Einführung

in

Maschinensprache

Autor: Dieter Lindenberg

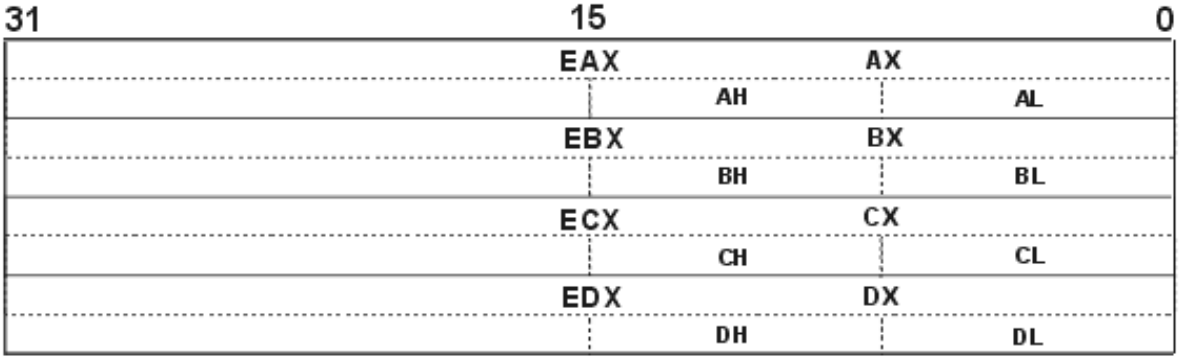
Version 2008

Inhaltsverzeichnis

Speicheradressen und Register des Pentium-Prozessors

Der Pentium-I-Prozessor besitzt 14 sog. Register, welche 32 Bit breit sind. Mit diesen Registern rechnet der Computer.

Mit den ersten vier Registern werden normalerweise mathematische Rechnungen durchgeführt.
Die ersten 16-Bit dieser vier Register unterteilen sich jeweils noch in einen High- bzw. Low-Anteil:



Alle Speicherzellen im Arbeitsspeicher des Pentium-I und auch auf den verschiedenen Laufwerken sind nur 8 Bit breit.

Die genaue Adresse einer Zelle im Arbeitsspeicher ermittelt man mit zwei Zahlen, der sog. Segment-Adresse und der sog. Offset-Adresse. Beides sind vierstellige Hexadezimalzahlen. Leider ist die Adresse der gesuchten Zelle nicht einfach eine achtstellige Hexzahl, sondern man muss diese Adresse etwas umständlich berechnen:
Die Segmentadresse wird um eine Stelle nach links verschoben (also eine Null angehängt). Danach wird die Offset-Adresse addiert. Das Ergebnis ist die Adresse der Zelle.
Beispiel: 08F1:0120 bedeutet

```
  08F10
+  0120
-------
  9030
```

Es fällt sofort auf, dass man die Adresse einer Zelle leider unterschiedlich darstellen kann. Offensichtlich ergibt auch 08E1:0220 ebenfalls 9030

Für die Angabe der Segmente gibt es vier sog. Segment-Register:
CS = Coderegister (im Codesegment befindet sich der Programmcode),
DS = Datenregister (im Datensegment befinden sich benötigte Daten),
SS = Stackregister (im Stacksegment werden z.B. Rücksprungadressen und Parameter von Prozeduren gespeichert),
ES = Extrasegmentregister,

Die vier Segmente können beliebig gewählt bzw. eingestellt werden. Insbesondere können sie sich auch überlappen. Jedes Segment ist (wegen des Offsets) 64kB groß.

Um einzelne Zellen ansprechen zu können, werden natürlich noch die Offset-Adressen benötigt.
Für das Stacksegment gibt es zwei sog. *Pointerregister: SP* = Stack Pointer und *BP* = Base Pointer
Diese beinhalten ein Offset bezüglich des *SS*

Die beiden sog. *Indexregister SI* (= Source-Indexregister) und *DI* (=Destination-Indexregister) beinhalten ein Offset bezüglich des *DS*.

Außerdem gibt es noch zwei weitere Register:
Instruction-Pointer: Dieser adressiert zusammen mit dem *CS* den nächsten Befehl.

Es existiert ein *Flagregister*, welches Flags (z.B. Overflow, Zero...) enthält, die bestimmte Zustände ausdrücken. Der R-Befehl im Hilfsprogramm DEBUG zeigt den Zustand dieser Flags an:

Flag-Name	Set	Clear
Overflow (yes/no)	OV	NV
Direction (decrement/increment)	DN	UP
Interrupt (enable/disable)	EI	DI
Sign (negativ/positiv)	NG	PL
Zero (yes/no)	ZR	NZ
Auxiliary Carry (yes/no)	AC	NA
Parity (even/odd)	PE	PO
Carry (yes/no)	CY	NC

Diese Flags enthalten Informationen über die zuletzt durchgeführten logischen und arithmetischen Operationen. So wird z.B. das *Zero Flag* gesetzt, wenn das **Ergebnis einer Rechnung** Null ist. Dieses Flag wird z.B. nicht gesetzt, wenn man nur die Zahl Null in das AX-Register schreibt. Analog ändert sich das *Parity Flag* nur aufgrund einer Rechnung.

Das *Direction-Flag* zeigt an, in welcher Richtung Strings bearbeitet werden.

Die Datenbusleitung (vom Prozessor zum Speicher oder zu den Laufwerken) des Pentium-I-Prozessors ist normalerweise 64 Bit breit. Im sog. *Real-Adress-Modus*, in welchem der alte 8086-Prozessor simuliert wird, stehen jedoch nur 16 Bit Register und ein 20 Bit Datenbus zur Verfügung.

Erstes Maschinenprogramm

Der Pentium-Prozessor ist ein sog. „von-Neumann-Rechner“. Das bedeutet, dass die Maschinenbefehle und die Daten direkt hintereinander im Speicher stehen. Natürlich sind auch die Maschinenbefehle durch Zahlen codiert. Betrachte dazu folgendes Beispiel:

```
MOV AX, 406
INC AX
MOV BL, F
INT 20
```

Dieses Miniprogramm würde folgendermaßen kodiert im Speicher stehen (beginnend an der Offsetadresse $100, wie für COM-Programme üblich):

Offsetadresse	Befehle / Daten	Kommentar
100	B8	Lade folgende Zahl nach AX
101	06	
102	04	
103	40	Erhöhe die Zahl in AX um 1
104	B3	Lade folgende Zahl nach BL
105	F	
106	CD	Programmende
107	20	

Das obige Miniprogramm enthält einen 3-Byte-Befehl, zwei 2-Byte-Befehle und einen 1-Byte-Befehl.

Das Hilfsprogramm DEBUG

Dieses Hilfsprogramm lässt sich auf zwei Arten starten. Entweder gibt man unter *Ausführen* den Befehl *debug* ein, oder man startet zuerst die MSDOS-Eingabeaufforderung (gib dazu unter *Ausführen* den Befehl *cmd* ein!) und gibt dort die Anweisung *debug*. Letztere Möglichkeit hat den Vorteil, dass man unter DOS auch das aktuelle Verzeichnis (in dem man evtl. speichern möchte) einstellen kann. DOS-Fenster lassen sich übrigens immer mit der Tastenkombination <*Alt*>+<*Return*> zwischen Teil- und Vollbildmodus umschalten. DOS Fenster werden mit *Exit* geschlossen.

Als neues Prompt erscheint in DEBUG ein Gedankenstrich.

q Quit-Befehl. Das Hilfsprogramm DEBUG wird verlassen.

d100 Display-Befehl. Zeigt den Inhalt von 128 Speicherzellen im Daten-Segment ab Zelle $100 an. Im rechten Teil des Bildschirms wird versucht, die Inhalte als ASCII-Code zu interpretieren (American **S**tandard **C**ode for **I**nformation **I**nterchange).

d9 2F zeigt 39 Speicherzellen an (von $09 bis $2F).

Der Display-Befehl zeigt normalerweise immer das Daten-Segment an.

d 01:5 zeigt im Segment 0001 ab der Zelle 0005 den Inhalt von 128 Speicherzellen an.

f 120 14F FF Fill-Befehl. Füllt den Speicherinhalt des Datensegmentes ab Zelle $120 bis Zelle 14F mit der Zahl FF.

Nach Eingabe des Befehls f 0:0 100 0 erfolgte früher immer ein schwerer Computerabsturz. Seitdem DOS allerdings nur noch von Windows simuliert wird, wird nur die DOS-Ebene unsachgemäß verlassen.

E 100 Enter-Befehl. Hiermit kann man direkt in das Datensegment Zahlen hineinschreiben (Maschinencodezahlen). Nach Eingabe von E100 erscheint der Inhalt der Zelle 100. Dieser kann direkt überschrieben werden mit einer zweistelligen (!) Hexzahl. Bei Betätigung der Leertaste erscheint der Inhalt der nächsten Zelle usw. Mit RETURN wird der Enter-Modus abgebrochen.

Mit dem Enter-Befehl können nur reine Zahlen eingegeben werden.

Gib nun (im Code-Segment !) ab Zelle 100 folgendes Programm ein:

```
B2
 1
B4
 2
CD
21
CD
20
```

g — Go-Befehl. Er startet das Programm, welches im Code-Segment in der durch das IP-Register bestimmten Speicherstelle beginnt. Nach Beendigung des Programms werden die ursprünglichen Registerinhalte wieder hergestellt.

g 11b — Breakpoints. Das Programm startet und wird genau an dem Befehl unterbrochen, der in der Zelle $11b steht. Die Register werden anschließend ausgegeben.
Vor einem nochmaligen Start muß erst der Befehlszeiger korrigiert werden.

a100 — Assemble-Befehl. Hiermit können im Daten-Segment *Mnemonics* ab Zelle $100 eingegeben werden.
Beispiel:
```
mov dl,1
mov ah,2
int 21
int 20
```

T — Der Trace-Befehl. Es wird nur ein Befehl (nicht eine ganze Funktion!) ausgeführt. Danach werden alle Registerinhalte angezeigt.

T 5 — Die nächsten 5 Programm-Instruktionen werden ausgeführt.

P — Wie der Trace-Befehl. Allerdings werden Interruptroutinen komplett ausgeführt.

Das obige Programm bringt ein sog. happy-face auf den Bildschirm. Das Programm im Speicher könnte man sich nun mit Hilfe des Display-Befehls wieder anschauen. Eine bessere Möglichkeit dazu bietet der nächste Befehl:

U 100, 106 — Unassemble-Befehl. (Der Buchstabe "d" für disassemble war schon für den Display-Befehl vergeben). Der nebenstehende Befehl unassembliert die Inhalte der Zellen 100 bis 106.

Uns interessieren zunächst nur vier 16-bit-Register. Diese sind das AX-, BX-, CX- und das DX-Register.
Zum einen kann man in diese Register ganze 16-bit-Zahlen hineinschreiben (Beispiel: Mov BX, 1FB5), zum anderen lassen sich diese vier Register auch teilweise (Lower- und higher-part) adressieren.
Beispiel: Der Befehl Mov AH,1 schreibt in die höherwertige Hälfte des AX-Registers die Zahl 1.

R Register-Befehl. Die Inhalte aller Register werden angezeigt.

RAX zeigt den Inhalt des AX-Registers an und wartet auf die Eingabe eines neuen Inhalts.

Um ein Programm speichern zu können, sind drei Schritte erforderlich: Erstens muß DEBUG mitgeteilt werden, wie das zu speichernde Programm heißen soll. Zweitens müssen wir mitteilen, wie groß das Programm ist und drittens wird der Befehl erteilt, das Programm in das aktuelle Verzeichnis zu schreiben.

N Namens-Befehl. Wir wollen unser Programm "Ascii.com" nennen.
Dazu ist folgender Befehl notwendig: nascii.com
Wichtig: keine Leertaste !

Die Länge des Programms (Anzahl der Bytes) muß im CX- und im BX-Register stehen. Insgesamt haben wir also eine Länge von 32 bits zur Verfügung. Die 16 Lower-bits stehen im CX-Register, der höherwertige Teil steht im BX-Register. Wenn ein Programm in Zelle 100 startet und in Zelle 109 endet, so ist das Programm 10 Bytes lang bzw. $A Bytes lang.
Nachdem die Länge des Programms (mit Hilfe der Registerbefehle RBX und RCX) in die entsprechenden Register hineingeschrieben worden ist, wird das Programm mit dem folgenden Befehl in das aktuelle Verzeichnis geschrieben:

W Write-Befehl. Schreibt ein Programm, dessen Name und Länge bereits festgelegt ist, in das aktuelle Verzeichnis.

DOS-Befehle: ***CLS*** löscht den Bildschirm, ***CD..*** wechselt in das nächste übergeordnete Verzeichnis, ***CD Verzeichnisnahme*** wechselt in ein untergeordnetes Verzeichnis, ***CD*** wechselt in das Stammverzeichnis.

l Load-Befehl. Der Name des zu ladenden Programms muß vorher mit dem N-Befehl festgelegt worden sein. Die Länge des Programms braucht nicht vorher festzustehen. Sie ergibt sich automatisch.

M 110 120 200 Move-Befehl. Verschiebt den Bereich von 110 bis 120 nach (Anfangsadresse) 200. Bezieht sich auf DS.

S 110 120 "Hallo" Search-Befehl. Sucht im angegebenen Bereich nach dem Text "Hallo". Bezug auf DS.

Aufgabe:
Das Datum des BIOS (= **B**asic **I**nput-**O**utput-**S**ystem), eine Art Versionsnummer, ist im ROM ab der Speicherstelle FFFF : 5 in Form von acht aufeinanderfolgenden ASCII-Zeichen abgelegt: MM/DD/YY (englische Datumsform). Ermittle dieses Datum!

Transportbefehle

Beachte im folgenden, dass *DEBUG* alle Zahlen als Hex-Zahlen interpretiert, der *Turbo-Assembler* aber alle Zahlen als Dezimalzahlen versteht. Möchte man im *Turbo-Assembler* eine Hex-Zahl eingeben, so muss dieser, falls sie mit einem Buchstaben beginnt, eine Null vorangestellt werden (weil sie sonst als Variablenname interpretiert wird).
Beispiele für Hex-Zahlen im *Turbo-Assembler*: 1A4h, 0B1h, 123h

Der allgemeine Transportbefehl lautet: **Mov Ziel, Quelle**

```
Mov AX,10
Mov AX,[10]
Mov AL,[10]
Mov [10],AX
Mov [10],AL
Mov AX,BX
```

Anstelle des AX-Registers können für obige Transportbefehle auch die folgenden Register benutzt werden:
AX, AL, AH, BX, BL, BH, CX, CL, CH, DX, DL, DH, SP, BP, SI, DI

Im Turbo-Assembler kann man auch eine Speicherstelle direkt mit einer Zahl beschreiben, also etwa `Mov [2A3Bh],10`. Dies ist in *DEBUG* nicht möglich.

