [Z2] genauso "arbeiten".

3) Es kann statt des Befehls JB auch mit einem anderen Befehl (z.B. JA) gearbeitet werden.

4) Man kann zusätzlich noch das Minimum der beiden Zahlen an eine Speicherstelle kopieren.

5) Durch Z1 DW 4 wird im Arbeitsspeicher die Zahl 4 in 2 Bytes abgespeichert.
Durch Z1 DB 4 wird im Arbeitsspeicher die Zahl 4 in 1 Byte abgespeichert.
Was geschieht, wenn die Daten im Arbeitsspeicher in jeweils nur 1 Byte abgespeichert werden ? Also durch:
Z1 DB 4
Z2 DB 8
Max DB 6

6) Warum speichern wir die Zahlen (durch den Turbo Assembler) an den Adressen Z1, Z2, Max ?
Das kann doch auch unser Programm. Z.B. könnte 4 an die Adresse 0000:0020 kopiert werden.(durch den "Pseudobefehl" MOV [00000H:00020H],4). Konkret:

```
MOV AX, 0
MOV DS, AX
MOV BX, 020H
MOV CX, 4
MOV [BX], CX
```

Aufgaben:

1) Schreiben Sie ein Assembler-Programm, das alle ganzen Zahlen zwischen zwei vorgegebenen ganzen Zahlen z1, z2 addiert.
Das Ergebnis soll in z3 abgelegt werden.

3 Systemsoftware

3.1 Motivation

Bis jetzt konnten wir in den vorhergehenden Assemblerprogrammen noch kein Zeichen über Tastatur eingeben bzw. auf dem Bildschirm ausgeben, d.h. wir konnten noch nicht auf die Hardware zugreifen.
Will der Assemblerprogrammierer auf Hardware (z.B. Festplatte, Bildschirm usw.) zugreifen, dann gibt es dazu zwei Möglichkeiten:

3.1.1 Direkter Zugriff auf die Hardware

Alle PC´s haben eine Tastatur, eine Grafikkarte, einen Bildschirm sowie ein Diskettenlaufwerk. Oft sind auch noch Drucker, Modems, Mäuse und Festplatten installiert. Alle dieser Geräte werden über eine Reihe sehr komplizierter Zugriffe (mit den Befehlen IN, OUT) auf die Ein / Ausgabeadressen (Ports), oder spezielle Speicherstellen (oder beides zusammen) angesteuert. Wenn man z.B. einen neuen Bildschirmmodus der Grafikkarte auswählen will, braucht man über 30 OUT Befehle. Die Tastatur ist noch wesentlich schwieriger in der Programmierung.
Um z.B. einen einzelnen Tastendruck von der Tastatur zu verarbeiten, benötigt man mehrere hundert Zeilen von Befehlen in der Assemblersprache.

3.1.2 (Indirekter) Zugriff auf die Hardware mittels Systemsoftware

Wenn man die gerade beschriebenen komplizierten Ansteuerungsprogramme nicht selbst programmieren will, (um ein einigermaßen vernünftiges Assemblerprogramm zu erhalten), kann man eine Dienstleistung (Systemsoftware) des Betriebssstems MS-DOS in Anspruch nehmen.

Für den reinen Anwender ist MS-DOS das Programm, das alle Aktionen des Computers vom Einschalten des Computers bis zum Ausschalten überwacht. MS-DOS nimmt zwar einen Teil des wertvollen 640 KB Arbeitsspeichers weg, aber es bleibt einem nichts anderes übrig als MS-DOS zu verwenden. Es ist das Betriebssystem, das einem die Eingabeaufforderungen A> oder C> zur Verfügung stellt und Befehle (wie z.B. DIR) entgegennimmt und ausführt.
Dies ist jedoch nur der offensichtliche Teil des Betriebssystems MS-DOS.

Ein anderer Teil des Betriebssystems besteht aus der Systemsoftware. Diese Dienstleistung des Betriebssystems ist für den Programmierer wichtig.
Die Systemsoftware ist Software, die als Schnittstelle zwischen Anwendungsprogrammen (wie z.B. WINWORD) und der Hardware des Computers dient. Die Systemsoftware regelt in erster Linie die komplizierte Ansteuerung der einzelnen Ein / Ausgabegeräte.

3.1.3 Speicherbelegung der Systemsoftware

Die Systemsoftware besteht aus verschiedenen Programmen (besser Unterprogramme), die der Programmierer benutzen kann, um auf Hardware zuzugreifen. Sie befindet sich im sogenannten *Adaptersegment* und besteht aus dem sogenannten
- *System-BIOS* und dem
- *Zusatz-BIOS.*

Das Adaptersegment (auch **U**pper **M**emory **B**locks oder kurz UMB genannt), befindet sich im Bereich zwischen 640 KB und 1 MB und liegt damit zwischen den Adressen `A000:0000` und `F000:FFFF`.
Den Bereich zwischen `A000:0000` und `A000:FFFF` nennt man A Segment,
Den Bereich zwischen `B000:0000` und `B000:FFFF` nennt man B Segment,
Den Bereich zwischen `C000:0000` und `C000:FFFF` nennt man C Segment,
Den Bereich zwischen `D000:0000` und `D000:FFFF` nennt man D Segment,
Den Bereich zwischen `E000:0000` und `E000:FFFF` nennt man E Segment,
Den Bereich zwischen `F000:0000` und `F000:FFFF` nennt man F Segment.

