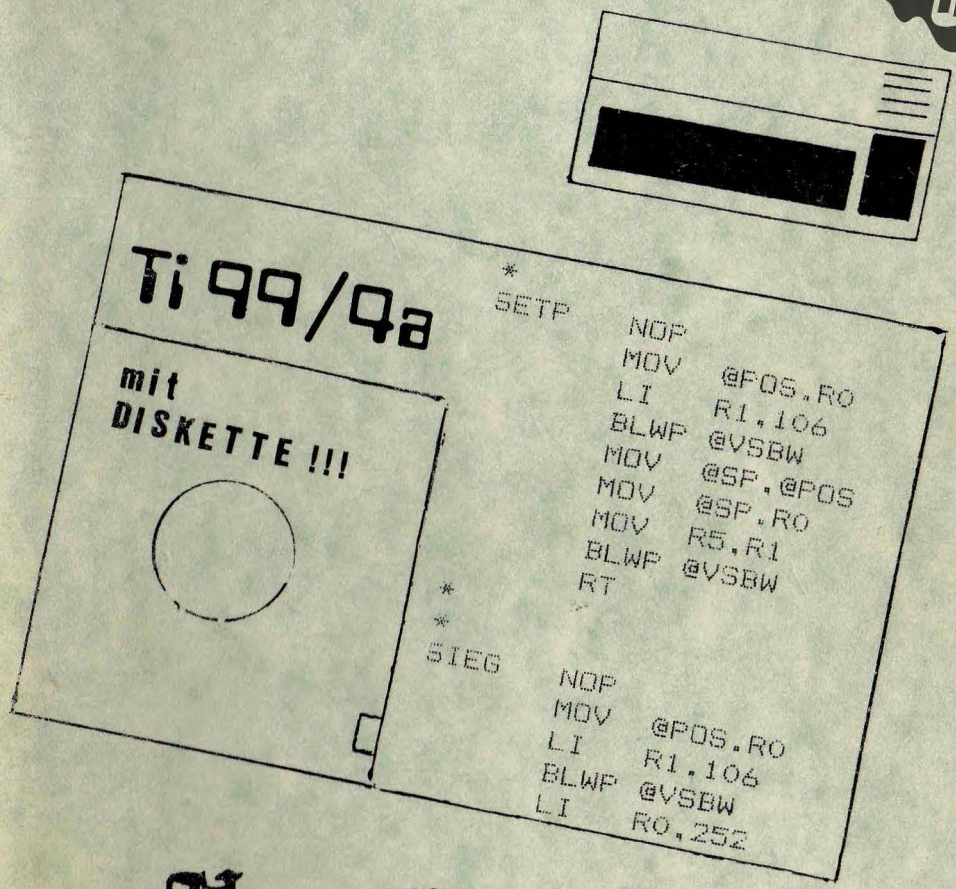
RAUSCH & HAUB



Hagera

ASSEMBLER-KURS II

f. TMS9900 (TI-99/4a)

von H.-G. RAUSCH

HAGERA

A S S E M B L E R - K U R S

3. überarbeitete Fassung 1985

AUTOR: H.-G. RAUSCH

Hans-Georg Rausch
Assembler Kurs für den TI-99/4a Homecomputer
in Verbindung mit dem Editor/Assembler-Paket
und einer Speichererweiterung.

3. überarbeitete Fassung

5300 Bonn.

VORWORT

Sie haben sich dazu entschieden, die Assemblersprache des TI-99/4a zu erlernen. Hierzu sollen Ihnen dieser Einführungskurs und die beiliegende Diskette eine wertvolle Hilfe sein. Der Kurs ist durch die Benutzung des TI-Assemblers entstanden. Er kann und sollte nicht auf andere Assembler-Versionen übertragen werden.

Sie haben vielleicht schon Erfahrungen mit einem anderen Kurs gemacht. Man sitzt vor seinem Gerät, lernt mühsam einige grundlegende Begriffe der Assembler-Sprache und versucht, die gewonnenen Erkenntnisse am TI auszuprobieren. Meißtens ohne Erfolg und nach einiger Zeit mit steigendem Unwollen.

Damit Ihnen dies bei unserem Kurs nicht so ergeht, nachfolgend ein paar einleitende Worte:

Wichtig ist, daß Sie fast alles außer acht lassen, was Sie über andere Assembler-Sprachen gelernt haben. Die Maschinensprache des TI-99/4a ist mit keiner anderen zu vergleichen.