Die vier Segmentregister CS, DS, SS und ES können hingegen nicht direkt mit einer Zahl geladen werden.
Möglich aber ist das Laden der Segmentregister von einem anderen Register oder von (zwei) Speicherzellen aus:

```
Mov DS,AX
Mov DS,[10]
Mov [10],DS
Mov [10],AL
```

Man kann auch nicht von einer Speicherzelle direkt in eine andere Speicherzelle transportieren. Also **`Mov [12],[13]`** geht nicht.

Überhaupt keinen Move-Befehl gibt es im Zusammenhang mit dem Befehlsregister IP.

Folgende beide Befehle sind ebenfalls nicht erlaubt:

```
Mov AL,BX
Mov  AX,BH
```

Der Move-Befehl bezieht sich (bei Speicherzellen) immer auf das DS-Segment-Register. Im *Turbo-Assembler* (**aber leider nicht in *DEBUG* !**) kann man aber auch ausdrücklich ein anderes Segment angeben: **`Mov AX,ES:[10]`**

Der DOS-Textbildschirm

Der DOS-Textbildschirm besaß früher 80 Spalten (nummeriert von 0 bis 79) und 25 Zeilen (von 0 bis 24). Heute kann man in den Eigenschaften des DOS-Fensters unter dem Register *Layout* die entsprechenden Zahlen einstellen. Damit sind 80*25=2000 Pixel (=picture element) definiert. Im Großbild-Modus gibt es mehr als 25 Zeilen. Das Pixel mit den Koordinaten (0; 0) befindet sich oben links.

Der Textbildschirmspeicher für Farbmonitore beginnt an der absoluten Adresse $B8000. Für jedes Pixel sind zwei Byte reserviert. Das erste Byte enthält den ASCII-Code des Zeichens und das zweite Byte (das sog. Attribut-Byte) bestimmt, wie dieses Zeichen gedruckt werden soll (reverse, blinkend, Farbe, usw.). Damit werden für den DOS-Textbildschirm 4000 Byte benötigt. Er endet also bei $B8F9E .

Das Zeichen oben links entspricht der Speicherzelle B800:0000. Diese enthält den ASCII-Code des Zeichens. Die nächste Zelle, also B800:0001 enthält das zugehörige Attribut-Byte. Die Bedeutung der einzelnen Bits dieses Attribut-Bytes ist für Monochromschirme anders als für Farbbildschirme. Für letztere gilt folgende Codierung (das Blinken funktioniert nur im Vollbildmodus):

Bit 7: Blinken, 0 = normal und 1 = blinkend
Bits 654: Hintergrundfarbe, genauer: 000 = schwarz
001 = blau
010 = grün
011 = hellblau
100 = rot
101 = violett
110 = orange
111 = weiß
Bit 3: Intensität, 0 = normal und 1 = intensiviert
Bits 210: Zeichenfarbe, Code identisch mit dem Code für die Hintergrundfarbe

Ein normal dargestelltes Zeichen (weiß auf dunklem Hintergrund) besitzt also das folgende Attribut: %0000 0111

Aufgaben mit DEBUG

1. Bringe das (normal dargestellte) Zeichen "A" (ASCII-Code dezimal 65) in die Textbildschirmzelle $B800:0EFE (rechts unten, bei 25 Zeilen) des Farbmonitors!
   a) Benutze nur die Pseudobefehle von DEBUG !
   b) Schreibe ein entsprechendes Assembler-Programm !
   c) Lasse dieses Zeichen hell blinken! Probiere danach verschiedene Farben für dieses Zeichen!

2. Untersuche an obiger Aufgabe die Wirkung des Befehls INT 20. Vergleiche dazu die Registerinhalte vor, während (mit dem *Trace*-Befehl) und nach dem Programmablauf! Die Routine INT 20 selbst sollte nicht mit dem Trace-Befehl durchlaufen werden.

3. Bringe das Wort "Goethe" auf den Bildschirm! Die ASCII-Codes der Kleinbuchstaben sind um dezimal 32 größer (bit 5 = 1) als die der entsprechenden Großbuchstaben

Lösung der Aufgabe 1.c) (links mit DEBUG, rechts mit dem Turboassembler):

```
MOV AX,B800
MOV DS,AX
MOV AL,41
MOV AH,8F
MOV [0EFE],AX
INT 20
```

```
MOV AX,0B800h
MOV DS,AX
MOV AL,41h
MOV AH,8Fh
MOV [0EFEh],AX
INT 20h
```

Bemerkung: Ein etwaiger Befehl MOV AX,41 wäre identisch mit dem Befehl MOV AX,0041 .
Ein Attribut-Byte Null würde das Zeichen unsichtbar machen.

Lösung der Aufgabe 3 (mit dem Turboassembler):

```
MOV AX, 0B800h  ; Beginn des Textbildschirms
MOV DS, AX

MOV AH, 7       ; Zeichenattribut
MOV AL, 71      ; Buchstabe G
MOV [0], AX
MOV AL, 111
MOV [2], AX
MOV AL, 101
MOV [4], AX
MOV AL, 116
MOV [6], AX
MOV AL, 104
MOV [8], AX
MOV AL, 101
MOV [10], AX

MOV AH, 1       ; Warten auf Tastendruck
INT 21h

INT 20h         ; Ende
```

Maschinenbefehle

INT 20h — Entspricht dem *"END."*-Befehl in Pascal.
Das Hilfsprogramm DEBUG speichert vor Programmablauf die aktuellen Registerinhalte (aber nicht die Flags!) und schreibt sie nach dem Programm wieder zurück. Zusätzlich bewirkt ***DEBUG***, dass der Instruction-Pointer wieder auf die Startadresse (100) dieses Programms zeigt.
Der *Turbo-Assembler* lässt alle Register auf dem letzten (vom Programm verursachten) Stand!

Hinweis: Dieser INT 20 – Befehl ist, wie alle Interrupt-Routinen, eine längere Funktion. Sollte man versuchen, diese Routine mit dem Trace-Befehl abzuarbeiten, so wird dies eine längere Wanderschaft durch den Speicher !

JMP 100 — Sprungbefehl nach Zelle 100. Man kann hier beliebig weit springen. **In *DEBUG* gibt es nur einen Sprung zu einer Zellennummer hin.**

JMP Marke — In *Turbo-Assembler* kann nur nach Marken hin gesprungen werden (beliebig weit).

INC AX — Inkrementiert das AX-Register um 1. Statt des AX-Registers können auch folgende Register erhöht werden:
AX, AL, AH, BX, BL, BH, CX, CL, CH, DX, DL, DH, SP, BP, SI, DI
Die vier Segment-Register CS, DS, SS, ES und alle Speicherzellen können mit INC nicht erhöht werden.

DEC AX — Dekrementiert das AX-Register. Alles analog zum INC-Befehl.

ADD AL, BL — Das Ergebnis steht jeweils im linken Register (AL := AL+BL)
ADD BX, CX — Es ist nicht möglich, ein Byte-Register und ein Wort-Register
ADD DL, 17 — zu addieren.
ADD [123], AL — Die Offset-Speichernummern beziehen sich auf das
ADD [123], AX — Code-Segment CS.
Die vier Segmentregister und das Befehlsregister dürfen im ADD-Befehl nicht benutzt werden.
Man kann zwar eine Zahl direkt zum Inhalt eines Registers, aber nicht zum Inhalt einer Speicherzelle addieren.

SUB AL,BL Alles analog zum Additionsbefehl (AL := AL - BL)

Für die **vorzeichenlose** Multiplikation wird der Befehl MUL Quelle benutzt:

MUL CL Um zwei **Bytes** miteinander zu multiplizieren, muss ein Faktor in AL stehen, der andere in einem 8-Bit-**Register**. Das Ergebnis steht dann in AX.
Leider kann man weder mit dem Inhalt einer Speicherzelle noch direkt mit einer Zahl multiplizieren.

MUL CX Um zwei **Words** miteinander zu multiplizieren, muß ein Faktor in AX stehen, der andere in einem 16-Bit-Register. Das High-Word des Ergebnisses steht in DX, das LOW-Word in AX.
Leider kann man weder mit dem Inhalt einer Speicherzelle noch direkt mit einer Zahl multiplizieren.

Für die **vorzeichenlose** Division wird der Befehl DIV Quelle benutzt:

DIV CL Um ein Wort durch ein Byte zu dividieren, muß das Wort in AX stehen, das Byte in irgendeinem 8-Bit-Register oder in einer Speicherzelle. Der Quotient steht anschließend in AL, der Rest in AH.
Es muß allerdings sicher gestellt sein, daß der entstehende Quotient höchstens 8 Bit lang ist. Das ist durchaus nicht immer der Fall: 2000 DIV 2 = 1000. In diesem Fall käme es zu folgender Fehlermeldung "Divide Overflow".

DIV CX Um ein Doppelwort durch ein Wort zu dividieren, muß das High-Wort in DX, das Low-Wort in AX stehen. Das Divisor-Wort muß in einem Register oder im Speicher (in zwei aufeianderfolgenden Zellen) stehen. Der Quotient erscheint anschließend in AX, der Rest in DX.

Die Multiplikationsbefehle und insbesondere die Divisionsbefehle sind, verglichen mit anderen Befehlen, sehr zeitaufwendig.

XCHG AX, BX
XCHG CL, DH
XCHG [12], DX
XCHG [12], BL Exchange-Befehl. Segmentregister und Befehlszeiger dürfen in diesem Befehl nicht auftauchen.

NOP No Operation. Diese Anwendung bewirkt nichts, benötigt aber etwas Zeit.

Aufgaben mit DEBUG

1. Gib in einer Endlos-Schleife *Happy-Faces* aus. Abbruch des Programms erfolgt mit <Strg>+<Pause> bzw. <Ctrl>+<Break> oder mit <Ctrl>+<C>.

2. Untersuche das *Parity-Flag*:
   a) Schreibe in das AL-Register die Zahl 1, addiere anschließend die Zahl 2.
   b) Schreibe in das AL-Register die Zahl 2, addiere anschließend die Zahl 1.
   Hinweis: Starte und durchlaufe deine Programmbefehle in *DEBUG* mit dem *Trace*-Befehl (Ausnahme: die letzte Anweisung *INT 20h* muß mit dem *P*-Befehl durchlaufen werden) und beobachte nach jedem Programmschritt die Flags! Um dies kontrollieren zu können, muss man nach jedem Programmdurchlauf *DEBUG* neu starten, weil *DEBUG* nicht die Flags zurücksetzt (nur der *Go*-Befehl setzt alle Register wieder zurück). Notiere deine Resultate!

3. Untersuche, was bei einem Überlauf bei einer Addition passiert! Interessant ist der Fall, bei dem aus einer 7-bit-Zahl eine 8-bit-Zahl wird (weil das MSB als Vorzeichen interpretiert wird), und der Fall, bei dem aus einer 8-bit-Zahl eine 9-bit-Zahl wird.
   a) Addiere im AL-Register die Zahlen 7F und 2
   b) Addiere im AL-Register die Zahlen FF und 2

4. Führe die zu 3) analoge Aufgabe mit dem 16-Bit-Register AX durch ! Addiere dazu die Zahlen
   a) 7FFF und 2 b) FFFF und 2

5. Untersuche die Addition mit Speicherzellen. Interessant ist auch hier insbesondere der mögliche Überlauf.
   a) Schreibe in die Speicherstelle [150] die Zahl 1. Addiere dazu die Zahl 3. Kontrolliere dein Ergebnis mit dem Display-Befehl d 150
   b) Schreibe in die Speicherstelle [150] die Zahl 7F. Addiere dazu die Zahl 3. Kontrolliere dein Ergebnis mit dem Display-Befehl d 150
   c) Schreibe in die Speicherstelle [150] die Zahl FF. Addiere dazu die Zahl 3. Kontrolliere dein Ergebnis mit dem Display-Befehl d 150
   d) Schreibe in die Speicherstelle [150] die Zahl 1 und in [151] die Zahl 3. Schreibe in das Register AX die Zahl $303. Addiere nun: ADD [150], AX und kontrolliere das Ergebnis mit dem Display-Befehl!
   e) Schreibe in die Speicherstelle [150] die Zahl FF und in [151] die Zahl 7F. Schreibe in das Register AX die Zahl 3. Addiere nun: ADD [150], AX und kontrolliere das Ergebnis mit dem Display-Befehl!
   f) Schreibe in die Speicherstelle [150] die Zahl FF und in [151] die Zahl FF. Schreibe in das Register AX die Zahl 3. Addiere nun: ADD [150], AX und kontrolliere das Ergebnis mit dem Display-Befehl!

6. Untersuchung der Darstellung negativer Zahlen. Beachte auch jeweils die Flags!
   Führe im AL-Register (und auch im AX-Register) die Rechnung 5-7 durch!

7. Berechne die Produkte (Dezimalzahlen !) a) 32·16 b) 1024·128

8. Wie reagiert der Rechner bei einer Division durch Null?

9. Berechne die Division $FFFF : 2
   a) als Wort : Byte b) als Doppelwort : Wort

10. Untersuche mit nachfolgenden Aufgaben, wie beim DIV-Befehl die Flags gesetzt werden!
    12 : 2 = 6 Rest 0
    1 : 2 = 0 Rest 1
    0 : 2 = 0 Rest 0

Das Hilfsprogramm Turbo-Assembler

Im Turbo-Assembler-Menue sollte man unter ***O**ptions* mit den Funktionstasten F5 bzw. F8 einstellen, ob der erzeugte Maschinencode im Arbeitsspeicher stehen soll oder als COM-File auf der Festplatte. Üblicherweise wird der Arbeitsspeicher gewählt. Außerdem sollte mit F1 *Screen on* gewählt werden. Dies bewirkt, dass man beim Assemblieren sofort vernünftige Fehlermeldungen erhält. Damit diese Fehlermeldungen bei längeren Quelltexten auch noch lesbar sind, muss noch F6 gewählt werden.

Hexadezimalzahlen, die mit einem Buchstaben beginnen, muss im Turbo-Assembler eine Null vorangestellt werden, damit man sie von Variablennamen unterscheiden kann. Beispiel: MOV AX, 0B000h

Schreibe nun im Editor folgendes Programm, welches in einer Endlosschleife Happy-Faces auf dem Bildschirm ausgibt. Schreibe mit Hilfe des Tabulators in Spalten. Die erste Spalte sollte grundsätzlich für eventuelle Markennamen reserviert sein.

```
            MOV  AH, 2      ; einzeilige Kommentare werden durch ein
            MOV  DL, 1      ; Semikolon eingeleitet.
Zeichen     INT  21h        ; Beachte die Hexadezimaldarstellung !
            JMP Zeichen
            INT  20h
```

Verlass den Editor mit *Escape*. Durch anschließendes RETURN erscheint wieder das Turbo-Assembler-Menue. Assembliere nun das Programm! Starte es anschließend !

Da sich das Programm in einer Endlosschleife befindet, lässt es sich normalerweise nur so abbrechen: Betätige nacheinander die beiden Tastenkombinationen <Strg><Pause> und <Strg><C> .

Bei normal beendeten Programmen kehrt das Programm sofort nach Beendigung zum Turbo-Assembler-Bildschirm zurück und zeigt dort die Registerbelegung an. Man beachte, dass der Befehl *INT 20h* hier nicht die Register zurücksetzt (im Gegensatz zum Hilfsprogramm DEBUG).