3.1.3.1 Das System-BIOS

Das System-BIOS befindet sich im F Segment. Das System-BIOS besteht aus Unterprogrammen, mit denen die Diskette, die Festplatte (mit AT-Bus Schnittstelle) und die total veralteten Grafikkarten CGA und MDA angesprochen werden können.
Das System-BIOS ist physikalisch in einem ROM auf dem Motherboard (Hauptplatine) des Computers.

Zum System-BIOS gehört auch ein kleines Programm (BIOS-Startroutine), das den Anfang macht, daß das Betriebssystem in den Arbeitsspeicher geladen wird, und damit die Steuerung des Systems übernimmt.
Wie geschieht das ?

3.1.3.1.1 Das Laden des Betriebssystems

Im System-BIOS befindet sich ein kleines Programm (BIOS-Startroutine), das den Urlader, auch Bootstrap genannt, von einer bestimmten Stelle der Diskette bzw. Festplatte in den Arbeitsspeicher lädt (kopiert) und dann zum Programmbeginn dieses Urladers im Arbeitsspeicher verzweigt (und damit dieses Programm ausführt !).
Diese bestimmte Stelle der Diskette oder Festplatte heißt Bootsektor oder Bootrecord (Kopf 0, Zylinder 0, Sektor 1).
Der Bootsektor enthält die Parameter, die die Diskette hinreichend beschreiben und ein kleines Programm zum Booten, d.h. zum Laden des Betriebssystems.
Dieses Programm nennt man den Urlader oder Bootstrap.

Die BIOS-Startroutine macht nacheinander folgende Untersuchungen:

1) Test: Befindet sich beim Einschalten des Computers im Diskettenlaufwerk *eine* Diskette ?
ja:
Der Urlader testet, ob sich das Betriebssystem (d.h. die Betriebssystemdateien) auf dieser Diskette befindet.
Findet der Urlader die notwendigen Systemdateien, dann werden diese in den Hauptspeicher geladen, findet er sie nicht, wird eine entsprechende Fehler-Meldung (keine Systemdiskette) ausgegeben.
nein:
gehe zu 2)

2) Test: Besitzt der Computer *eine* Festplatte?
ja:
Der Urlader versucht (über die Partitionstabelle) die bootfähige Partition der Festplatte zu ermitteln.
Findet der Urlader diese, dann lädt er die notwendigen Systemdateien in den Hauptspeicher, findet er sie nicht, wird eine entsprechende Fehler-Meldung ausgegeben.

Bemerkungen:
Bei moderneren Computern kann im SETUP festgelegt weden, ob die Reihenfolge von 1) und 2) vertauscht werden soll.