Viele von Ihnen werden bereits eine Programmiersprache beherrschen, in den meißten Fällen BASIC. Basic und Assembler arbeiten grundverschieden. Die Denkweise der Assembler-Sprache ist eine völlig andere. Trotzdem haben wir uns bemüht, Parallelen zum TI-(Extended)-Basic aufzuzeigen und Bekanntes aus dieser relativ einfach zu erlernenden Sprache in unseren Kurs einfließen lassen.

Zu bemerken sei außerdem, daß Sie nach diesem Kurs sicher kein perfekter Programmierer in der TI-Maschinensprache sind. Dieser Kurs soll lediglich anhand von recht einfachen Beispielen in die Maschinensprache des TI einführen. Weitere Informationen können Sie aus dem Handbuch zum Editor/Assembler entnehmen.

Der Vollständigkeit halber haben wir diesem Kurs eine Tabelle der Assemblerbefehle, der Maschinen-Unterprogramme sowie einige nützliche Detailinformationen beigefügt. Die Diskette enthält alle in diesem Buch gelisteten Programme und Teilprogramme im Quellen- und Objektcode, nicht jedoch die kurzen Beispiele.

Alle erläuterten Beispiele sind, um dies vorweg zu nehmen, ohne Erweiterung des Programms nicht sichtbar lauffähig. Halten Sie sich daher beim Programmieren möglichst an die aufgegebenen Probleme am Ende des Einleitungskurses.

Für Hinweise und Ratschläge in jeder Form zu diesem Kurs oder einem anderen HAGERA-Programm sind wir stets dankbar.

Hans-Georg Rausch
(Autor)

Zu diesem Kurs

Um mit diesem Kurs die Assembler-Sprache des TI-99/4a erlernen zu können, benötigen Sie die folgende Konfiguration:

- Konsole TI-99/4a
- Monitor oder (Farb-)Fernseher
- Editor/Assembler Modul und Disketten
- Speichererweiterung (Box oder extern)
- Diskettenlaufwerk

Ein Drucker zum Auslisten der von Ihnen entwickelten Programme sei empfohlen.

Der Kurs ist in mehrere Kapitel unterteilt. Im ersten Kapitel befindet sich die Einführung in die Maschinensprache des TI-99/4a mit kleinen Beispielen; zum Beispiel, wie man ein Zeichen auf den Bildschirm bringt oder Töne produziert.

Der Zweite Teil befasst sich mit der Entwicklung kürzerer Programme und der Lösung kleiner Probleme.

Im Dritten Teil finden Sie den Ausschnitt aus einem Spielprogramm, an dem Sie die gewonnenen Kenntnisse vertiefen können. Erweitern Sie das Spiel wie dort angegeben oder nach ihren eigenen Wünschen.
Der Spielausschnitt ist Zeile für Zeile dokumentiert, so daß Sie den Programmablauf Schritt für Schritt nachvollziehen können.

Teil IV enthält Lösungsvorschläge für die

meißten Übungsaufgaben aus diesem Buch.

Die Teile V und VI beinhalten eine Übersicht der Mnemonics und Unterprogramme. Außerdem finden Sie hier wichtige Speicheradressen und Utilities.

Im Teil VII des Kurses werden Begriffe, deren Bedeutung sich nicht schon aus dem Text ergibt, ausführlich erläutert. Schlagen Sie dort nach, wenn Sie etwas nicht wissen.

Welche Programme auf der Diskette enthalten sind, erfahren Sie im Anhang (Teil 8.).

Trotz sorgfältiger Überarbeitung kann dieses Buch keinesfalls alle Fragen, die im Zusammenhang mit dem Erlernen der Assembler Sprache auftreten, klären. Auch kann nicht garantiert werden, daß der Innalt dieses Buches oder die Diskette völlig frei von Fehlern sind. Ein Anspruch auf Schadenersatz besteht nicht, auch wenn entstehende Schäden durch fehlerhafte, falsche oder mangelnde Infomation entstanden sind. Eventuelle Ansprüche beschränken sich in jedem Fall auf den Kaufpreis dieses Kurses.

Wir hoffen nun, daß Sie viel Freude an diesem Kurs haben. Arbeiten Sie am Rechner anstatt theoretische Überlegungen anzustellen. Dies wünscht Ihnen Ihre

INHALT

QUELLENNACHWEIS:

Texas Instruments Learning Center
- Handbuch zum Editor/Assembler

URHEBERRECHTLICHER HINWEIS:

1.1

I. EINFÜHRUNG

1. Umgang mit dem Editor/Assembler

Setzen Sie das EDITOR/ASSEMBLER-Modul in den Modulschacht ein und schalten Sie die Geräte wie gewohnt an. Wählen Sie von der Hauptwahlliste

- 2 - für EDITOR/ASSEMBLER

schieben Sie die im Editor/Assembler-Paket enthaltene Diskette, Part A, in Laufwerk 1 und wählen Sie den Editor mit 1-EDIT von der Hauptwahlliste, die wie folgt aussieht:

1 - TO EDIT
2 - TO ASSEMBLER
3 - TO LOAD AND RUN
4 - TO RUN
5 - TO RUN PROGRAM FILE

Es erscheint die Editor-Auswahlliste:

1 - LOAD
2 - EDIT
3 - SAVE
4 - PRINT
5 - PURGE

Wenn Sie -2- drücken, wird der Editor geladen und steht für das Eintippen von Assembler Programmen bereit.