Aufgaben mit dem Turboassembler

1. Der Turboassembler zeigt nach dem Ende eines Programms die letzten Registerinhalte an (in Hexadezimaldarstellung). Kontrolliere damit jeweils, ob dein Programm funktioniert! Mach dir die Ergebnisse klar!

a) Schreibe unter Zuhilfenahme der Register AL und BL ein Programm zur Berechnung der Summe 5 + 6

b) Schreibe unter Zuhilfenahme der Register AH und BH ein Programm zur Berechnung der Summe 5 + 6

c) Schreibe unter Zuhilfenahme der Register AX und BX ein Programm zur Berechnung der Summe 256 + 6

d) Schreibe unter Zuhilfenahme der Register AL und BL ein Programm zur Berechnung der Summe 255 + 2

e) Schreibe unter Zuhilfenahme der Register AX und BX ein Programm zur Berechnung der Summe 255 + 2

f) Schreibe unter Zuhilfenahme der Register AL und BL ein Programm zur Berechnung der Summe 5 − 6

g) Schreibe unter Zuhilfenahme der Register AH und BH ein Programm zur Berechnung der Summe 5 − 6

h) Schreibe unter Zuhilfenahme der Register AX und BX ein Programm zur Berechnung der Summe 5 − 6

2. Berechne die Produkte (Dezimalzahlen !) a) 32·16 b) 1024·128

3. Wie reagiert der Rechner bei einer Division durch Null ?

4. Berechne die Division $FFFF : 2

a) als Wort : Byte b) als Doppelwort : Wort

Interrupts

Ein laufendes Programm wird durch sog. Interrupts unterbrochen. Es gibt einige Hardware-Interrupts wie z.B. die Betätigung der Maustaste, das Ankommen von Daten aus einem Netzwerk, der Empfang eines Tonsignals an der Soundkarte usw.
Die möglichen unterschiedlichen Interrupts haben jeweils eine eigene Nummer. Beim Eintreffen eines Interrupts werden alle Registerinhalte gesichert (auf dem sog. ***Stack***) und ein der jeweiligen Interruptnummer entsprechendes kurzes Programm wird ausgeführt. Danach werden die alten Registerinhalte wieder vom *Stack* zurückgeholt und das alte Programm wird weiter ausgeführt.

Dieselbe Interrupt-Technik nutzt man nun auch softwaremäßig, um bestimmte Funktionen auszuführen, z.B. für das Ausdrucken eines Zeichens auf dem Bildschirm.

Die ersten 1024 Byte des Rechners enthalten Zeiger auf Funktionen, die im ***ROM*** (**R**ead **O**nly **M**emory) stehen. Ein Zeiger besteht aus vier Byte, jeweils zwei für Segment- und Offsetadresse der entsprechenden *ROM*-Routine. Jeder dieser Zeiger ist mit einer entsprechenden Interruptnummer (von 0 bis 255) assoziiert.

Man kann nun leicht herausfinden wo im *ROM* eine bestimmte Interrupt-Routine beginnt. Nehmen wir an, wir suchen die Routine *INT* 16h . Unser Zeiger besetzt in der Zeropage die 4 Zellen ab 16h·4 = 22d·4 = 88d = 58h.
Diese vier Zellen enthalten die Zahlen 2E, E8, 0 und F0.
Üblicherweise werden 16-bit-Zahlen so gespeichert, dass erst das *Low-Byte*, dann das *High-Byte* gespeichert wird. Dasselbe gilt nun auch für Offset- und Segmentnummer. Demnach beginnt die gesuchte Routine an der Stelle F000:E82E. Das entspricht der absoluten Adresse FE82E .

INT 10h **Bildschirmfunktionen des ROM-BIOS**.
Die Funktionsnummer muss beim Aufruf des Interrupts in AH stehen.

2 Setzen der Cursor-Position
Die Position des Cursors kann gesetzt werden. Dazu wird die Nummer der Seite (normalerweise 0) in BH geladen, die Zeilennummer (0 bis 24 in DH) und die Spaltennummer (0 bis 79) in DL.

3 Ermitteln der Cursorposition
Die Seitennummer (normalerweise 0) wird ins BH-Register geladen. Nach Aufruf des Interrupts steht die Zeilennummer in DH, die Spaltennummer in DL. Zusätzliche, für uns unwichtige Informationen über das Aussehen des Cursors befinden sich dann in CX.

8 Zeichen und Attribut lesen
Seitennummer (0) nach BH . Nach Aufruf ist der Zeichencode des Zeichens an der Cursorposition in AL und das Attribut in AH.

9 Zeichen und Attribut schreiben
Zeichencode nach AL . Attribut nach BL . Seitennummer (0) nach BH . Die Anzahl der zu schreibenden Zeichen nach CX . Dann Aufruf.
Wichtig: Die Funktion setzt den Cursor nicht automatisch weiter.

INT 21h Die Interruptroutine 21h ruft das ***D**isk-**O**perating-**S**ystem (DOS)* auf. Welche Funktion nun *DOS* ausführt, hängt davon ab, welche Zahl sich im AH-Register befindet. Es gibt ungefähr 100 verschiedene *DOS*-Funktionen.

1 Keyboard Input Function

Befindet sich die Zahl 1 im AH-Register, so wird nach Aufruf von INT 21h solange gewartet, bis ein Zeichen von der Tastatur eingegeben wird. Dieses wird anschließend auf dem Bildschirm ausgegeben (Echo). Zusätzlich wird der *ASCII*-Code des Zeichens in das AL-Register geschrieben. (vgl. auch *DOS*-Funktion 7).

Sonderzeichen wie z.B. die Cursortasten senden zwei Zahlen in den Tastaturpuffer, von denen die erste immer 0 ist.

Beispiel:	Cursor up:	0, 72
	Cursor down:	0, 80
	Cursor left:	0, 75
	Cursor right:	0, 77

In diesen Fällen (falls die erste Zahl 0 war) muss die Funktion zweimal aufgerufen werden, damit auch die zweite Zahl aus dem Tastaturpuffer gelesen werden kann.

2 Zeichenausgabe

Befindet sich die Zahl 2 im AH-Register, so bewirkt INT 21h, dass dasjenige Zeichen auf dem Bildschirm (an der Stelle, an der sich der Cursor befindet) ausgegeben wird, dessen Zeichencode (größtenteils identisch mit dem ASCII-Code) sich im DL-Register befindet. Leider befindet sich nach der Funktionsausführung dieser Zeichencode auch im AL-Register! Außerdem wird der Cursor um eine Stelle weiter bewegt.

7 Keyboard Input Function ohne Echo

Wie DOS-Funktion 1, aber ohne Echo. Die Bedeutung der *Ctrl-Break* Kombination wird ignoriert.

8 Keyboard Input Function ohne Echo.

Wie DOS-Funktion 1, aber ohne Echo. *Ctrl-Break* wird akzeptiert.

9 Print String Function

Strings, die mit dieser Funktion auf dem Bildschirm ausgegeben werden sollen, müssen mit einem Dollarzeichen $ beendet werden. Im DX-Register muss die Offset-Adresse des Strings stehen.

Beispiel für DEBUG:

```
100:   MOV DX, 109          Beachte die Hex-Darstellung!
103:   MOV AH, 9
105:   INT 21
107:   INT 20
109:   DB 'Informatik ist gut!$'
11D:
```

Hierbei ist DB ein sog. *Pseudobefehl* (d.h. eine Anweisung für das Hilfsprogramm *DEBUG*) und bedeutet „Define Byte“. *DEBUG* wird hiermit angewiesen, **alle** zwanzig Zeichen (einschließlich Leerzeichen) zwischen den Hochkommata im ASCII-Code in den Speicher zu schreiben.
Dieser Befehl muss unbedingt am Ende des Programms stehen!
Begründung: Stände der *ASCII*-Code mitten im Programmtext, so würde der Prozessor versuchen, diesen Code als codierte Maschinenbefehle zu interpretieren.
Beachte auch, dass der nächste Befehl erst in der Zelle $11D stehen kann!

Beispiel für den TurboAssembler:

```
        MOV   DX, Offset Marke
        MOV   AH, 9
        INT   21h
        INT   20h
Marke   DB    'Guten Morgen!$'
```

Beachte, dass der *Mov-Befehl* im Zusammenhang mit Speicherzellen (hier: *Offset Marke*) sich immer auf das DS-Register bezieht !

Bemerkungen:

- 109: DB '20' bewirkt, dass anschließend in der Zelle $109 die Zahl dez 50 und in Zelle $10A die Zahl dez 48 steht.
- 109: DB 20 bewirkt, dass anschließend in der Zelle $109 die Zahl $20 (mit dem Programm DEBUG) bzw. dez 20 (mit dem Turbo-Assembler) steht
- 109: DB 1 2 A1 bewirkt, dass in den drei Zellen $109 bis $10B die drei Zahlen stehen.

A Buffered Keyboard Input Function

Im DX-Register muß die Offsetadresse des Pufferspeichers stehen.

```
100:  MOV DX, 109          Beachte die Hex-Darstellung!
103:  MOV AH, A
105:  INT 21
107:  INT 20
109:  DB 5  5..5..5..5..5..5..5   Die Fünfen dienen nur dem besseren
111:                       Verständnis des Folgenden.
```

Der Pseudobefehl DB 5 5 5 ..... sollte (in DEBUG) am Ende des Programms stehen.

Der kleine Pufferspeicher für die zu erwartende Eingabe, welcher ab Zelle $109 beginnt, wird folgendermaßen organisiert: Das erste Byte enthält die maximal mögliche Anzahl (also hier 5) des zu erwartenden Eingabe-Strings. Daraus folgt schon, dass der String höchstens 255 Zeichen (**einschließlich** des RETURNs) enthalten darf.

109	10A	10B	10C	10D	10E	10F	110
5	5	5	5	5	5	5	5

Das zweite Byte des Pufferspeichers wird erst nach der Stringeingabe die tatsächliche Zeichenanzahl enthalten. Die darauf folgenden Bytes werden den String im ASCII-Code enthalten.
Gibt man beispielsweise den String ’AB’ ein (mit Abschluß durch RETURN), so sieht der Pufferspeicher anschließend so aus:

109	10A	10B	10C	10D	10E	10F	110
5	2	dez 65	dez 66	dez 13	5	5	5

Wichtig: Die *Buffered Keyboard Input Function* speichert zwar ein *RETURN*, aber kein *LINEFEED*. Das gespeicherte Return-Zeichen wird aber bei der Stringlänge nicht mitgezählt. Der einzugebende String kann noch während der Eingabe editiert werden.

Beispiel für den TurboAssembler

```
        MOV  DX, Offset Marke
        MOV  AH, 0Ah
        INT    21h
        INT    20h
Marke   DB 8    Dup(5)
```

In diesem Beispiel kann wieder ein (einschließlich RETURN) 5-Zeichen-langer String eingelesen werden.

B Check Keyboard Status Function

Es wird kontrolliert, ob eine Taste gedrückt wurde. Die Funktion wartet nicht. Rückkehr mit
AL = FF falls Taste gedrückt,
AL = 0 falls keine Taste gedrückt

Gebraucht wird diese Funktion meistens so:

```
        MOV AH, 0Bh
Marke   INT 21h
        INC AL
        JNZ Marke
        .....
```

Achtung: Die Funktion liest selbst das Zeichen nicht. Der Tastaturpuffer wird also nicht gelöscht. Dazu sollte man extra noch einmal die Funktion 1, 7 oder 8 verwenden.

C Löschen des Tastaturpuffers

Diese Funktion löscht den Tastaturpuffer und kann dann u.a. eine der Funktionen 1, 7 oder 8 aufrufen.
Die Nummer der zusätzlich gewünschten Eingabefunktion muß in das Register AL geladen werden.
Wichtig: Falls kein nachfolgender Aufruf einer Eingabefunktion gewünscht ist, sollte man vorher die Zahl 0 ins Register AL laden !

2A Datum lesen

Nach Aufruf enthält DH den Monat (1 bis 12), DL den Tag (1 bis 31), CX das Jahr, AL den Wochentag (0 bis 6 entsprechend Sonntag bis Samstag).

2B Datum setzen

Analog zu 2A, nur umgekehrt.

2C Tageszeit lesen

Nach Aufruf enthält CH die Stunde (0 bis 23), CL die Minuten (0 bis 59), DH die Sekunden (0 bis 59) und DL die Hundertstel Sekunden (0 bis 99, aber die Genauigkeit beträgt nur 1/20 Sekunde).

2D Tageszeit setzen

Analog zu 2C, nur umgekehrt.

Die für die Zeit benötigten Timer befinden sich in den Zellen im nullten Segment: 0:046C und 0:046E

Aufgaben

1.a) Gib über die Tastatur eine einstellige Dezimalziffer ein. Der Rechner soll diese Ziffer als **Zahl** in das BX-Register schreiben. Hinweis: Die ASCII-Codes für die Ziffern 0 bis 9 liegen von dez 48 bis dez 57.

b) Gib über die Tastatur eine zweistellige Dezimalzahl ein. Der Rechner soll diese Zahl in das BX-Register schreiben.

2. Schreibe folgendes Programm: Der Rechner soll weiter nichts machen als nur auf einen Tastendruck warten. Löse diese Aufgabe mit der DOS-Funktion 8 des Interrupts 21h

3. Verwende in der Aufgabe 2 zusätzlich noch die DOS-Funktion C des INT 21h. Der Tastaturpuffer soll vorher gelöscht werden, bevor auf Tastendruck abgefragt wird.

Für die folgenden Aufgaben ist es notwendig, dass der Rechner vor Beendigung des Programms auf irgendeine Zeicheneingabe wartet (damit der letzte Bildschirm nicht sofort wieder vom Turbo-Assembler überschrieben wird). Setze also den Programmcode der Aufgabe 2 oder 3 jeweils an das Ende der folgenden Aufgaben!

4. Gib die beiden Buchstaben A und B **unter**einander aus ! (Dezimaler ASCII-Code: A = 65)
   a) nur mit INT 10h
   b) nur mit INT 21h Hier sollen die beiden Buchstaben am linken Bildschirmrand ausgegeben werden. Dafür benötigt man noch die beiden (dezimalen) ASCII-Codes: RETURN = 13, LINEFEED = 10

5. Schreibe (mit dem Move-Befehl) in das BX-Register eine einstellige Dezimalzahl. Diese Ziffer soll anschließend auf dem Bildschirm ausgegeben werden.

6. Schreibe (mit dem Move-Befehl) in das BX-Register eine zweistellige Dezimalzahl. Diese Zahl soll anschließend auf dem Bildschirm als Dezimalzahl ausgegeben werden.