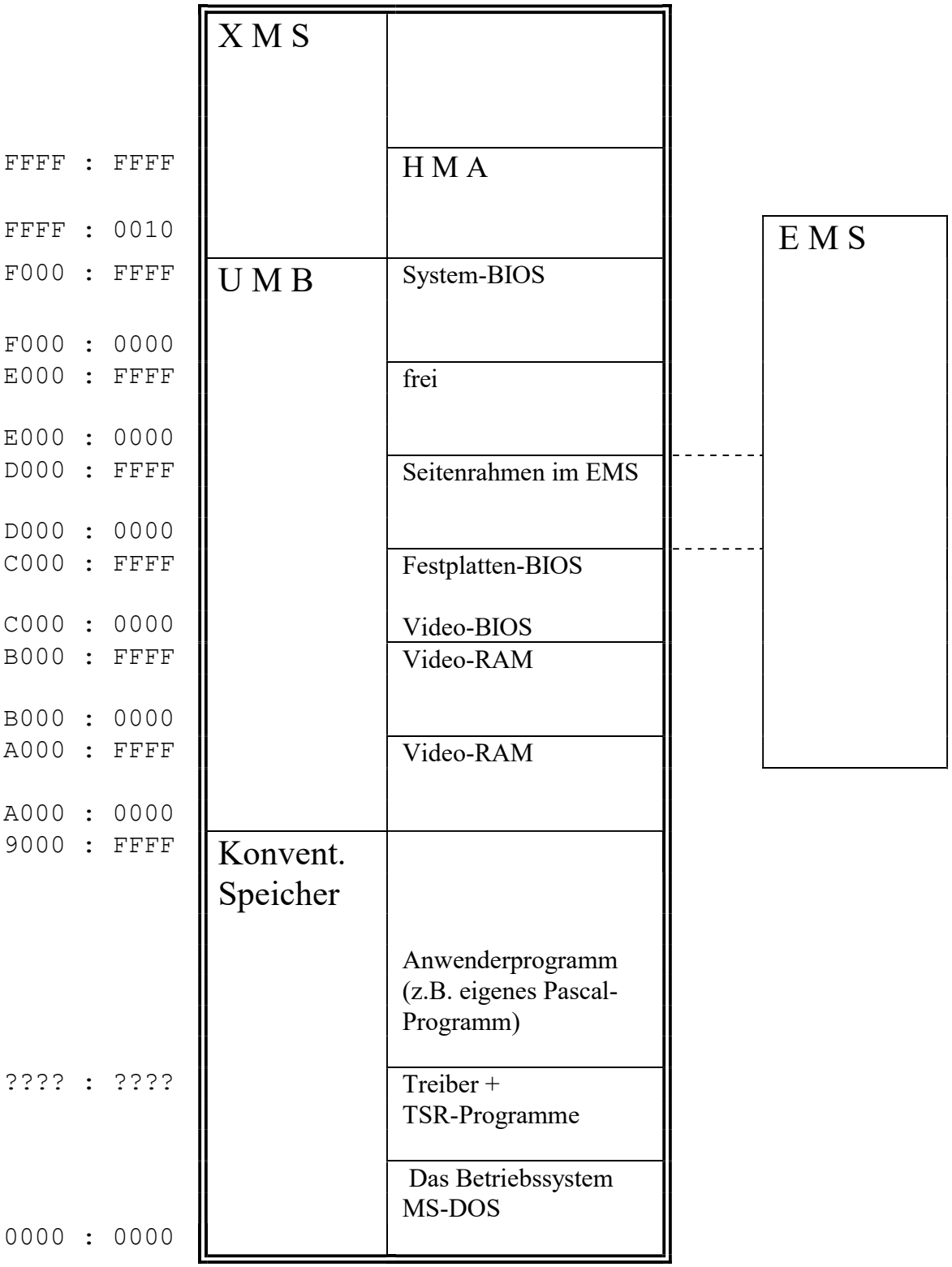
3.1.3.2 Zusatz-BIOS

Ein eventuelles Zusatz-BIOS befindet sich auf einem Teil der Segmente A bis E. Will man auf einem Computer eine andere Festplatte (z.B. eine SCSI Festplatte), eine andere Grafikkarte (z.B. eine SVGA Grafikkarte) als im System-BIOS beschrieben, oder eine spezielle Hardware (z.B. Netzwerkkarte, Soundkarte usw.) dann braucht man das Zusatz-BIOS.
Das Zusatz-BIOS besteht aus Unterprogrammen, mit denen diese Zusatz Hardware (z.B. SCSI Festplatte, SVGA Grafikkarte, Netzwerkkarte, Soundkarte usw.) angesprochen werden kann.
Das Zusatz-BIOS muß nicht alle oben beschriebenen Segmente ausfüllen. Wenn dies der Fall ist, dann hat es Löcher.

Bem:
1) Das Zusatz-BIOS ist physikalisch in einem ROM auf der entsprechenden Hardware.
2) Im Zusatz-BIOS beginnt das Unterprogramm für die Ansteuerung der Festplatte, das Festplatten BIOS üblicherweise bei `C800:0000` und das Grafikkarten BIOS (Video-BIOS) üblicherweise bei `C000:0000`.
3) Die Löcher im Adaptersegment treten in der Praxis oft auf und können mittels des Speichermanagers EMM386 genutzt werden.
Wenn man diesen Speichermanager einsetzt, gaukelt er MS-DOS vor, daß sich in den Löchern echter Speicher befindet. In diesen nun ‘neu erschlossenen’ Speicher können sogenannte speicherresidente Programme oder Treiber geladen werden und der knappe Hauptspeicher entlastet werden.
4) Mittlerweile wird statt eines Festplatten-BIOS bzw. eines Grafikkarten-BIOS häufig ein speicherresidentes Programm (Treiber) benutzt.

3.1.3.3 Speicherbelegung

Blockschaltbild des Arbeitsspeichers (Hauptspeicher)



3.1.4 Verwendung von Systemsoftware

Die verschiedenen Unterprogramme der Systemsoftware lassen sich in zwei Hauptgruppen unterteilen:

(1) MS-DOS Funktionen
(2) BIOS Funktionen

Bem:
Die MS-DOS Funktionen und BIOS Funktionen werden über INT xx (xx steht für eine Zahl zwischen 0 und 255) Befehle, sogenannte Interrupts aufgerufen, siehe später.

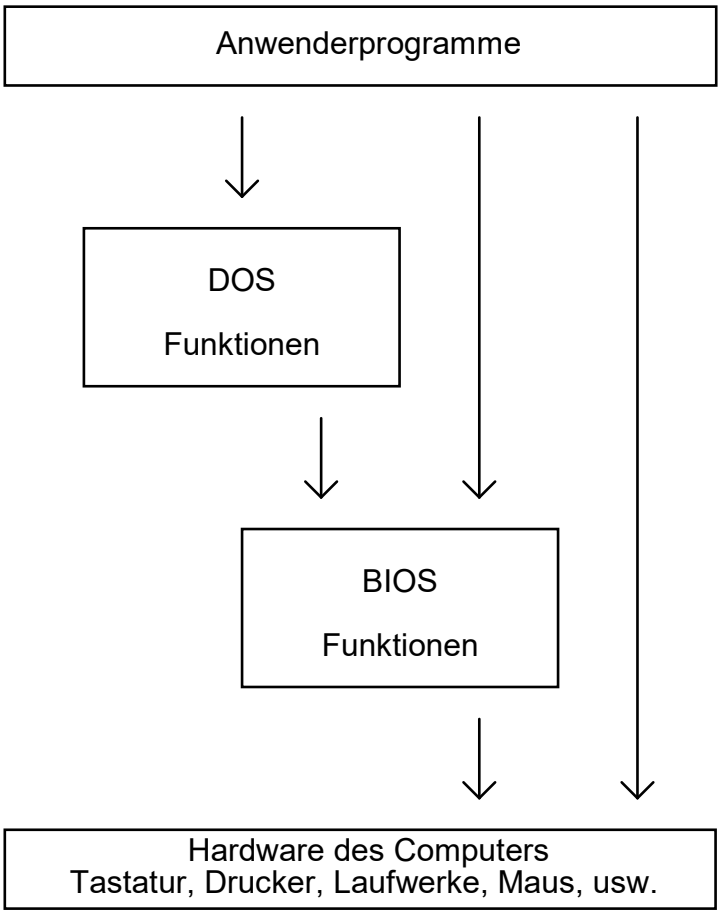
Die MS-DOS Funktionen sind im Idealfall unabhängig von der Hardware. Sie verwenden die hardwareabhängigen BIOS Funktionen, (um die Hardware anzusprechen) die der Hersteller des Computers entwickelt, einkauft oder anpaßt.
Manche Anforderungen lassen sich nicht mit den MS-DOS Funktionen erfüllen. Dann muß man die BIOS Funktionen nehmen.
Im Gegensatz zu MS-DOS und den Anwenderprogrammen, wird das BIOS nicht von der Diskette oder Festplatte geladen. Das BIOS ist im ROM untergebracht.
Das BIOS ist die der Hardware am nächsten stehende Software im PC. Es ist im allgemeinen besser, wenn man die BIOS Funktionen verwendet, als daß man die Hardware direkt programmiert, da das BIOS die Unterschiede verschiedener Computer und Geräte auszugleichen vermag.
Wenn man die Hardware dagegen selbst programmiert, muß man dieses Programm jeweils der unterschiedlichen Hardware eines Computers anpassen.