Wer mit dem TI-WRITER arbeitet, kennt den Umgang mit dem Editor bereits in seinen

wesentlichen Funktionen. Sie sind im Handbuch zum Editor/Assembler gut erläutert. Wir wollen hier jedoch demjenigen, welcher der englischen Sprache nicht mächtig ist oder nicht über den TI-Writer verfügt, einen kurzen Überblick bieten.

Der EA-Editor ist ein bildschirmorientierter Editor mit folgenden Funktionen:

FCTN-1 löscht das Zeichen unter dem Cursor. Die Zeile wird zusammengezogen, damit keine Lücke zurückbleibt.

FCTN-2 erlaubt das Einfügen einzelner Zeichen.

FCTN-3 löscht die Zeile, in der sich der Cursor befindet.

FCTN-4 Die vorangegangenen 24 Zeilen werden angezeigt (Rückwärts blättern).

FCTN-5 Zeigt den jeweils rechts folgenden Textausschnitt der angezeigten 24 Zeilen. Wenn Sie die Funktion im rechten Ausschnitt benutzen, wird der linke Ausschnitt angezeigt. Es gibt 3 horizontale Ausschnitte zu je 40 Zeichen, die sich gegenseitig überlappen.

FCTN-6 Die folgenden 24 Zeilen werden angezeigt (vorwärts blättern).

FCTN-7 Bewegt den Cursor nach rechts wie bei einer Schreibmaschine zu einer Tabulator Position. Wenn der Tabulator nicht verändert wird, steht er auf folgenden Positionen:

(1) Zeilenanfang (Beginn des LABEL-Felds)
(8) Beginn des Mnemonic-Felds
(13) Beginn des Operanden-Felds
Sowie (26),(31),(46),(60) und (80).

Auf die Begriffe Label, Mnemonic und Operand gehen wir später noch ein.

FCTN-8 Erlaubt das Einfügen einer Zeile an der Cursor-Position.

FCTN-9 Schaltet in den Command-Modus, in dem die Editor-Funktionen zugänglich sind. Wenn Sie sich im Command Mode befinden, gelangen Sie zurück zur EA-Auswahlliste.

Mit den Cursorfunktionen können Sie den Cursor beliebig auf dem Bildschirm bewegen. Wenn Sie am linken Rand FCTN-S drücken (Cursor nach links), werden Pseudo-Zeilennummern angezeigt, die Ihnen helfen sollen, einen bestimmten Programmteil zu finden. Am oberen und unteren Rand des Bildschirms scrollt der Text jeweils eine Zeile weiter, am rechten Rand wird das nächste horizontale Fenster angezeigt (siehe Next Window Funktion).

Im Command Mode stehen Ihnen folgende Funktionen zur Verfügung, die ausführlich im EA-Handbuch beschrieben sind.

EDIT - Zurückschalten in den Edit-Modus, um Text an angezeigter Stelle zu schreiben oder zu ändern.

FIND - Suchen nach einer Zeichenkette.

REPLACE - Austauschen einer Zeichenkette.

MOVE - Verschieben eines Programmblocks.

INSERT - Einfügen eines Diskettenfiles an eine bestimmte Stelle im Editor durch Angabe von Pseudo-Zeilennummern.

COPY - Kopieren eines Programmblocks.

SHOW - Bildschirmausgabe ab einer bestimmten Zeilennummer.

DELETE - Löschen eines Programmblocks.

ADJUST - Zeigt die Zeichen 75 bis 80 einer Zeile, die normalerweise nicht sichtbar sind. Wenn Sie in der Editor zurückkehren wollen, müssen Sie zunächst den Cursor aus diesem Bereich herausbewegen.

TAB - Ermöglicht das Ändern der Tabulator Positionen.

HOME - Bewegt den Cursor an die obere linke Ecke des Bildschirms.

Mithilfe dieser Funktionen können Sie nun Ihr Programm eingeben. Probieren Sie gleich einmal aus:

*END OF FILE sehen Sie am oberen Rand.