Lösungen

1.a)
```
MOV AH,7
INT 21h  ; Zeicheneingabe ohne Echo
SUB AL,48
MOV BL,AL
INT 20h
```

b)
```
MOV AH,7
INT 21h  ; Zeicheneingabe ohne Echo
SUB AL,48
MOV CL,10
MUL CL   ; Dezimalstelle verschieben
MOV BL,AL
MOV AH,7
INT 21h  ; Zeicheneingabe ohne Echo
SUB AL,48
ADD BL,AL
INT 20h
```

2.
```
MOV AH,8
INT 21h
INT 20h
```

3.
```
MOV AH,0Ch
MOV AL,8
INT 21h
INT 20h
```

4.a)
```
MOV BH,0     ;Setzen der Cursorposition
MOV DH,5     ;willkürliche Zeile
MOV DL,20    ;willkürliche Spalte
MOV AH,2
INT 10h

MOV AL,65    ;Ausgabe des Zeichens A
MOV BL,7
MOV BH,0
MOV CX,1
MOV AH,9
INT 10h
```

```
MOV BH,0     ;Setzen der Cursorposition
MOV DH,6     ;Zeile erhöhen
MOV DL,20    ;dieselbe Spalte
MOV AH,2
INT 10h

MOV AL,66    ;Ausgabe des Zeichens B
MOV BL,7
MOV BH,0
MOV CX,1
MOV AH,9
INT 10h

MOV AH,8     ;Warten auf Zeicheneingabe
INT 21h
INT 20h
```

4.b) Die beiden Buchstaben werden am linken Bildschirmrand ausgegeben.

```
MOV AH,2     ;Zeichenausgabefunktion
MOV DL,10    ;Linefeed
INT 21h

MOV DL,13    ;RETURN
INT 21h

MOV DL,65    ;Buchstabe A
INT 21h

MOV DL,10    ;Linefeed
INT 21h

MOV DL,13    ;RETURN
INT 21h

MOV DL,66    ;Buchstabe B
INT 21h

MOV AH,7     ;Warten auf Zeicheneingabe
INT 21h
INT 20h
```

```
5.  MOV BX,5      ;willkürliche Ziffer

    MOV DL,BL
    ADD DL,48     ;Zeichencode der Ziffer
    MOV AH,2      ;Ausgabefunktion
    INT 21h

    MOV AH,7      ;Warten auf Eingabezeichen
    INT 21h
    INT 20h


6.  MOV BX,52     ;willkürliche Zahl

    MOV AX,BX     ;Wegen Divisionsbefehl
    MOV CL,10
    DIV CL        ;Ergebnis in AL, Rest in AH

    MOV DX,AX     ;jetzt Ergebnis in DL, Rest in DH
    ADD DL,48     ;Zeichencode des Ergebnisses
    MOV AH,2      ;Zeichenausgabe
    INT 21h

    MOV DL,DH     ;Rest nach DL
    ADD DL,48     ;Zeichencode des Ergebnisses
    MOV AH,2      ;Zeichenausgabe
    INT 21h

    MOV AH,7
    INT 21h
    INT 20h
```

Aufgaben

7. Der Rechner soll einen Zeilensprung ausgeben (ASCII-Code: dez 10) und anschließend auf einen Tastendruck warten und dies auch dem Benutzer mit den Worten *„Bitte Taste drücken!“* mitteilen (siehe DOS-Funktion 9 des INT 21h !).

8. Mit der DOS-Funktion A des INT 21h soll ein String eingegeben werden. Anschließend erfolgt die Ausgabe eines Zeilensprunges und mithilfe der DOS-Funktion 9 des INT 21h wird der anfänglich eingegebene String wieder ausgegeben.

9. Gib mithilfe der DOS-Funktion 2A des INT 21h das aktuelle Datum aus. Beispiel: „Heute ist der 13.09.“ Der ASCII-Code des Dezimalpunktes ist dez 46.

10. Gib mithilfe der DOS-Funktion 2C des INT 21h die aktuelle Uhrzeit aus. Beispiel: „Es ist jetzt 9 Uhr, 20 Minuten und 33 Sekunden.“

Lösungen

```
7.        MOV DX,Offset Marke
          MOV AH,9          ;Print String Function
          INT 21h
          MOV AH,8          ;keyboard input function
          INT 21h           ;ohne Echo
          INT 20h
   Marke  DB 'Bitte Taste drücken!$'
```

```
8.       MOV DX,Offset Marke
         MOV AH,9      ;Print String Function
         INT 21h

         MOV AH,2      ;Zeichenausgabe
         MOV DL,10     ;Linefeed
         INT 21h
         MOV DL,13     ;RETURN
         INT 21h
```

```
        MOV DX,Offset Eingabe
        MOV AH,0Ah
        INT 21h

        MOV AH,2      ;Zeichenausgabe
        MOV DL,10     ;Linefeed
        INT 21h


        MOV DX,Offset Eingabe
        ADD DX,2
        MOV AH,9      ;Print String Function
        INT 21h

        MOV AH,8      ;keyboard input function
        INT 21h       ;ohne Echo
        INT 20h

Marke   DB 'Bitte Text mit maximal 20 Zeichen
        eingeben! Als letztes Zeichen ein Dollar-
        zeichen eingeben!$'
Eingabe DB 21 DUP(20)
```

```
9.  MOV AH,2      ;Zeichenausgabe
    MOV DL,10     ;Linefeed
    INT 21h
    MOV DL,13     ;RETURN
    INT 21h

    MOV DX,Offset Marke
    MOV AH,9      ;Print String Function
    INT 21h


    MOV AH,2Ah    ;Datum lesen
    INT 21h       ;Tag in DL, Monat in DH
    MOV [200],DX ;Sicherung
    MOV AX,0
    MOV AL,DL
    MOV CL,10
    DIV CL        ;Zehnertag in AL, Einertag in AH
```

```
    MOV DX,AX    ;Zehnertag in DL, Einertag in DH

    ADD DL,48    ;Ausgabe des Zehnertages
    MOV AH,2
    INT 21h

    MOV DL,DH    ;Ausgabe des Einertages
    ADD DL,48
    MOV AH,2
    INT 21h

    MOV DL,46    ;Ausgabe des Punktes
    MOV AH,2
    INT 21h

    MOV DX,[200] ;Datum in DH
    MOV AL,DH
    MOV AH,0
    MOV CL,10
    DIV CL       ;Zehnermonat in AL, Einermonat in AH
    MOV DX,AX    ;Zehnermonat in DL, Einermonat in DH

    ADD DL,48    ;Ausgabe des Zehnermonates
    MOV AH,2
    INT 21h

    MOV DL,DH    ;Ausgabe des Einermonates
    ADD DL,48
    MOV AH,2
    INT 21h

    MOV DL,46    ;Ausgabe des Punktes
    MOV AH,2
    INT 21h

    MOV AH,2     ;Zeichenausgabe
    MOV DL,10    ;Linefeed
    INT 21h

    MOV AH,8     ;keyboard input function ohne Echo
    INT 21h
    INT 20h

Marke   DB 'Heute ist der $'
```

```
10. MOV DX,Offset Marke1 ;Print String Funktion
    MOV AH,9
    INT 21h

    MOV AH,2Ch   ;Uhrzeit lesen
    INT 21h      ;Stunde in CH, Minuten in CL
                 ;Sekunden in DH, Hundertstel in DL

    MOV [200],CX     ;Sicherung Stunden + Minuten
    MOV [202],DX     ;Sicherung Sekunden

    MOV AX,0
    MOV AL,CH        ;Stunde nach AX
    MOV BL,10
    DIV BL           ;Quotient in AL, Rest in AH
    MOV CX,AX        ;Sicherung


    MOV DL,CL        ;Ausgabe der Zehnerstunde
    ADD DL,48        ;wegen ASCII-Code
    MOV AH,2
    INT 21h

    MOV DL,CH        ;Ausgabe der Einerstunde
    ADD DL,48        ;wegen ASCII-Code
    MOV AH,2
    INT 21h

    MOV DX,Offset Marke2 ;Print String Funktion
    MOV AH,9
    INT 21h

    MOV AX,[200] ;Stunden nach AH, Minuten nach AL

    MOV AH,0         ;Nur noch Minuten in AX
    MOV BL,10
    DIV BL           ;Quotient in AL, Rest in AH
    MOV CX,AX        ;Sicherung

    MOV DL,CL        ;Ausgabe der Zehnerminute
    ADD DL,48        ;wegen ASCII-Code
    MOV AH,2
    INT 21h
```

```
    MOV DL,CH         ;Ausgabe der Einerminute
    ADD DL,48         ;wegen ASCII-Code
    MOV AH,2
    INT 21h


    MOV DX,Offset Marke3 ;Print String Funktion
    MOV AH,9
    INT 21h


    MOV AX,[202] ;Sekunden in AH, Hundertstel nach AL

    MOV AL,AH         ;Sekunden nach DL
    MOV AH,0          ;Nur noch Sekunden in AX
    MOV BL,10
    DIV BL            ;Quotient in AL, Rest in AH
    MOV CX,AX         ;Sicherung


    MOV DL,CL         ;Ausgabe der Zehnersekunde
    ADD DL,48         ;wegen ASCII-Code
    MOV AH,2
    INT 21h

    MOV DL,CH         ;Ausgabe der Einersekunde
    ADD DL,48         ;wegen ASCII-Code
    MOV AH,2
    INT 21h

    MOV DX,Offset Marke4 ;Print String Funktion
    MOV AH,9
    INT 21h

    MOV AH,8     ;Warten auf Zeicheneingabe
    INT 21h

    INT 20h

Marke1   DB 'Es ist jetzt $'
Marke2   DB ' Uhr, $'
Marke3   DB ' Minuten und $'
Marke4   DB ' Sekunden$'
```

Die Loop-Schleife

LOOP 105
LOOP Marke

Schleifenbefehl. Man bringt vor dem Aufruf des Loop-Befehls eine Zahl (= Anzahl der Schleifendurchläufe bei der REPEAT-Schleife) ins 16-bit-Register CX.
Der Loop-Befehl dekrementiert nun **zuerst** das CX-Register. Falls **danach** der Inhalt von CX noch ungleich Null ist, wird zur Zelle 105 bzw. zur Marke gesprungen. Andernfalls wird der nächste Befehl ausgeführt.

Beispiel in DEBUG:

```
100:   MOV CX, 100     Beachte die Hex-Darstellung!
103:   MOV DL, 0
105:   MOV AH, 2
107:   INT 21
109:   INC DL
10B:   LOOP 107
10D:   INT 20
```

Dasselbe Beispiel mit dem Turbo-Assembler:

```
       MOV CX, 256
       MOV DL, 0
       MOV AH, 2
Marke  INT 21h
       INC DL
       LOOP Marke
       INT 20h
```

Aufgaben

1. Gib das große Alphabet auf dem Bildschirm aus!
2. Gib sämtliche ASCII-Zeichen ab der (Dezimal-)Nummer 32 aus!
3. Gib zehnmal den Spruch „Informatik ist gut“ aus. *Carriage Return* hat den ASCII-Code 13 und *LINEFEED* den ASCII-Code 10.
4. Berechne die Potenz 2^{10} durch wiederholtes Multiplizieren. Benutze dazu den Loop-Befehl!
5. Schreibe in das AX-Register eine beliebige Zahl. Gib anschließend die einzelnen Dezimalziffern rückwärts aus!
6. Schachtel zwei LOOP-Schleifen ineinander. Die äußere soll viermal, die innere 10 mal durchlaufen werden. Es soll nur das AX-Register um 1 erhöht werden, also insgesamt 40 mal.

Lösungen

Aufgabe 1

```
          MOV CX,26
          MOV AH,2    ;Ausgabefunktion von INT 21h
          MOV DL,65   ;ASCII-Code von A
Ausgabe   INT 21h
          INC DL
          LOOP Ausgabe

          MOV AH,8    ;Warten auf Tastendruck, damit
          INT 21h     ;das Ergebnis sichtbar bleibt

          INT 20h
```

Aufgabe 2

```
          MOV CX,224  ; = 256-32
          MOV AH,2    ;Ausgabefunktion von INT 21h
          MOV DL,32   ;ASCII-Code von A
Ausgabe   INT 21h
          INC DL
          LOOP Ausgabe
          MOV AH,8    ;Warten auf Tastendruck, damit
          INT 21h     ;das Ergebnis sichtbar bleibt
          INT 20h
```

Aufgabe 3

```
          MOV CX,10
Marke     MOV DX, Offset Spruch
          MOV AH,9    ;Print String Function
          INT 21h
          MOV AH,2    ;Ausgabefunktion von INT 21h
          MOV DL,13   ;ASCII-Code von Carriage Return
          INT 21h
          MOV DL,10   ;ASCII-Code von Zeilensprung
          INT 21h
          LOOP Marke

          MOV AH,8
          INT 21h
          INT 20h
Spruch    DB 'Informatik ist gut$'
```

Aufgabe 4

```
          MOV AX,1
          MOV BX,2
          MOV CX,10

Marke     MUL BX
          LOOP Marke
          INT 20h
```

Aufgabe 5

```
          MOV DX,0          ;High-Wort
          MOV AX,12345      ;Low-Wort
          MOV BX,10
          MOV CX,5

Marke     DIV BX
          MOV [200],AX      ;Sicherung des Quotienten
          ADD DL,48         ;ASCII-Code von 0 = 48
          MOV AH,2          ;Zeichenausgabefunktion
          INT 21h
          MOV DX,0          ;High-Wort wieder 0
          MOV AX,[200]
          LOOP Marke

          MOV AH,8          ;Warten auf Tastendruck
          INT 21h
          INT 20h
```

Aufgabe 6

```
          mov ax,0
          mov cx,4          ;die äußere Schleife wird
                            ;viermal durchlaufen
aussen    mov bx,cx         ;Sicherung
          mov cx,3          ;innere Schleife wird 3 mal
innen     inc ax            ;durchlaufen
          loop innen

          mov cx,bx
          loop aussen

          int 20h
```

Befehle im Zusammenhang mit Binärzahlen

AND AL, BL
AND AX, BX
AND CX, 5

Logisches „UND“. Das Ergebnis steht im linken Register.
Beispiel:
1011 0100
AND 0010 1101
0010 0100

TEST AX, 12

Der Test-Befehl ist praktisch eine UND-Verknüpfung, welche aber nur die Flags beeinflusst. Die Register werden nicht verändert.

XOR AL, BL
XOR AX, BX
XOR DX, 5

Logisches „Exklusiv-Oder“. Das Ergebnis steht im linken Register.
Beispiel:
1011 0100
XOR 0010 1101
1001 1001

OR AL, BL
OR AX, BX
OR DL, 5

Logisches „Einschließendes-Oder“. Das Ergebnis steht im linken Register.
Beispiel:
1011 0100
OR 0010 1101
1011 1101

SHL DL,1

Shift Left. Alle Bits werden um eine Stelle nach links gerückt (= Multiplikation mit 2). Das MSB (**M**ost **S**ignificant **B**it, ganz links) erscheint im Carryflag (Übertragsflag): Das LSB (**L**east **S**ignificant **B**it) wird 0.

MOV CL, 3
SHL BX, CL

Soll mehrfach nach links gerückt werden, dann funktioniert dies nur mit Hilfe des CL-Registers.

Völlig analog funktioniert das Shiften nach rechts SHR.