Ein Anwenderprogramm hat immer die Möglichkeit direkt auf die Hardware zuzugreifen, doch ist es einfacher, die MS-DOS und BIOS Funktionen zu verwenden.
Diese Funktionen können in allen Anwenderprogrammen intensiv genutzt werden. Mit diesen Funktionen kann man Ein / Ausgabegeräte ansteuern (Dateien lesen und schreiben, Tastatureingaben erhalten), Speicherbereiche reservieren, andere Programme aufrufen u.s.w.

zusammengefaßt:

MS-DOS Funktionen	BIOS-Funktionen
werden von Festplatte geladen	sind im ROM
hardwareunabhängig	hardwareabhängig
hardwareferner	hardwarenäher
greifen auf BIOS-Funktionen zu	spricht Hardware direkt an

3.1.4.1 Die Systemsoftware als Software-Schnittstelle



3.1.4.2 Die Benutzung von Systemsoftware mittels Assemblerbefehl

Um ein Unterprogramm der Systemsoftware zu verwenden (aufzurufen), braucht man einen speziellen Assemblerbefehl, den sogenannten *Interrupt*.

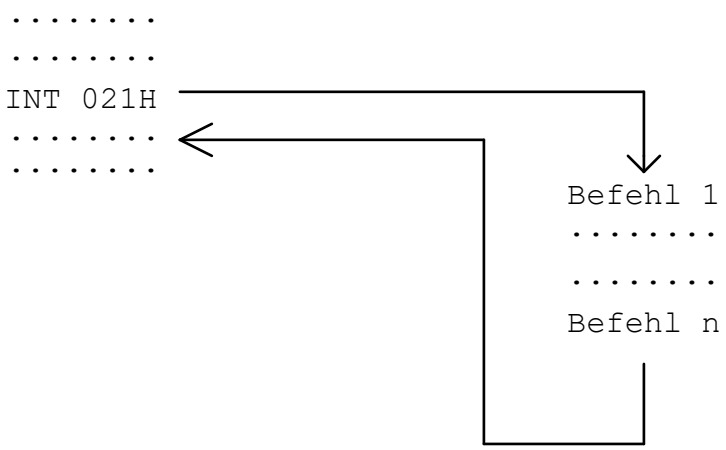
Ein Interrupt Assemblerbefehl hat die symbolische Darstellung :
`INT xx` (xx steht für eine Zahl zwischen 0 und 255)

Zum Beispiel:
`INT 021H` (21 Hex wird als 021H in einem Assemblerprogramm dargestellt)

3.1.4.2.1 Genaue Beschreibung des INT (Assembler)Befehls

Der Interrupt ähnelt einem JMP Befehl.
D.h. durch diesen Befehl wird CS:IP verändert und das Programm macht daher an einer anderen Adresse im Arbeitsspeicher weiter (wo eine Folge auszuführenden Befehle steht). Wenn diese Befehle ausgeführt wurden, veranlaßt der letzte dieser Befehle, daß CS:IP so verändert wird, daß der nächste auszuführende Befehl der INT xx folgende Befehl (in der nächsten Zeile nach INT xx) ist.

Zeichnerische Darstellung:



3.1.4.2.2 Benutzung von MS-DOS und BIOS Funktionen durch einen INT

(1) MS-DOS Funktionen
werden vom Betriebssystem MS-DOS verwendet. Am wichtigsten ist der Interrupt 21H, durch den Anwenderprogramme die MS-DOS Funktionen aufrufen können.
Die meisten der MS-DOS Funktionen werden durch den INT 21H aufgerufen.
Jede Funktion hat eine Funktionsnummer, die ins Register AH kopiert werden muß, bevor der Interrupt ausgeführt wird.

Beispiel:
Beschreibung: Programm beenden

```
MOV AH,04CH       ; Funktionsnummer 4C
INT 021H
```

(2) BIOS Funktionen
Sowohl MS-DOS als auch Anwenderprogramme verwenden die BIOS Funktionen.
Es gibt die BIOS Funktionen INT 10H bis INT 1FH.
Jede Funktion hat eine Funktionsnummer, die ins Register AH kopiert werden muß, bevor der Interrupt ausgeführt wird.
Der INT 10H ist ein wichtiger BIOS Interrupt, der zur Bildschirmsteuerung verwendet wird (z.B. Grafikmodus einstellen).

Beispiel:
Beschreibung: Prüft, ob Tastatur gedrückt wurde (Zeichen vorhanden).

```
MOV AH,01H
INT 016H
```

Wenn das ZF (Zero Flag) 1, dann ist kein Zeichen vorhanden, also wurde keine Taste gedrückt.

Bemerkungen:
1) Manchmal ist es notwendig, direkt auf die Hardware zuzugreifen (mit IN bzw. OUT)
Z.B. müssen Kommunikationsprogramme die serielle Schnittstelle des PC direkt mit IN und OUT programmieren, da weder MS-DOS noch BIOS geeignete Funktionen bieten.

2) !! Achtung !!
Der INT Befehl veranlaßt, daß an einer bestimmten Adresse im Arbeitsspeicher dort stehende Befehle (ein „Unterprogramm“) ausgeführt werden.
Diese Befehle können die Inhalte sämtlicher Register verändern.
Wenn man also einen Wert in einem Register gespeichert hat und dann einen INT aufruft, kann der Inhalt dieses Registers durch die Befehle des Unterprogramms verändert werden.

Beispiel:

```
MOV BX, 7
MOV AH, 01H
INT 021H
```

In diesem Beispiel hat jetzt (nach Abarbeitung des Assemblerbefehls INT 21H durch den Mikroprozessor) zwar das Register BX immer noch der Wert 7, doch dies muß für andere Register bei andern MS-DOS bzw. BIOS Funktionen nicht notwendigerweise so sein !!

3) Die in der nachfolgenden Beschreibung (Zusammenstellen einiger MS-DOS und BIOS Funktionen) unter dem Wort Input stehenden Register übermitteln einer MS-DOS oder BIOS-Funktion die Informationen, die diese braucht um, die angeforderte „Dienstleistung“ auszuführen.

Dazu ein Analogon:
Wenn man für eine ganze Gruppe nicht ein aufwendiges Menü kochen will, das viele Befehlen eines Kochbuchs erfordert, nimmt man die Dienstleistung einer Gaststätte in Anspruch.
Dazu reicht es aber nicht aus, dem Wirt „Ich bestelle“ zu sagen (Gast → Wirt).
Zusätzlich teilt man dem Wirt noch mit:
- ‘Menü 23’
- ‘Unsere Gruppe sitzt ab Tischnummer (Adresse) 189’
- Jetzt erst kann die Dienstleistung angefordert werden durch:
  ‘Bitte das Essen erst in einer Stunde auftischen’

Ist ein angefordertes Menü der Speisekarte einer Gaststätte nicht mehr vorhanden, so besteht die Dienstleistung der Gaststätte darin, dies dem Gast mitzuteilen (Wirt → Gast):
- ‘Menü 23 ist ausgegeangen und daher nicht mehr erhältlich’

Zusammenstellung einiger MS-DOS und BIOS Funktionen

Funktion **INT 21H, 1H**

Beschreibung Wartet auf die Eingabe eines Zeichens von der Tastatur. Mit Echo, mit Ctrl-C Test

Input AH: 1H

Output AL: eingegebenes Zeichen (im ASCII-Code)

Funktion **INT 21H, 7H**

Beschreibung Wartet auf die Eingabe eines Zeichens von der Tastatur.Kein Echo, kein Ctrl-C Test.

Input AH: 7H

Output AL: eingegebenes Zeichen (im ASCII-Code)

Funktion **INT 21H, AH**

Beschreibung Wartet auf die Eingabe einer Zeichenkette (string) von der Tastatur. Sie muß durch die RET Taste abgeschlossen werden. Vor dem Abschluß mit RET kann die Zeichenkette durch die Taste Backspace editiert werden. Sobald die Anzahl der eingegebenen Zeichen die maximale Zeichenzahl überschreitet, (Eingabepuffer läuft über), wird das zuletzt eingegebene Zeichen ignoriert und ein Ton ausgegeben. Mit Echo, mit Ctrl-C Test. Die Funktion legt die Zeichenkette an einer vom Programmierer definierten Adresse ab (Beisp: Kette DB 020H, 03H 'Hier wird Speicher reserviert`). Der Eingabepuffer muß folgenden Aufbau haben:

Byte 0: maximale Zeichenzahl, einschließlich ASCII Wert von RET, also DH (Beisp. 20H)

Byte 1: hier schreibt die Funktion die tatsächlich eingegebene Zeichenzahl ohne RET (Beisp. 3H)

Byte 2: hier beginnt die eingegebene Zeichenkette (ASCII Werte), die mit DH endet (ASCII von RET). (Im Beisp. wird der string 'Hier wird Speicher reserviert' überschrieben).

Input AH: AH

DS: Segment der Adresse des ersten Zeichens (im Eingabepuffer)

DX: Offset der Adresse des ersten Zeichens (im Eingabepuffer)

Output –

Funktion **INT 21H, BH**

Beschreibung Stellt fest, ob eine Taste gedrückt wurde.