SAR AX, 1

Shift arithmetic right. Im Gegensatz zu SHR behält die neue Zahl auf jeden Fall das alte Vorzeichen. Es werden also nur 7 bit geschoben.

ROL DL, 1	Rotate Left. Alle Bits werden um eine Stelle nach links gerückt. Das MSB erscheint ganz rechts **und** im Carryflag.
MOV CL, 3 ROL BX, CL	Soll mehrfach rotiert werden, dann nur so: diese mehrfache Rotation funktioniert nur mit dem CL-Register.

Völlig analog funktioniert die Rechts-Rotation ROR.

NEG DL	Der Inhalt des entsprechenden Registers wird mit -1 multipliziert. Wirkt auch auf SP,BP,SI,DI. Nicht anwendbar auf IP, Segmentregister und Speicherzellen. Bei "NEG [123] wüßte man nicht, ob ein Byte oder ein Wort negiert werden sollte.
NOT Ziel	Logisches Not. Invertierung des Bitmusters.
CBW	Konvertiert ein Byte in ein Wort. Das Byte muß in AL stehen, das Wort wird dann in AX erscheinen. Falls das Byte in AL positiv war, wird AH mit $00 gefüllt. Falls das Byte in AL negativ war, wird AH mit $FF gefüllt.

Aufgaben

1. Schreibe in das AX-Register eine beliebige Binärzahl. Anschließend soll nur das vierte Bit dieser Zahl invertiert (getoggelt) werden.
   Hinweise: Benutze die XOR-Anweisung! Das LSB ist das nullte Bit
2. Beschreibe in Worten, wie der Rechner feststellen könnte, ob das vierte Bit der Binärzahl in AX gleich 1 oder gleich 0 ist!
3. Die ASCII-Codes der Groß- und Kleinbuchstaben unterscheiden sich um dezimal 32. Beispiel: A=65 und a=97. Bei den Großbuchstaben ist das fünfte Bit Null, bei den Kleinbuchstaben Eins.
   Schreibe nun folgendes Programm: Eingegeben wird ein Großbuchstabe, ausgegeben wird der entsprechende Kleinbuchstabe. Das sollte möglichst auch anders herum funktionieren!

4. Die normale Multiplikation ist, wie bereits erwähnt, sehr zeitaufwendig. Die Multiplikation mit einer Zweierpotenz lässt sich hingegen sehr schnell durch entsprechend häufige SHL-Anweisungen durchführen. Schnelle Compiler ersetzen deshalb auch normale Multiplikationen durch Multiplikationen mit Zweierpotenzen und anschließenden Additionen.
   Beispiel: $x \cdot 10 = x \cdot (2^3 + 2) = x \cdot 2^3 + x \cdot 2$
   Schreibe eine Zahl, die kleiner als 25 ist, in das AL-Register und multipliziere sie, wie oben beschrieben, mit 10.

5. Zeitvergleich (Stand 2007): Führe obige Multiplikation mit 10 insgesamt 10000·FFFF mal durch. Einmal wie in Aufgabe 4, und einmal mit dem normalen MUL-Befehl. Welche Version dauert länger?

Lösungen

1. XOR AX, 16
INT 20h

2. AND AX, 16
Nach diesem Befehl enthält das AX-Register entweder den Wert 0 (falls vorher das vierte Bit 0 war) oder den Wert 32 (falls vorher das vierte Bit 1 war).

3.
```
MOV AH, 1     ;Zeicheneingabe mit Echo
INT 21h
XOR AL, 32    ;viertes Bit toggeln
MOV DL,AL     ;Zeichenausgabe
MOV AH,2
INT 21h
INT 20h
```

4.
```
MOV AL, 11    ;beliebige Zahl
MOV DL, AL    ;Sicherung
MOV CL, 3     ;Multiplikation mit 8
SHL AL, CL
SHL DL, 1     ;Multiplikation mit 2
ADD AL, DL
INT 20h
```

```
5.          MOV CX, 10000     ;für äußere Schleife
   aussen   MOV BX, CX        ;Sicherung
            MOV CX,0FFFFh     ;für innere Schleife

   innen    MOV AL, 11        ;beliebige Zahl
            MOV DL, AL
            SHL AL, 1         ;CL darf wegen LOOP-
            SHL AL, 1         ;Schleife nicht geändert
            SHL AL, 1         ;werden
            SHL DL, 1
            ADD AL, DL
            LOOP innen

            MOV CX, BX        ;für äußere LOOP-Schleife
            LOOP aussen

            INT 20h
```

Dieses Programm (Aufgabe 5) benötigt (im Jahre 2007) etwa 4 Sekunden. Eine Programmierung mit dem normalen Multiplikationsbefehl dauert etwa genauso lange.

Ports

Zusätzlich zu dem Arbeitsspeichersystem kann der Pentium-Prozessor auf ein zweites sog. Ein-/Ausgabe Portsystem mit 2^{16} = 64k Portadressen zugreifen. Man kann sie also mit einer 16-stelligen Binärzahl bzw. mit einer vierstelligen Hexadezimalzahl adressieren. Es gibt hier demnach keine umständliche Segment- und Offsetadressierung.

Diese sog. Ports sind Speicherzellen, deren Inhalt sowohl vom Rechner selbst als auch von Peripherie-Geräten (z.B. Drucker, Netzwerkkarte, Soundkarte usw.) gelesen bzw. geschrieben werden kann. Der Rechner kann also mithilfe dieser Ports Informationen austauschen mit Peripheriegeräten. Auch die Verbindung, d.h. der Informationsaustausch mit dem Internet findet über einen solchen Port statt (genauso wie die Kommunikation des Rechners mit dem Netzwerk-Server).

Der Adressbus von und zu den Ports ist mit den ersten 16 bit-Leitungen des Memory-Adressbusses identisch. Zusätzliche Kontroll-Leitungen (memory-read, memory-write, input-read, input-write) entscheiden, ob ein Port oder eine Speicherzelle angesprochen werden soll.

OUT 61, AL OUT 61, AX	Dieser Befehl schreibt den Inhalt des AL-Registers in den Port 61. Der Absender muss immer entweder das AL- oder das AX-Register sein.
MOV DX, 1234 OUT DX, AL	Portadresse kann entweder eine konstante Zahl (ein Byte, also höchstens 255) sein oder eine 16-Bit-Zahl, die vorher ins DX-Register geladen wurde.
IN AL, 61	analog

Programmierung des internen Lautsprechers

Jeder PC besitzt einen eigenen kleinen Lautsprecher (nicht zu verwechseln mit einer wahrscheinlich vorhandenen Soundkarte und angeschlossenen Lautsprecherboxen !). Dieser interne Lautsprecher wird von einem Chip angesteuert, welcher selbst wiederum mit mehreren Ports verbunden ist. Der Chip erzeugt eine Schwingung der Frequenz 1 193 180 Hz, welche auf den Lautsprecher gegeben werden kann. Außerdem kann man diese Frequenz durch einen Faktor n teilen, wobei n eine 16-Bit-Zahl ist. Der Zusammenhang zwischen dieser natürlichen Zahl n und der Lautsprecher-Frequenz f lautet also:
f = 1 193 180 Hz / n

Die frequenzbestimmende 16-Bit-Zahl n muß in den Port 42h geschrieben werden. Dies geschieht aber byteweise. Zuerst wird das Low-Byte von n in den Port 42h geschrieben, anschließend das High-Byte. Damit diese etwas seltsame Vorgehensweise möglich ist, muß vorher in den Port 43h eine ganz bestimmte Bit-Kombination (die Dezimalzahl 182) geschrieben werden.

Der interne Lautsprecher kann nur Signale empfangen, wenn im Port 61h die beiden niederwertigsten Bits, also das nullte und das erste, auf 1 gesetzt werden. Haben diese beiden Bits den Wert 0 , so wird damit der interne Lautsprecher wieder abgeschaltet. Die anderen Bits dieses Ports 61h dürfen auf keinen Fall geändert werden, weil sie z.B. Einfluss auf die System-Uhr haben.

```
        IN      AL, 61h     ; Einschalten des Lautsprechers mithilfe
        OR      AL, 3       ; der beiden LSB
        OUT     61h, AL     ; dez 3 = %0000 0011

        MOV     AL, 182     ; Vorbereitung auf die 2-Byte-Frequenzeingabe
        OUT     43h, AL

        MOV     AX, 2711    ; bewirkt eine Frequenz von 440 Hz
        OUT     42h, AL     ; Output LowByte
        MOV     AL, AH      ; Output High-Byte
        OUT     42h, AL

        MOV     BX, 1500    ; geschachtelte Warteschleife
Warte1  MOV     CX, 0FFFFh
Warte2  LOOP    Warte2
        DEC     BX
        JNZ     Warte1

        IN      AL, 61h     ; Ausschalten des Lautsprechers
        AND     AL, 252     ; die beiden LOW-Bits werden auf Null gesetzt.
        OUT     61h, AL     ; dez 252 = %1111 1100

        INT     20h
```

Die folgende Tabelle zeigt den Zusammenhang zwischen den Frequenzen der eingestrichenen C-Dur-Tonleiter und dem zugehörigen Faktor n:

Note	Frequenz in Hz	Faktor n
C	261,63	4560
D	293,66	4063
E	329,63	3619
F	349,23	3416
G	391,00	3043
A	440,00	2711
H	493,88	2415
C	523,25	2280

Bemerkung: Das Frequenzverhältnis zweier aufeinanderfolgender Halbtöne beträgt bekanntlich $\sqrt[12]{2} \approx 1{,}059$

Aufgaben

1. Ermittle, wie lang im letzten Programm die geschachtelte Warteschleife sein muss, damit ein Ton möglichst exakt 1 Sekunde lang klingt!
2. Ermittle mit obigem Programm deine Hörschwelle, indem du den höchsten Ton erzeugst, den du gerade noch hören kannst! Ermittle dessen Frequenz!
3. Simuliere den Klang einer Polizeisirene, indem du 20mal hintereinander zwei entsprechende Töne abwechselnd abspielst!
4. Gib nacheinander alle Töne der C-Dur-Tonleiter aus.
5. Gib zwei Töne aus, die unterschiedlich lang sind!
6. Die Anfangstöne der deutschen Nationalhymne (in C-Dur) sind: C,D,E,D,F,E,D,H,C,A,G,F,E,D,E,C,G. Programmiere dieses Lied!
7. Wenn in einem Lied zweimal derselbe Ton direkt hintereinander gespielt wird, muss der Computer zwischen den beiden Tönen eine sehr kleine Pause einschieben, da man sonst nur einen einzigen längeren Ton anstatt der beiden Einzeltöne hört. Zusätzlich ist natürlich darauf zu achten, dass alle Takte gleich lang sind!
   Die Anfangstöne des bekannten Kinderliedes „Hänschen Klein“ (in C-Dur) sind: G,E,E,F,D,D,C,D,E,F,G,G,G. Programmiere dieses Lied!

Wenn man den internen Lautsprecher sehr schnell immer wieder (mit einer **zufälligen** Wartezeit) einfach nur aus- und einschaltet, so wird ein sog. *weißes Rauschen* erzeugt:

```
            MOV   DX, 1400h  ; Initialisierung für Warteschleife

            IN    AL, 61h    ; Lautsprecher wird ausgeschaltet
            AND   AL, 252    ; dez 252 = %1111 1100

            MOV   BL, AL     ; Sicherung des Ports 61h

EIN_AUS     MOV   AL, BL     ; Lautsprecher ein- bzw. ausschalten
            XOR   AL, 3
            OUT   61h, AL

            ADD   DX, 9248h  ; Hier wird eine zufällige Wartezeit
            MOV   CL, 3      ; programmiert
            ROR   DX, CL
            MOV   CX, DX
            OR    CX, 10     ; garantiert Mindestwartezeit

Warte       LOOP  Warte
            MOV   BL, AL
            MOV   AH, 0Bh    ; Tastendruck ?
            INT   21h        ; Falls ja, dann ist AL=FF
            INC   AL         ; ansonsten AL=0
            JNZ   EIN_AUS

            INT 20h
```

Hinweis: Dieses Programm funktioniert nur richtig, wenn der Turbo-Assembler im Vollbildmodus arbeitet. Ansonsten wird das Rauschen nach einigen Sekunden durch eine Erzeugung von länger dauernden Zufallstönen ersetzt. Eine Erklärung dafür beruht auf technischen Feinheiten und kann hier leider nicht geliefert werden.

Aufgabe: Erzeuge das Geräusch eines Maschinengewehres durch "abgehacktes" weißes Rauschen.

Vergleiche und bedingte Sprünge

CMP AL, DL
CMP AX, DX
CMP AL, 12h
CMP CL, [12]

Der Vergleich geht so vonstatten, als **würde** die rechte Zahl von der linken subtrahiert. Aber es werden nur die Flags entsprechend beeinflusst!

Alle bedingten Sprünge orientieren sich an den Flags. Diese Anweisungen stehen üblicherweise direkt hinter Vergleichsbefehlen, sehr oft auch direkt hinter Rechnungen.

Gesprungen wird immer zu einer Marke bzw. zu einer Speicherzelle hin. **Diese dürfen nicht weiter als –128 oder +127 Byte entfernt sein.**

CMP AX, 5

JE Marke	Jump if Equal. Gesprungen wird, falls AX = 5
JNE Marke	Jump if Not Equal. Gesprungen wird, falls AX ≠ 5
JZ Marke	Jump if Zero. Gesprungen wird, falls AX = 5
JNZ Marke	Jump if Not Zero. Gesprungen wird, falls AX ≠ 5

Die folgenden vier Befehle gelten für vorzeichenlose Zahlen. Bei diesen wird das MSB also nicht als Vorzeichen interpretiert.

JA Marke	Jump if Above. Gesprungen wird, falls AX > 5
JAE Marke	Jump if Above or Equal. Gesprungen wird, falls AX ≥ 5
JB Marke	Jump if Below. Gesprungen wird, falls AX < 5
JBE Marke	Jump if Below or Equal. Gesprungen wird, falls AX ≤ 5

Bei den folgenden vier Befehlen wird das MSB als Vorzeichenbit interpretiert. ($\rightarrow$ 130 < 120 für Bytes)

JL Marke	Jump if Less. Gesprungen wird, falls AX < 5
JLE Marke	Jump if Less or Equal. Gesprungen wird, falls AX ≤ 5
JG Marke	Jump if Greater.
JGE Marke	Jump if Greater or Equal.

Ist das Sprungziel weiter als 128 Bytes entfernt, so nutzt man aus, dass ein **unbedingter** Sprung beliebig weit sein darf. Anstelle des Befehls JE Marke schreibt man einfach:

```
        JNE  weiter
        JMP  Marke
weiter  …….
```

Aufgaben

1. Programmiere durch Verschachtelung von Schleifen eine Delay-Zeit von etwa einer Sekunde. Benutze dazu die Anweisungen DEC, JZ oder JNZ.

2. Es soll ein Zeichen von der Tastatur eingelesen werden. Falls es ein Großbuchstabe war, soll der Rechner einen Piepton (Ausgabe des ASCII-Codes 7, funktioniert nur im Vollbildmodus !) von sich geben.