Input AH: BH

Output AL=0: Taste wurde nicht gedrückt

AL=255: Taste wurde gedrückt.

Funktion **INT 21H, CH**

Beschreibung Wartet auf Zeichen von der Tastatur.Kein Echo, kein Ctrl-C Test. Der Tastaturpuffer wird gelöscht und anschließend die in AL angegebene Eingabefunktion durchgeführt.

Input AH: CH

AL: 1H,6H,7H,8H, oder AH

Output AL=0: Der Tastaturpuffer war leer.

Funktion **INT 21H, 2H**

Beschreibung Gibt das Zeichen auf der momentanen Cursor-Position des Bildschirms aus.

Mit Ctrl-C Test.

Input AH: 2H

DL: auszugebendes Zeichen

Output -

Funktion **INT 21H, 9H**

Beschreibung Die Zeichenkette (string) wird auf der momentanen Cursor -Position des Bildschirms ausgegben. Der string muß durch ein Dollarzeichen $ abgeschlossen sein.

Input AH: 9H

DS: Segment der Adresse des ersten Zeichens,

DX: Offset der Adresse des ersten Zeichens.

Output -

Funktion **INT 10H, 2H**

Beschreibung Der Cursor wird auf dem Bildschirm positioniert. Die Koordinaten des Cursors in der linken oberen Bildschirmecke sind (0,0).

Input AH: 2H

BH: 0H (im Normalfall)

DH: Zeilennummer (0-25)

DL: Spaltennummmer (0-80)

Output -

Funktion **INT 10H, 3H**

Beschreibung Die Position (Zeilennummer, Spaltennummer) des Cursor wird bestimmt.

Input AH: 3H

BH: 0H (im Normalfall)

Output DH: Zeilennummer

DL: Spaltennummer

CH, CL: Startzeile, Endzeile des Cursors (sagt etwas über Cursor aus, hier: unwichtig)

Funktion **INT 19H**

Beschreibung Warmstart. Das Betriebbsystem MS-DOS wird neu geladen.

Input -

Output -

Bemerkungen:
1)
Input und Output beziehen sich auf die Werte der angegebenen Register vor und nach Ausführung des jeweiligen Interrupts
2)
Mit Ctrl-C Test bedeutet: Bei Drücken dieser Zeichenkombination wird das Programm abgebrochen.
3)
Mit Echo bedeutet: Das von der Tastatur eingegebene Zeichen erscheint auf dem Bildschirm.
4)
Ab Adressse 40H:1EH befindet sich der 32 Byte große Tastatur-Puffer, in dem sich die Tastaturdaten befinden. Zu jeder gedrückten Taste, die in einem Programm verwendet wird, werden hier 2 Bytes abgelegt. Im ersten Byte befindet sich der ASCII - Wert des Zeichens und im 2. Byte (hier: unwichtig) befindet sich der zugehörige Scan-Code.

3.1.4.3 Beispiele

1. Beispiel für die Anwendung einer MS-DOS Funktion

Das Maximum zweier über Tastatur eingegebener einstelliger (!) Zahlen wird auf dem Bildschirm ausgegeben.

```
.MODEL SMALL
.STACK 100H
.DATA
; Durch folgende Anweisungen werden durch den Turbo Assembler Daten im Arbeitsspeicher
; angelegt.
Z1 DB 1
Z2 DB 2
Max DB 3


; Hier beginnt das eigentliche Programm, das aus Befehlen besteht.
.CODE
; Die folgenden 2 Befehle können durch den "Pseudobefehl "      MOV DS, @DATA
; beschrieben  werden.
; Diese Befehle bewirken, daß das (gemeinsame) Segment aller unter  .DATA aufgeführten
; Adressen  nach DS kopiert werden. Dies ist wichtig, weil das Segment von allen im
; Programm vorkommenden symbolischen Adressen der Inhalt von Register DS ist.
; Bsp:  MOV AL, [Max] kopiert den Inhalt der Adresse DS:Max ins Register AL.
; Die obigen 2 Befehle ("MOV DS, @DATA) veranlassen, daß in DS das "richtige" Segment
; steht.
; Das Programm gliedert sich logisch nach dem EVA Prinzip
MOV AX, @DATA
MOV DS, AX

                                ;***     E I NG A B E     ***
; Programm erwartet die Eingabe der ersten einstelligen Zahl:
MOV AH, 01H       ; Input AH: 1  Output AL: eingegebenes Zeichen
INT 021H
MOV [Z1], AL

; Programm erwartet die Eingabe der zweiten einstelligen Zahl:
MOV AH, 01H       ; Input AH: 1  Output AL: eingegebenes Zeichen
INT 021H
MOV [Z2], AL

; Die ASCII Werte der 2 eingegebenen einstelligen Zahlen befinden sich nun an den Adressen
; Z1 bzw. Z2 (genauer DS:Z1 bzw. DS:Z2). Nun kann mit der Verarbeitung begonnen werden:

                              ; ***    V E R A R B E I T U N G     ***
MOV AL, [Z1]     ; erste eingegebene Zahl nach Register AL kopieren
MOV BL, [Z2]     ; zweite eingegebene Zahl nach Register BL kopieren
CMP AL, BL
JB M1
MOV [Max], AL   ; AL ist Maximum
```

```
JMP Schluss
M1:
MOV [Max], BL  ; BL ist Maximum
JMP Schluss
Schluss:
```

;*** A U S G A B E ***

; bringt das Maximum auf den Bildschirm

```
MOV AH, 02H
MOV DL, [Max]
INT 021H
```

; Programm beenden

```
MOV AH, 04CH
INT 021H
END
```

Bemerkungen:
1)
Können bei diesem Programm auch mehrstellige Zahlen eingegeben werden ?

2)
Wie muß das Programm verändert werden, daß auch mehrstellige Zahlen eingegeben werden können ?

3) Schreiben Sie ein Assembler-Programm, das ein einstelliges über Tastatur eingegebenes Paßwort (ohne Echo) mit dem tatsächlichen Paßwort X vergleicht und bei einem falschen (richtigen) Paßwort die Meldung „falsches Paßwort“ („richtiges Paßwort“) ausgibt.
Das Programm kann luxeriöser gemacht werden, wenn man dem Anwender 3 Versuche der Eingabe eines Paßworts bietet.

2. Beispiel für die Anwendung einer MS-DOS Funktion

ist genau das gleiche wie das 1. Beispiel, nur daß zusätzlich noch Meldungen (aus Zeichenketten bestehend) auf dem Bildschirm ausgegeben werden.

```
.MODEL SMALL
.STACK 100H
.DATA
; Durch folgende Anweisungen werden durch den Turbo Assembler Daten im Arbeitsspeicher
; angelegt.
Z1 DB 1
Z2 DB 2
Max DB 3
; 0DH, 0AH bedeutet Carriage Return, Line Feed. Dadurch wird vor der Ausgabe der
; Zeichenkette (mit INT 21H, Fkt. 9) ein Zeilenvorschub durchgeführt. Das Dollarzeichen $
; muß  am Ende einer jeden Zeichenkette stehen, wenn sie mit INT 21H, Fkt. 9
; ausgegeben  wird.
Meld1 DB 'Eingabe der ersten einstelligen Zahl: $'
Meld2 DB 0DH, 0AH, 'Eingabe der zweiten einstelligen Zahl: $'
Meld3 DB 0DH, 0AH, 'Das Maximum ist: $'


; Hier beginnt das eigentliche Programm, das aus Befehlen besteht.
.CODE
; Die folgenden 2 Befehle können durch den "Pseudobefehl "      MOV DS, @DATA
; beschrieben  werden.
; Diese Befehle bewirken, daß das (gemeinsame) Segment aller unter  .DATA aufgeführten
; Adressen  nach DS kopiert werden. Dies ist wichtig, weil das Segment von allen im
; Programm vorkommenden symbolischen Adressen der Inhalt von Register DS ist.
; Bsp:  MOV AL, [Max] kopiert den Inhalt der Adresse DS:Max ins Register AL.
; Die obigen 2 Befehle ("MOV DS, @DATA) veranlassen, daß in DS das "richtige" Segment
; steht.
; Das Programm gliedert sich logisch nach dem EVA Prinzip
MOV AX, @DATA
MOV DS, AX

                                ;***     E I NG A B E     ***
; "Eingabe der ersten einstelligen Zahl:"
MOV DX, OFFSET Meld1
MOV AH, 09H
INT 021H

; Programm erwartet die Eingabe der ersten einstelligen Zahl:
MOV AH, 01H       ; Input AH: 1  Output AL: eingegebenes Zeichen
INT 021H
MOV [Z1], AL

; "Eingabe der zweiten einstelligen Zahl:"
MOV DX, OFFSET Meld2
MOV AH, 09H
INT 021H
```

```
; Programm erwartet die Eingabe der zweiten einstelligen Zahl:
MOV AH, 01H       ; Input AH: 1  Output AL: eingegebenes Zeichen
INT 021H
MOV [Z2], AL

; Die ASCII Werte der 2 eingegebenen einstelligen Zahlen befinden sich nun an den Adressen
; Z1 bzw. Z2 (genauer DS:Z1 bzw. DS:Z2). Nun kann mit der Verarbeitung begonnen werden:

                        ; ***    V E R A R B E I T U N G    ***
MOV AL, [Z1]    ; erste eingegebene Zahl nach Register AL kopieren
MOV BL, [Z2]    ; zweite eingegebene Zahl nach Register BL kopieren
CMP AL, BL
JB M1
MOV [Max], AL   ; AL ist Maximum
JMP Schluss
M1:
MOV [Max], BL  ; BL ist Maximum
JMP Schluss

Schluss:

                              ;***    A U S G A B E    ***
; "Das  Maximum ist:"
MOV DX, OFFSET Meld3
MOV AH, 09H
INT 021H

; bringt das Maximum auf den Bildschirm
MOV AH, 02H
MOV DL, [Max]
INT 021H

; Programm beenden
MOV AH, 04CH
INT 021H
END
```

Bemerkungen:
1)
Können bei diesem Programm auch mehrstellige Zahlen eingegeben werden ?

2)
Wie muß das Programm verändert werden, daß auch mehrstellige Zahlen eingegeben werden können ?

3) Schreiben Sie ein Assembler-Programm, das ein einstelliges über Tastatur eingegebenes Paßwort (ohne Echo) mit dem tatsächlichen Paßwort X vergleicht und bei einem falschen (richtigen) Paßwort die Meldung „falsches Paßwort“ („richtiges Paßwort“) ausgibt.
Das Programm kann luxeriöser gemacht werden, wenn man dem Anwender 3 Versuche der Eingabe eines Paßworts bietet.