3. Es soll eine Dezimalziffer über die Tastatur eingegeben werden. Der Rechner soll einen Piepton ausgeben, falls diese Ziffer ungerade war.

4. Erzeuge einen Ton, dessen Höhe kontinuierlich steigt! (Hinweise: erniedrige den Faktor n – siehe Seite 39 - mit einer Schleife! Außerdem sollte die Länge jedes einzelnen Tones jetzt deutlich niedriger sein!)

5. Erzeuge einen Ton, dessen Höhe zuerst sinkt und dann steigt.

6. Erzeuge einen Ton, dessen Höhe mehrmals sinkt und steigt.

7. Ermittlung der Hörschwelle: Erzeuge einen Ton, dessen Höhe kontinuierlich sinkt (steigt). Auf Tastendruck wird der Lautsprecher ausgeschaltet und die Frequenz des letzten Tones wird ausgegeben.

8. Man soll mit den Zifferntasten 1 bis 8 die C-Dur-Tonleiter spielen können. Beendet wird das Programm mit der ESCAPE-Taste (ASCII-Code 27).

9. Für viele Anwendungen ist es wichtig, eine Prozedur „Taste“ zu haben. Schreibe ein Programm, welches das Wort „Taste !“ ausgibt und dann auf einen Tastendruck wartet. Beachte, dass diese Prozedur eventuell mehrmals aufgerufen werden soll! Dieses Programm ließe sich jederzeit mit dem Block-Read-Befehl weiter benutzen.

10. Eine mehrstellige natürliche Zahl wird eingegeben. Eingabeschluss durch Return. Diese Dezimalzahl befindet sich anschließend in AX. Speichere das Programm unter dem Namen READ_AX.

11. Eine mehrstellige Hex-Zahl wird eingegeben. Eingabeschluss durch Return. Ihr Wert befindet sich anschließend in AX.

12. Der Großbuchstabe A soll mit den Cursor-Tasten-gesteuert über den Bildschirm wandern. Achte auf eine mögliche Überschreitung des Bildschirmrandes!

Lösungen

Aufgabe 4

```
        IN    AL, 61h  ; Einschalten des Lautsprechers
        OR    AL, 11b  ; mithilfe der beiden LSB
        OUT   61h, AL  ; 11b = %0000 0011

        MOV   AL, 182  ; Vorbereitung auf die 2-Byte-
        OUT   43h, AL  ; Frequenzeingabe


        MOV   DX,2711
neu     MOV   AX, DX   ; bewirkt Frequenz von 440 Hz
        OUT   42h, AL  ; Output LowByte
        MOV   AL, AH   ; Output High-Byte
        OUT   42h, AL

        MOV   BX, 15   ; geschachtelte Warteschleife
Warte1  MOV   CX, 0FFFFh
Warte2  LOOP  Warte2
        DEC   BX
        JNZ   Warte1

        DEC   DX
        JNZ   neu

        IN    AL, 61h  ;Ausschalten des Lautsprechers
        AND   AL, 252  ;die beiden LOW-Bits werden auf
                       ;Null gesetzt.

        OUT   61h, AL  ;dez 252 = %1111 1100

        INT   20h
```

Aufgabe 8

```
Anfang   MOV AH, 7    ;Zeichen-Eingabefunktion
         INT 21h

         CMP AL, 49   ;Ziffer 1 ?
         JZ  MarkeC
         CMP AL, 50   ;Ziffer 2 ?
         JZ  MarkeD
         CMP AL, 51   ;Ziffer 3 ?
         JZ  MarkeE
         CMP AL, 52   ;Ziffer 4 ?
         JZ  MarkeF
         CMP AL, 53   ;Ziffer 5 ?
         JZ  MarkeG
         CMP AL, 54   ;Ziffer 6 ?
         JZ  MarkeA
         CMP AL, 55   ;Ziffer 7 ?
         JZ  MarkeH
         CMP AL, 56   ;Ziffer 8 ?
         JZ  MarkeCC
         CMP AL, 27   ;Escape-Taste ?
         JZ  ENDE

         JMP Anfang


MarkeC   MOV DX, 2280
         JMP Ton
MarkeD   MOV DX, 2032
         JMP Ton
MarkeE   MOV DX, 1810
         JMP Ton
MarkeF   MOV DX, 1708
         JMP Ton
MarkeG   MOV DX, 1522
         JMP Ton
MarkeA   MOV DX, 1356
         JMP Ton
MarkeH   MOV DX, 1208
         JMP Ton
MarkeCC  MOV DX, 1140
```

```
Ton       IN  AL, 61h ;Vorbereitung des Lautsprechers
          OR  AL, 3
          OUT 61h, AL
          MOV AL, 182
          OUT 43h, AL

          MOV AX, DX   ;Tonausgabe
          OUT 42h, AL
          MOV AL, AH
          OUT 42h, AL

          MOV BX, 1000
Warte1    MOV CX, 0FFFFh
Warte2    LOOPWarte2
          DEC BX
          JNZ Warte1

          IN  AL, 61h ;Ausschalten des Lautsprechers
          AND AL, 252
          OUT 61h, AL

          JMP Anfang


ENDE      INT 20h
```

Aufgabe12

```
          MOV BH, 0    ;Cursor in die Mitte setzen
          MOV DH, 12   ;Zeilennummer
          MOV DL, 40   ;Spaltennummer
          MOV AH, 2
          INT 10h

Ausgabe   MOV AH, 2    ;Zeichenausgabe
          MOV DL, 65   ;Buchstabe A
          INT 21h

          MOV AH, 3    ;Ermitteln der Cursorposition
          MOV BH, 0    ;Seitennummer
          INT 10h
          DEC DL       ;Cursor eine Spalte zurueck
          MOV AH, 2    ;Setzen des Cursors
          INT 10h
```

```
NEU     MOV AH, 7    ;Zeicheneingabe
        INT 21h
        CMP AL, 27   ;ESCAPE ?
        JZ  ENDE
        CMP AL, 0    ;Sonderzeichen?
        JNZ Neu
        INT 21h      ;es war ein Sonderzeichen
        CMP AL, 72   ;Cursor up
        JZ  UP
        CMP AL, 80   ;Cursor down
        JZ  DOWN
        CMP AL, 75   ;Cursor left
        JZ  LEFT
        CMP AL, 77   ;Cursor right
        JZ  RIGHT

        JMP Neu      ;es war anderes Sonderzeichen

UP      MOV AH, 3    ;Ermitteln der Cursorposition
        MOV BH, 0    ;Seitennummer
        INT 10h
        DEC DH       ;Cursor eine Zeile höher
        MOV AH, 2    ;Setzen des Cursors
        INT 10h      ;funktioniert nur völlig
        JMP Ausgabe  ;richtig im Vollbildmodus!

DOWN    MOV AH, 3    ;Ermitteln der Cursorposition
        MOV BH, 0    ;Seitennummer
        INT 10h
        INC DH       ;Cursor eine Zeile tiefer
        MOV AH, 2    ;Setzen des Cursors
        INT 10h
        JMP Ausgabe

LEFT    MOV AH, 3    ;Ermitteln der Cursorposition
        MOV BH, 0    ;Seitennummer
        INT 10h
        DEC DL       ;Cursor eine Spalte zurueck
        MOV AH, 2    ;Setzen des Cursors
        INT 10h
        JMP Ausgabe
```

```
RIGHT     MOV AH, 3    ;Ermitteln der Cursorposition
          MOV BH, 0    ;Seitennummer
          INT 10h
          MOV AH, 2    ;Setzen des Cursors
          INC DL       ;Cursor eine Spalte vor
          INT 10h
          JMP Ausgabe

ENDE      INT 20h
```

Zahlensystem-Umwandlungsprogramme

BINIHEX-Programm

Das folgende Programm wandelt eine Binärzahl, welche im BX-Register steht, in eine Hexadezimalzahl um und gibt sie aus.

```
            MOV  CH, 4      ; vier Hexadezimalziffern
Rotate      MOV  CL, 4
            ROL  BX, CL
            MOV  AL, BL
            AND  AL, 0Fh
            ADD  AL, 30h
            CMP  AL, 3Ah
            JL   Printit
            ADD  AL, 7

Printit     MOV  DL, AL
            MOV  AH, 2
            INT  21h
            DEC  CH
            JNZ  Rotate
            INT  20h
```

DeziBin-Programm

Das folgende Programm erwartet eine Dezimalzahl (< 65 535) von der Tastatur und schreibt die entsprechende Binärzahl ins BX-Register.

```
            MOV   BX, 0
Newchar     MOV   AH, 1     ; keyboard input
            INT   21h       ; function
            SUB   AL, 30h
            JL    EXIT
            CMP   9
            JG    EXIT
            CBW             ; Byte in AL wird zu Word in AX
            XCHG  AX, BX
            MOV   CX, 10
            MUL   CX
            XCHG  AX, BX
            ADD   BX, AX
            JMP   Newchar
EXIT        INT   20h
```

Aufgaben

1. Schreibe ein Programm zur Umwandlung von Dezimalzahlen in Hexadezimalzahlen. Eine Dezimalzahl soll eingegeben werden, die entsprechende Hexadezimalzahl wird ausgegeben.

Indirekte Adressierung

Das folgende Programm gibt einen String der Länge 5 auf dem Bildschirm aus.

```
        MOV   CX, 7                ; Stringlänge + Return + Linefeed
        MOV   BX, Offset Marke     ; Hier beginnt der String
Next    MOV   DL, [BX]
        MOV   AH, 2                ; Zeichenausgabe-Funktion
        INT   21h
        INC   BX
        LOOP  Next

        MOV   AH,1                 ; Warten auf Zeicheneingabe
        INT   21h

        INT   20h
Marke   DB    ‚Hallo’, 13, 10       ASCII-CODES für Return und Linefeed
```

Wichtig: Zur indirekten Adressierung können nur die vier Register BX, SI, DI und BP benutzt werden. Hierbei beziehen sich BX, SI und DI auf das Datensegment DS, während sich BP auf das Stacksegment SS bezieht.

Die Indexregister können auch teilweise kombiniert werden. Beispiele:

```
Mov AL, [BX+123]
Mov AX, [123+BX]
Mov BX, [BX+1]
Mov AX, [BX]
Mov [3+SI], AX
Mov [123+DI], AL
Mov AX, [BP+34]
MOV [BX], 17

Mov AX, [BX+SI+123]
Mov AX, [BX+DI]
Mov AX, [SI+BP]        // interessanterweise ist das möglich
```

Allerdings ist auch nicht jede beliebige Kombination erlaubt. Nicht erlaubt sind z.B. folgende Kombinationen:

```
Mov AX, [BX+BP]       // Bezug auf DS oder SS wäre nicht klar
Mov AX, [SI+DI]
MOV [BX], [SI]      // nicht klar, ob ein Byte oder ein Wort bewegt werden soll
```

Ein indirekter Adressierungsbefehl wird fast immer innerhalb einer Schleife ausgeführt.

Aufgaben

1. Das Datum des BIOS, eine Art Versionsnummer, ist im ROM ab der Speicherstelle FFFF : 5 in Form von 8 aufeinanderfolgenden ASCII-Zeichen abgelegt: MM/DD/YY (engl. Datumsform). Gib das ROM-BIOS-Datum auf dem Bildschirm aus !

2. Schreibe ein Programm „ClearScreen“, welches den Textbildschirm löscht, indem es ihn mit Leerzeichen (ASCII-Code 32, Attribut 7 = weißes Zeichen auf schwarzem Hintergrund) füllt.
Zur Erinnerung: Der Textbildschirmspeicher für Farbmonitore beginnt an der absoluten Adresse $B8000. Für jedes Pixel sind zwei Byte reserviert. Das erste Byte enthält den ASCII-Code des Zeichens und das zweite Byte das Attribut. Der Textbildschirm enthält (im Klein-Bild-Modus) 2000 Zeichen.

3. Die ersten 1024 Speicherzellen des Arbeitsspeichers werden als ***Zeropage*** bezeichnet. Sie enthält für den Rechner wichtige Informationen. Stelle die Inhalte der *Zeropage* auf dem Textbildschirm dar. (Beachte: Segmentadresse ist 0). Interpretiere diese Inhalte als ASCII-Codes und schreibe sie nacheinander direkt in den Bildschirmspeicher (ohne Verwendung der entsprechenden DOS-Funktion!). Beachte, dass jede Zelle der Zeropage gelesen werden soll, im Textbildschirmspeicher aber jede zweite Zelle das Attribut enthält. Packe das ganze Programm in eine Schleife, die solange durchlaufen wird, bis ein Tastendruck stattfindet (Funktionsnummer 0Bh des Interrupts 21h). So wird die *ZeroPage* immer wieder neu dargestellt. Betätige während des Programmablaufs die *Strg*- *Alt*- oder *AltGr*-Taste! Was fällt auf?

4. Mache dasselbe wie in Aufgabe 3 mit den Inhalten der Zellen 0000:0400 bis 0000: 0500. Beachte, dass sich in den Zellen 0000:046C bis 0:046F die Timer für die Zeitangaben befinden.

5. Eingegeben wird ein beliebiger String. Der Rechner ersetzt alle Kleinbuchstaben durch Großbuchstaben und gibt den String wieder aus.

6. Fülle mit *DEBUG* einen bestimmten Speicherbereich von zehn Byte mit der Zahl A0. Der Rechner soll anschließend diesen Speicherbereich kopieren in den direkt anschließenden 10-Byte-Speicherbereich. Kontrolliere dies mit dem Display-Befehl von *DEBUG*!