3. Beispiel für die Anwendung einer MS-DOS Funktion

Es wird über Tastatur eine genau (!) 3 stellige Zeichenkette (Ziffernfolge) eingegeben (zwischen 000 und 255). Dieses Programm verwandelt diese Zeichenkette in eine 1 Byte große Zahl und kopiert sie an die Adresse Zahl.
Beispiel: (Berechnung des Zahlenwerts aus einer Ziffernfolge)
Wenn die Ziffern 2 4 6 eingegeben wurden, berechnet man die Zahl 246
auf folgende Weise:
246 = 2 * 100 + 4 * 10 + 6 * 1

```
.MODEL SMALL
.STACK 100H
.DATA
Ep DB 04H, 00H, 'gong'
Zahl DB 255
Meld1 DB 0DH,0AH, 'Eingabe einer genau 3 stelligen
Zeichenkette: $'

; Hier beginnt das eigentliche Programm, das aus Befehlen besteht.
.CODE
; Die folgenden 2 Befehle können durch den "Pseudobefehl"      MOV DS, @ DATA
; beschrieben werden.
; Diese 2 Befehl bewirken, daß das (gemeinsame) Segment aller unter  .DATA  aufgeführten
; Adressen nach DS kopiert wird. Dies ist wichtig, weil das Segment von allen im Programm
; vorkommenden  symbolischen Adressen der Inhalt von DS ist.
; Bsp: MOV  DL, [Ep]  kopiert den Inhalt der Adresse DS:Ep ins Register DL.
; Die obigen 2 Befehle ("MOV DS, @DATA) veranlassen, daß in DS das "richtige" Segment
; steht.
MOV AX, @DATA
MOV DS, AX

; "Eingabe einer genau 3 stelligen Zeichenkette"
MOV DX, OFFSET Meld1
MOV AH, 09H
INT 021H

; Eingabe einer genau 3 stelligen Zeichenkette über Tastatur
MOV AH, 0AH
MOV DX, OFFSET Ep
INT 021H

; Die eingegebene Zeichenkette besteht aus 3 Ziffern.
; Die eingegebene Zeichenkette liegt im Hauptspeicher an der Adresse:
; Ep    :       04H (maximale Länge der eingegebenen Zeichenkette)
; Ep + 1:       03H (Länge der eingegeben Zeichenkette)
; Ep + 2:       1. Ziffer
; Ep + 3:       2. Ziffer
; Ep + 4:       3. Ziffer
; Ep + 5:       0DH  (ASCII Wert der Return Taste)
```

```
; Berechne:   1. Ziffer * 100
MOV BL, [Ep + 2]      ; In BL steht der ASCII Wert der 1.Ziffer
SUB BL, 48            ; In BL steht der Zahlenwert der 1.Ziffer
MOV AL, 100
MUL BL                ; Ergebnis der Multiplikation ist in AL ( Im Bsp: 2 * 100)
                      ; (Im Bsp: AL = 2 * 100)
MOV DL, AL            ; Da wir AL noch brauchen, wird dieses Zwischenergebnis
                      ; nach DL kopiert.  (Im Bsp: DL = 200)

; Berechne:   2. Ziffer * 10
MOV BL, [Ep+3]        ; In BL steht der ASCII Wert der 2. Ziffer
SUB BL, 48            ; In BL steht der Zahlenwert der 2.Ziffer
MOV AL, 10
MUL BL                ; Ergebnis der Multiplikation ist in AL (Im Bsp: AL = 4 * 10)
ADD DL, AL            ; Der Wert der Multiplikation wird zum Zwischenergebnis
                      ; hinzuaddiert. (Im Bsp: DL = 200 + 40)

; Berechne:   3. Ziffer * 1  ( = 3. Ziffer)
MOV BL, [Ep + 4]      ; In BL steht der ASCII Wert der 3. Ziffer
SUB BL, 48            ; In BL steht  der Zahlenwert der 3.Ziffer
ADD DL, BL            ; Addiere den Zahlenwert der 3. Ziffer zum Zwischenergebnis.
                      ; (Im Bsp: DL = 200 + 40 + 6)
MOV [Zahl], DL        ; Kopiere das Endergebnis an die Adresse Zahl

; Programm beenden
Schluss:
MOV AH, 04CH
INT 021H
END
```

Bemerkungen:
1)
Was passiert, wenn man für Ep zuwenig Speicher reserviert ? D.h., wenn man
Ep DB 04H, 00H, 'gong' durch
Ep DB 04H, 00H, 'z' ersetzt ?

2)
Verändern Sie dieses Programm wie folgt:
Es wird über Tastatur eine maximal (!!) 3 stellige Zeichenkette (Ziffernfolge) eingegeben (zwischen 000 und 255). Das Programm verwandelt diese Zeichenkette in eine 1 Byte große Zahl und kopiert sie an die Adresse Ep ("Eingabepuffer").

3)
Erstellen Sie folgendes Programm:
Über Tastatur sollen zwei 3 stellige Zahlen eingegeben werden (zwischen 000 und 255). Bestimme Sie das Maximum der 2 Zahlen und zeigen Sie es auf den Bildschirm. .

4)
Erweitern Sie das Programm auf 5 stellige Zahlen (zwischen 0 und $2^{16} - 1$)