7. Der gesamte Zeichensatz soll zusammen mit der jeweiligen Codezahl ausgegeben werden.

Lösungen

Aufgabe 1

```
        MOV AX, 0FFFFh
        MOV DS, AX

        MOV CX, 8          ; BIOS-Datum enthält 8 Zeichen
        MOV BX, 5          ; Offset-Adresse
Marke   MOV AH, 2          ; Zeichenausgabefunktion
        MOV DL, [BX]
        INT 21h
        INC BX             ; nächstes Zeichen
        LOOP Marke

        MOV AH, 1          ; Warten auf Tastendruck
        INT 21h

        INT 20h
```

Aufgabe 2

```
        MOV AX, 0B800h     ; Beginn des Textbildschirms
        MOV DS, AX

        MOV CX, 2000       ; Anzahl der Bildschirmpixel
        MOV BX, 0          ; Initialisierung
        MOV AH, 7          ; Attribut
        MOV AL, 32         ; ASCII-Code für Leerzeichen

Marke   MOV [BX], AX
        INC BX
        INC BX
        LOOP Marke

        MOV AH, 1          ; Warten auf Tastendruck
        INT 21h

        INT 20h
```

Aufgabe 3

```
Start  MOV CX, 1024     ; Anzahl Zellen in Zeropage
       MOV BX, 0        ; Initialisierung
       MOV SI, 0        ; Initialisierung
       MOV DH, 7        ; Attribut

Marke  MOV AX, 0        ; Zeropage
       MOV DS, AX

       MOV DL,[BX]      ; Zeichen lesen und sichern

       MOV AX, 0B800h   ; Beginn des Textbildschirms
       MOV DS, AX

       MOV [SI], DX     ; Zeichen + Attribut schreiben

       INC BX
       INC SI
       INC SI
       LOOP Marke

       MOV AH, 0Bh      ; Tastendruck: Ja/Nein
       INT 21h
       INC AL
       JNZ Start

       INT 20h
```

Deklaration von Variablen

Zur Deklaration von Variablen gibt es die beiden Pseudobefehle DB (Define Byte) und DW (Define Word).

```
Beispiel:          MOV BL, x
                   INT 20h
              x    DB   ?
```

Der (relativ beliebige) Variablenname x muss am linken Rand des Editors stehen. Aufgrund der beiden Zeichen „DB“ behandelt der Assembler x als Variable und nicht als Marke (z.B. würde *LOOP x* nicht funktionieren).
Der Assembler reserviert für x nun eine bestimmte Speicherstelle und greift jedes Mal, wenn x gebraucht wird, auf diese Speicherzelle zu.
Das Fragezeichen bedeutet, dass bei der Deklaration der Variablen x keine Vorbelegung für x stattfindet. Der Inhalt von x ist zunächst unbestimmt.
Es gibt allerdings auch die Möglichkeit, der Variablen x schon bei der Deklaration einen Wert zuzuordnen:

```
Beispiel:          MOV BL, x
                   INT 20h
              x    DB 17
```

Eine äquivalente Möglichkeit wäre: x DB 1 Dup(17)
(vgl. weiter unten: Felder)

Analog funktioniert der Pseudobefehl DW .

Es gibt auch die Möglichkeit, ganze Felder zu deklarieren:

```
Field1   DB 10   Dup(?)          ; Hier werden 10 Speicherzellen reserviert. Die
                                   Variable Field1 ist für den Assembler aller-
                                   dings nur identisch mit der ersten dieser 10
                                   Speicherzellen. Der Befehl MOV AL, Field1
                                   würde immer nur die erste Zelle ansprechen.
                                   Die Inhalte der 10 Komponenten sind hier
                                   nicht vorbelegt.

Field2   DB 10   Dup(0ffh)       ; Alle 10 Zellen haben die Vorbelegung 255.

Field3   DB      1,3,5,2,4,6     ; Die Vorbelegung ist angegeben

Field4   DB 10   Dup(1,3,5,2,4,6) ; Das entspräche 10 Feldern vom Typ
                                     Field3
```

Es gibt auch die Möglichkeit, ganze Texte als Variable abzuspeichern:

```
Text1       DB    "Goethe-Gymnasium"

Text2       DB    "Goethe","$"
```

Hier werden die entsprechenden ASCII-Codes der Buchstaben in den Speicher geschrieben.
Die Variable Text2 zeigt hier ebenfalls nur auf diejenige Speicherstelle, die den Zeichencode des Buchstaben G enthält.

Sehr wichtig: Der Turbo-Assembler reserviert die Speicherzellen für die Variablen leider nicht automatisch am Ende des Programmcodes, sondern an genau der Stelle, wo gerade der entsprechende Deklarationsbefehl im Assembler-Quelltext auftaucht.

Deshalb darf ein Deklarationsbefehl niemals am Anfang eines Programmes stehen. Der Rechner würde den Inhalt der ersten Zelle als Befehl (und nicht als Variablenspeicher) ansehen und entsprechend handeln.

Man sollte Deklarationsteile auch nicht irgendwo mitten im Assemblertext unterbringen. Das hätte zur Folge, dass man mit dem Hilfsprogramm DEBUG nicht mehr den Assembler-Quelltext lesen könnte.

Also: Variablendeklarationen immer an das Ende des Assemblertextes

Wenn mit X eine Variable oder ein ganzes Variablenfeld deklariert ist, gibt der Pseudo-Befehl **Offset X** die Offset-Speicheradresse von X an.
Beispiel:

```
Mov  DX, Offset X
....
X    DB 12    Dup(?)
```

Wichtig: Der Offset-Befehl und alle Variablen beziehen sich immer auf das DS-Register. Vorsicht, wenn DS im Programm geändert wird!

Dem Offset-Befehl **ähnlich** ist der Befehl LEA (=Load Effective Adress)
Beispiel: LEA AX, X <=> Mov AX, Offset X

Ist X ein Feld (von Bytes), so kann man mit folgenden Befehlen auf das **dritte** Element zugreifen:

```
LEA  DX, X[2]  <=>  Mov  DX, Offset X + 2
```

Aufgaben

1. Definiere ein Feld, welches nur 2 Zahlen (Bytes) enthält. Schreibe anschließend ein Programm, welches diese beiden Zahlen im Feld vertauscht. Um das Ergebnis kontrollieren zu können, lade anschließend dies beiden Zahlen in die Register AL und CL

2. Definiere ein Feld mit 10 Ziffern. Der Rechner soll
   a) die größte dieser Ziffern ins AX-Register schreiben.
   b) alle Ziffern auf dem Bildschirm untereinander ausgeben.
   c) das Feld sortieren und auf dem Bildschirm ausgeben.
   d) die Teilaufgaben a, b und c mit mehrstelligen Zahlen lösen.

Definition von Konstanten

Es gibt zwei Pseudobefehle, um Konstanten zu definieren.

```
Beispiel:    Name1    EQU    9
             Name2    =      8
```

Die Konstantennamen müssen im Editor des Assemblers am linken Rand stehen. Mit EQU kann man einer Konstanten nur ein einziges mal einen Wert zuweisen (wie normalerweise auch üblich). Mit dem Gleichheitszeichen kann man ein und demselben Namen mehrmals einen unterschiedlichen konstanten Wert innerhalb des Quelltextes zuweisen. Das letztere sollte man normalerweise allerdings nicht tun.

Für Konstanten werden keine Speicherzellen reserviert. Der Assembler ersetzt beim assemblieren sofort den Konstantennamen durch den entsprechenden Wert. Deshalb gibt es im folgenden Beispiel auch keine Konflikte:

```
k     EQU   9
      Mov   AL, k
      Mov   BX, k
```

Es gibt nicht die Möglichkeit, einen ganzen Text als Konstante zu definieren. Allerdings kann man den Zeichencode eines Zeichens als Konstante definieren:

```
ch    equ   'A'
```

Damit hätte ch den Wert 65 .

Der Stack (LIFO-Keller)

Lässt man sich alle Register ausgeben, so sieht man in der ersten Zeile auch den Stackpointer SP. Dieser zeigt auf die **zuletzt benutzte** Zelle des Stacks. Der Stack ist ein Speicherbereich, in dem normalerweise alle Rücksprungadressen (bei Prozedur- oder Funktionsaufrufen) und alle Übergabeparameter an Prozeduren bzw. Funktionen gespeichert werden.

Es können grundsätzlich nur Worte (kein einzelnes Byte) auf den Stack geschrieben werden (obwohl der Speicher natürlich nur aus Byte-Zellen besteht).

Der Stack beginnt bei der letzten Offset-Adresse des Segmentes und baut sich abwärts auf.

Wenn ein COM-File (< 64k) geladen wird, wird der Stack-Pointer automatisch auf das Ende des Segmentes gesetzt. Das geschriebene COM-Programm beginnt grundsätzlich bei der Offset-Nummer $100 und wächst zu höheren Adressen hin. Ein EXE-File beginnt bei der Offset-Adresse $0 .

PUSH AX	**Kopiert** den Inhalt von AX auf den Stack und der Stack-Pointer wird um 2 dekrementiert
PUSH [1234]	Kopiert die beiden Zellen 1234 und 1235 auf den Stack und der Stackpointer wird um 2 dekrementiert
PUSH zahl	Beide Anweisungen kopieren den Inhalt der Variablen *zahl*
PUSH [zahl]	auf den Stack und der Stackpointer wird um 2 dekrementiert.

Wichtig: Man kann nicht direkt eine Zahl auf den Stack legen. Die Assembler-anweisung PUSH 185 führt beim Turbo-Assembler leider nicht zu einer Fehlermeldung, sie wird jedoch völlig unsinnig übersetzt.

POP AX	Der umgekehrte Befehl von PUSH
POP [1234]	
POP zahl	
POP [zahl]	

Push- und Pop-Operationen sind sehr schnell.

Außer dem SP-Register gibt es noch das BP-(Base-Pointer-) Register. Dieses kann auf eine beliebige Zelle des Stacks zeigen.

Aufgaben

1. Schreibe beliebige Zahlen in die vier Register AX, BX, CX und DX. Speichere diese Inhalte auf den Stack und hole sie anschließend wieder zurück. Kontrolliere das Ganze mit dem Trace-Befehl von DEBUG!

Prozeduren

Es ist wichtig, dass im Quelltext **zuerst** das Hauptprogramm steht und anschließend irgendwelche Prozeduren. Begründung: der Turbo-Assembler übersetzt jeden Befehl sofort in den Maschinen-Code. Wenn ein Programm mit einer Prozedur beginnt, so wird es abstürzen, sobald es auf den Befehl **RET** trifft. Näheres siehe unten!

Die nachfolgende Prozedur wird mit dem Befehl CALL READ_AX aufgerufen. Der Call-Befehl rettet zuerst den aktuellen Befehlszeiger auf den Stack, führt dann die Prozedur aus und holt den Befehlszeiger beim **RET**-Befehl wieder vom Stack zurück.

```
.....           ; Hauptprogrammm
CALL  READ_AX
.....
INT   20h       ; Ende des Programms
```

READ_AX PROC Near

```
push   BX      ; Es brauchen natürlich nur diejenigen
push   CX      ; Register gerettet werden, die von
push   DX      ; der Prozedur verändert werden!

Mov    BX, 0   ; enthaelt Zwischenergebnis
Mov    CX,10   ; für Stellenverschiebungen
```

```
Newchar  Mov   AH, 1    ; keyboard input function
         INT   21h
         CMP   AL, 13   ; RETURN ?
         JE    EXIT

         SUB   AL, 48   ; ASCII-Codes von Zahlen liegen zwischen 48
         JB    Piep     ; und 58. Vorzeichen-Flag wurde gesetzt.
         CMP   AL, 9
         JA    Piep
         CBW            ; Byte in AL to word in AX

         XCHG  AX, BX
         MUL   CX
         ADD   BX, AX
         JMP   Newchar

Piep     mov   AH, 2    ; Zeichenausgabe
         mov   DL, 7    ; Piepton
         INT   21h
         JMP   Newchar

EXIT     XCHG  AX, BX

         POP   DX
         POP   CX
         POP   BX

         RET

READ_AX  ENDP
```

(Ob etwa die drei Worte *"READ_AX PROC NEAR"* vom Editor eingerückt werden oder nicht, scheint von der Länge des Prozedurnamens und von der Anzahl der eingeschobenen Tab-Sprünge abzuhängen).

```
WRITE_AX    PROC NEAR

            push    AX      ; Push-Operationen kopieren nur.
            push    CX
            push    DX
            push    SI

            Mov     SI, 0   ; Die Zahl in AX wird erst ziffernweise
                            ; abgespeichert und dann ziffernweise
                            ; ausgegeben.
            Mov     CX, 10  ; für Stellenverschiebung


Next        Mov     DX, 0   ; für Doppelwort-Division
            DIV     CX      ; AX wird durch 10 dividiert
            INC     SI
            Mov     [Zahl+SI], DL     ; den Rest abspeichern
            CMP     AX,0
            JE      Ausgabe

            JMP     Next


Ausgabe     Mov     DL, [Zahl+SI]     ; Ziffer zwecks Ausgabe holen
            ADD     DL, 48            ; Ziffer -> ASCII-Code
            Mov     AH, 2             ; Ziffernausgabe von DL
            INT     21h

            DEC     SI
            CMP     SI, 0
            JNZ     Ausgabe           ; naechste Ziffer ausgeben


            pop     SI
            pop     DX
            pop     CX
            pop     AX

            RET

WRITE_AX    Endp

Zahl        DB 8    Dup(0)
```

String-Aufgaben

1. Mit der DOS-Funktion A soll ein String eingegeben werden (Abschluss mit RETURN) und mit der Funktion 9 wieder ausgegeben werden (achte auf das Dollarzeichen!).

2. Ein String, welcher auch Leerzeichen enthalten soll, wird eingegeben. Danach wird dieser String ohne Leerzeichen wieder ausgegeben.

3. Ein String wird eingegeben. Kleinbuchstaben werden in Großbuchstaben umgewandelt. Der String wird anschließend wieder ausgegeben.

Lösung von Aufgabe 1:

; Ein- und Ausgabe eines Strings
; Die Prozeduren cr (Carriage Return) und Taste werden vorausgesetzt.

```
          call  cr
          call  Eingabe
          call  cr
          call  Ausgabe
          call  taste
          INT   20h

Eingabe   Proc  near
          Push  AX
          push  DX

          Mov DX, Offset text ; keyboard-
          Mov AH, 10          ; input-
          INT 21h             ; function

          pop DX
          pop AX
          RET
Eingabe Endp


Ausgabe Proc near
          push AX
          push BX
          push DX
          push SI

          Mov SI, Offset text
          MOV BH, 0
          MOV BL, [SI+1]     ; enthält Länge des Strings
          Mov [SI+BX+3], '$' ; Dollarzeichen als Abschluss
          Mov DX, SI
          Add DX, 2          ; Stringanfang
          mov AH, 9          ; Print String function
          int 21h
```

```
          pop SI
          pop DX
          pop BX
          pop AX
          RET
Ausgabe Endp

text      db 200  dup(200)
```

Lösung von Aufgabe 2:

```
          call Eingabe
          call cr
          call Ausgabe
          call cr
          call taste
          INT 20h

Eingabe   Proc near
          Push AX
          push DX

          Mov DX, Offset text   ; keyboard-
          Mov AH, 10            ; input-
          INT 21h               ; function

          pop DX
          pop AX
          RET
Eingabe Endp


Ausgabe Proc near
          push AX
          push BX
          push DX
          push SI
          push DI

          Mov SI, Offset text
          MOV DI, Offset neutext
```

```
        MOV CL, [SI+1]    ; enthält Länge des Strings
        MOV CH, 0         ; für Schleife
        INC SI
        INC SI            ; zeigt auf Beginn des Strings

Marke   MOV DL, [SI]
        CMP DL, 32        ; Leerzeichen?
        JE Marke2
        MOV [DI], DL
        INC DI
Marke2  INC SI
        LOOP Marke
        Mov [DI], '$'     ; Dollarzeichen als Abschluss

        Mov DX, Offset neutext
        mov AH, 9         ; Print String function
        int 21h


        pop DI
        pop SI
        pop DX
        pop BX
        pop AX
        RET
Ausgabe Endp

text    db 200   dup(200)
neutext db               ?
```

Rekursive Prozeduren

Prozeduren können auch problemlos rekursiv aufgerufen werden, wie man an folgendem Beispiel sieht:

```
        mov     AX, 4
        call    rek
        call    taste
        int     20h

rek     proc near
        call    cr
        dec     AX
        call    write_ax
        cmp     AX, 0
        je      stop
        call    rek
        call    write_ax
        inc     AX
stop    RET
rek     endp
```

Das Ergebnis dieses rekursiven Prozeduraufrufs ist das folgende:

```
3
2
1
0012
```

Aufgabe: Ändere die obige Prozedur so, dass in der letzten Ausgabezeile Leerzeichen eingefügt werden:

```
3
2
1
00 1 2
```

lokale Prozedurvariablen

Variablen, die nur innerhalb einer Prozedur gebraucht werden, sollten grundsätzlich lokal sein. Bei rekursiven Prozeduren ist es manchmal sogar unumgänglich, daß jeder neue (rekursive) Prozeduraufruf auch neue lokale Variablen erzeugt.

In Pascal werden die Speicherplätze für lokale Variablen direkt im Anschluß an den Maschinencode des gesamten Programms gelegt.
Wird eine Prozedur aufgerufen, so wird in diesem Speicherbereich (genannt Heap, engl. "Haufen") Platz·für lokale Variablen belegt. Beim Verlassen der Prozedur wird dieser Platz wieder frei gegeben.
Man benötigt also eine sog. Zeigervariable, genannt *heapptr*, deren Inhalt angibt, wo der freie Speicherplatz beginnt.
Dieser Zeiger wird als globale Variable realisiert. Er muß am Schluß des Assemblertextes deklariert werden.

Im folgenden Bild belegt der gesamte Programmcode (einschließlich aller Prozedurcodes und dem *heapptr*) die Zellen $100 bis $AA01.

Offset-Zellennummer	Inhalt der Zelle	
$FFFF		Stack
$FFFE		
$FFFD		
….		
….		
….		
….		
….		
$AA05		
$AA04		
$AA03		
$AA02		
$AA01	$AA	heappointer
$AA00	$02	
….		
….		
….		
….		
$100		Programmbeginn
…		
…		
…		
$0		

Benötigt nun eine Prozedur lokale Variablen, so wird der Inhalt der von *heapptr* gleich zu Beginn der Prozedur um den entsprechenden Platz erhöht. Dies muß zu Beginn der Prozedur geschehen, weil die Prozedur sich eventuell selber aufrufen könnte. Direkt vor dem Rücksprungbefehl RET muß der Zeiger natürlich wieder zurückgesetzt werden.
In dem so geschaffenen Platz können nun die lokalen Variablen untergebracht werden.
Betrachte dazu das folgende Beispiel, in welchem die Prozedur zwei lokale Variablen X und Y vom Typ Byte besitzt:

```
            call    rek
            int     20h
rek   proc near
            Push    AX
            Push    SI

            Mov     AX, heapptr       ; In der Prozedur werden lokal 2Byte
            ADD     AX, 2             ; benötigt für die Variablen  X und Y
            Mov     heapptr, AX       ;
            Mov     SI, AX            ; X ↔ [SI-2]   und   Y ↔ [SI-1]
            call    cr

            Mov     AL, [SI-2]         ; X
            Mov     AH, 0
            call    write_AX
            call    taste
            cmp     AX, 0
            JE      stop
            call    rek

stop        call    cr
            Mov     AL, [SI-1]        ; Y
            call    write_AX
            call    taste

            Mov     AX, heapptr
            SUB     AX, 2
            Mov     heapptr, AX
            Pop     SI
            Pop     AX
            RET
rek         endp
********** hier stehen benötigte Prozeduren  **********
heapptr DW      Offset heapptr + 2
feld        DB      1,2,3,4,5,6,7,8,9,10,0,0,0
```

Die Speicherplätze für die lokalen Variablen enthalten anfangs normalerweise irgendwelche zufälligen Zahlen. Im obigen Beispiel wurden sie nur vordeklariert, um das Resultat besser verstehen zu können.

Im obigen Beispiel wird auch überhaupt keine Rechnung mit den lokalen Variablen durchgeführt. Die „zufälligen“ Werte dieser Variablen werden nur ausgegeben.

Das Resultat des obigen Programms ist das folgende:

```
1
3
5
7
9
0
0
10
8
6
4
2
```

Parameterübergabe call-by-value

Die Parameterübergabe an eine Prozedur geschieht üblicherweise mit Hilfe des Stacks. Bei der *Call-by-Value*-Übergabe wird der Wert (value) des zu übergebenden Parameters auf den Stack gelegt. Dies muß direkt vor dem Prozeduraufruf geschehen.
Natürlich kann man auch mehrere Parameter so übergeben.

Die Prozedur selbst kann mit Hilfe des Basepointers BP jederzeit auf diesen Wert zugreifen.
Zu beachten ist, daß auf dem Stack grundsätzlich nur 16-bit-Worte abgelegt werden können.

Direkt nach der Rückkehr aus dem Unterprogramm muss der vorher auf den Stack gelegte Value-Parameter dort wieder entfernt werden. Ansonsten könnte dieser Wert auf dem Stack als falsche Rücksprungadresse mißverstanden werden. Dies läßt sich am besten durch den Befehl `ADD SP, 2` bewirken. (Andere Möglichkeiten werden später in einer Aufgabe behandelt).
Falls mehrere Parameter übergeben wurden, so muss der Stackpointer natürlich entsprechend wieder erhöht werden.

Im folgenden Beispiel wird eine Prozedur beschrieben zur Ausgabe einer Ziffer. Der entsprechende Pascal- oder Delphi-Befehl lautet: *Ziffernausgabe(8)* Vergleiche dazu auch die Stack-Belegung ! Um das Verständnis des Problems nicht unnötig zu verkomplizieren, werden keine Register gerettet.

```
     MOV  AX,8
     PUSH AX

     Call ZAusgabe
     ADD  SP,2          ;möglich wäre auch: POP AX

     Call Taste
     INT  20h

ZAusgabe proc near
     MOV  BP,SP
     ADD  BP,2
     MOV  DL,[BP]
     ADD  DL,48
     MOV  AH,2
     INT  21h
     RET
ZAusgabe Endp
```

	….	
	….	
SP vor dem Push AX Befehl →	….	
	0	
SP nach Push AX Befehl →	8	
	…	Rücksprung-Adresse
SP nach call ZAusgabe,→	…	

Damit eine Prozedur mit call-bye-value-Parameter auch rekursiv aufgerufen werden kann, muß natürlich der alte BP-Wert gerettet werden, bevor man das BP-Register benutzt, um auf den Übergabeparameter zuzugreifen.
Aber auch bei nicht-rekursiven Prozeduren sollte das BP-Register grundsätzlich immer erst gerettet werden. Man kann schließlich nicht ausschließen, daß das aufrufende Programm selbst auch mit dem Basepointer arbeitet.

Im folgenden Beispiel wird eine Prozedur geschrieben zur Berechnung des Quadrates einer Zahl. Vergleiche dazu auch die Stack-Belegung !

	….	
	….	
SP vor dem Push z Befehl →	….	
	z-High	
SP nach Push z Befehl →	z-Low	
	…	Rücksprung-Adresse
SP nach call quadrat, vor push BP →	…	
	BP-High	
SP nach Push BP →	BP-Low	← SP=BP nach Mov BP,SP
	AX-High	
	AX-Low	
	…	
	…	

Zunächst das Hauptprogramm:

```
                push    z
                call    quadrat
                ADD     SP,2
                call    cr
                mov     AX, 5
                call    write_AX
                call    cr

                push    ax
                call    quadrat
                ADD     SP, 2

                call    cr
                call    taste
                int     20h
z               dw      3

quadrat proc near               ; ein value-Parameter
                push    BP
                Mov     BP, SP  ; Der Value-Parameter liegt nun in   [BP+4]
                push    AX
                push    CX
                push    DX      ; wird bei Multiplikation geändert

                Mov     AX, [BP+4]
                Mov     CX, [BP+4]
                MUL     CX
                call    write_AX
```

```
        pop     DX
        pop     CX
        pop     AX
        pop     BP
        RET
quadrat endp
```

Dass diese Parameterübergabe auch rekursiv funktioniert, sieht man an folgendem Programm:

```
;Program Quadratzahlen
                push    z
                call    quadratrek
                ADD     SP, 2
                call    taste
                int     20h
z               dw      1

quadratrek proc near            ; ein value-Parameter
                push    BP
                Mov     BP, SP  ; Der Value-Parameter liegt nun in [BP+4]
                push    AX
                push    CX
                push    DX      ; wird bei Multiplikation geändert

                Mov     AX, [BP+4]
                Mov     CX, [BP+4]
                CMP     CX, 10
                JE      STOP
                MUL     CX
                call    cr
                call    write_AX
                INC     CX
                push    CX
                call    quadratrek
                ADD     SP, 2

STOP            pop     DX
                pop     CX
                pop     AX
                pop     BP
                RET
quadratrek endp
```

Aufgabe: Was würde der folgende Prozeduraufruf bewirken:

```
push z
call quadrat
pop z
```

Hinweis: Was geschieht, wenn die Prozedur das globale z oder den Übergabeparameter ändert ?

Parameterübergabe Call-by-Reference

Auch bei der Reference-Übergabe von Parametern wird der Stack benutzt. Allerdings wird hier nicht der Wert (Value) der Übergabevariablen abgelegt, sondern die Nummer der Speicherzelle, in der sich die Variable befindet. Damit wird praktisch ein Zeiger (auf die Variable) übergeben.

Damit dürfte auch klar sein, daß man bei dieser Übergabeform nur Variablen (und keine Werte) übergeben kann.

Die Prozedur selbst kann nur mit Hilfe eines Indexregisters auf diesen Zeiger zugreifen. Im folgenden Beispiel wird wieder eine Prozedur geschrieben zur Berechnung des Quadrates einer Zahl.

(zunächst das Hauptprogramm):

```
            call    cr
            mov     AX, z
            call    write_ax        ; 6 (=Wert von z) ausgeben

            mov     AX, Offset z    ; Übergabe des Zeigers auf die
                                      Variable
            push    AX
            call    quadrat
            ADD     SP, 2           ; wegen sauberer Stackverwaltung

            call    cr
            mov     ax, z
            call    write_ax        ; 36 (=Wert von z) ausgeben

            call    cr
            call    taste
            int     20h
z           dw      6
```

```
quadrat proc near                ; mit 1 Reference-Parameter
        push    BP               ; Retten

        mov     BP, SP           ; Jetzt ist BP ein Zeiger auf die
        mov     BP, [BP+4]       ; Reference-Variable

        push    AX               ; Retten
        push    CX
        push    DX               ; wegen Multiplikation

        mov     AX, [BP]         ; hier findet die eigentliche
                                   Quadratur statt.
        mov     CX, [BP]
        MUL     CX
        mov     [BP], AX

        pop     DX
        pop     CX
        pop     AX
        pop     BP
        RET
quadrat endp
```

Aufgaben

1. Vergleiche die beiden Aufrufe für call-by-reference:

```
push  AX                    push  AX
call  Prozedur              call  Prozedur
ADD   SP, 2                 pop   AX
```

2. Was würde der Befehl PUSH Offset z bewirken ? Mit welchem kürzeren Befehl wäre dieser Befehl identisch ?

3. Warum könnte man bei der Reference-Übergabe nicht folgendermaßen vorgehen:
   Man legt den Wert, des Reference-Parameters auf den Stack. Dann speichert man diesen Wert innerhalb der Prozedur in einer lokalen Variablen ab. Es folgt die eigentliche Prozedur. Am Prozedurende legt man den (neuen) Wert der entsprechenden lokalen Variablen auf dieselbe Stelle des Stacks zurück. Hier kann der neue Wert vom Hauptprogramm wieder übernommen werden.
   Hinweis: Betrachte das folgende Pascal-Programm !

```
Program Reference;
VAR x: INTEGER;

Procedure Unbekannt(VAR a: INTEGER);
   Begin
     a := a*a;
     x := 5;
     WRITELN(a, x)
   End;

BEGIN
  x := 4;
  Unbekannt(x)
END.
```

4. Der Befehl *RET 6* ist in seiner Wirkung identisch mit der Befehlsfolge
   RET
   ADD SP,6

   Erläutere, wie dieser Befehl für Prozeduren eingesetzt werden kann !

Funktionen

Funktionen liefern immer ein Ergebnis. Dieses Ergebnis muß irgendwie gespeichert werden. Bei Funktionsaufrufen handelt es sich immer um eine Zuweisung wie etwa im folgenden Pascal-Beispiel: z := Funktionsname(x, y)

Fast immer müssen bei Funktionen Parameter übergeben werden.

Die obige Pascal-Zuweisung läßt sich folgendermaßen übersetzen:

```
push  z       ; Damit wird auf dem Stack eine Speicherstelle zur
                Übernahme des Funktionsergebnisses angelegt.
push  x       ; Parameterübergabe
push  y
call  Funktionsname
ADD   SP, 4
Pop   z  ;    Damit enthält z das Ergebnis der Funktion
```

Die Funktion selbst kann, wie schon bekannt, mit einem Indexregister auf die Parameter zugreifen und auch entsprechend das Funktionsergebnis ablegen.

Im folgenden Beispiel wird die Fakultät einer Zahl mit Hilfe einer rekursiven Funktion berechnet.

Zunächst das entsprechende Pascal-Programm:

```
Program Fakultaet;
  Function fak(n: INTEGER): INTEGER;
    Begin
      IF n = 1 THEN fak := 1
      ELSE fak := n * fak(n-1)
    End;

  BEGIN
    WRITELN(fak(5))
  END.
```

Nun ein entsprechendes Assembler-Programm:

```
            Mov   CX, 5          ; Es soll 5! berechnet werden

            push  AX             ; für Aufnahme des Fkt.-Ergebnisses
            push  CX             ; Fkt.-Parameter
            call  fakultaet
            ADD   SP,2           ; Parameter übergehen
            pop   AX             ; Fkt-Ergebnis

            call  cr             ; Carriage Return + Linefeed
            call  write_AX       ; Fkt-Ergebnis steht in AX
            call  taste
            int   20h


fakultaet  proc near
            push BP              ; Retten
            Mov  BP,SP           ; Funktionsergebnis soll nach [BP+6]
                                 ; Value-Parameter ist in [BP+4]

            push AX              ; Retten
            push CX
            push DX              ; wegen Multiplikation, High-Word in DX

            Mov  CX,[BP+4]       ; Fkt.-Parameter
            CMP  CX, 1
            JE   Fkt_Ende        ; Fak(1) = 1

            push AX              ; für Fkt.-Ergebnis von Fak(n-1)
            DEC  CX              ; neuer Fkt.-Parameter für Fak(n-1)
            push CX
            INC  CX
            call fakultaet
            ADD  SP,2            ; Parameter übergehen
            pop  AX              ; Ergebnis Fak(n-1)

            MUL  CX              ; Fak(n) = n*Fak(n-1)
            Mov  [BP+6], AX      ; Funktionsergebnis Fak(n)
            JMP  Stop

Fkt_Ende    Mov  [BP+6], 1       ; Funktionsergebnis Fak(1)
Stop        Pop  DX
            Pop  CX
            Pop  AX
            Pop  BP
            RET
fakultaet   endp
```