Drücken Sie <ENTER> und <~~SHIFT~~ FCTN><S>

0001
~~0002~~ *END OF FILE

erscheint auf dem Bildschirm. Die erste Zeile steht jetzt zur Eingabe bereit:

```
0001          DEF  PROG1
0002          REF  VSBW
0003 PROG1    LI   R0,>0000
0004          LI   R1,>4100
0005          BLWP @VSBW
0006 LOOP     NOP
0007          JMP  LOOP
0008          END
```

Damit haben Sie bereits Ihr erstes kleines Assembler-Programm geschrieben. Schauen Sie sich gleich einmal an, was Sie geschrieben haben. Dazu müssen Sie das Programm in Maschinencode umwandeln, denn so, wie das Programm jetzt aussieht, kann Sie der TI leider nicht verstehen. Das Umwandeln in Maschinencode geschieht folgendermaßen:

Drücken Sie FCTN-9, um in den Command-Mode zu gelangen. Drücken Sie erneut FCTN-9, und sie erhalten die EDITOR-Auswahlliste. Speichern Sie jetzt das Programm mit der SAVE-Funktion ab. Wenn Sie nur ein Laufwerk besitzen, müssen Sie die PART-A-Diskette herausnehmen und ihre eigene initialisierte Diskette einlegen. Wenn Sie zwei Laufwerke besitzen, legen Sie die PART-A-Diskette immer in Laufwerk 1 und ihre eigene Diskette in ein anderes Laufwerk. Beantworten Sie VARIABLE 80 FORMAT mit Y. Es erscheint die Nachricht FILENAME?
Geben Sie ein: DSK1.PROGRAM <ENTER>.

Ihr Programm wird jetzt abgespeichert. Wenn es fertig abgespeichert ist, erscheint wieder die Editor-Auswahlliste. Legen Sie jetzt ggf. wieder die PART-A-Diskette ein und drücken Sie FCTN-9.

Wenn Sie jetzt 2-ASSEMBLE auswählen, erscheint die Nachricht 'LOAD ASSEMBLER Y/N' auf dem Bildschirm. Laden Sie den Assembler mit Y <ENTER>. Sobald der Assembler geladen ist, tauschen Sie ggf. die Disketten wieder aus. Das Assemblerprogramm beginnt einen Eingabedialog mit Ihnen:

SOURCE FILE NAME?
Sie geben ein: DSK1.PROGRAM
(oder DSK2.PROGRAM je nachdem, wo sich ihre Programmdiskette befindet).

OBJEKT FILE NAME?
Sie geben ein: DSK1.OPROGRAM bzw. DSK2...

LIST FILE NAME?
Sie geben ein: <ENTER>, wenn Sie kein Listing wünschen, oder eine Datei, auf der Sie ein Listing wünschen. Das Listing erfolgt im Variable-80 Dateiformat, z.B.:
PIO. ← der Punkt ist notwendig!

OPTIONS?
Sie können bzw. müssen dem Assembler verschiedene Optionen angeben.

Sie geben ein: R Bestimmt Registesymbole (setzt R0 bis R15 gleich zu 0 bis 15. R muß eingegeben werden (beim gezeigten und bei allen anderen Beispielen aus diesem Buch).

Sie geben ein: L wenn Sie eine List-Datei gewählt haben. Wenn Sie L nicht spezifizieren, wird keine Liste erstellt.

Sie geben ein: S wenn ein Speicherauszug im Listing enthalten sein soll.

Sie geben ein: C wenn die Assemblierung im CONDENSED FORMAT, also in komprimierter Form erfolgen soll. Das spart Diskettenplatz.

Wenn Sie mehr als eine Option wünschen, geben Sie die Kennbuchstaben ohne Leerzeichen an, also z.B.: RL

Für den Anfang sollte R genügen. Probieren Sie aber ruhig alle Möglichkeiten mit diesem kleinen Programm aus.

Wenn Sie Option beantwortet haben, erscheint die Nachricht ASSEMBLER EXECUTING und das Maschinenprogramm wird generiert. Tritt während des Prozesses ein Fehler auf, so wird dieser angezeigt:

SYMBOL TABLE OVERFLOW
CANT GET COMMON
CANT GET MEMORY
DSR ERROR

Bei diesen Fehlern bricht der Assembler ab. Es handelt sich um sogenannte schwerwiegende Fehler, die der Überarbeitung oft des gesamten Programms bedürfen.

Es gibt jedoch auch eine ganze Reihe leichter Fehler, die zumeist auf Tippfehlern bei der Eingabe im Editor beruhen:

SYNTAX ERROR
INVALID REFerence
OUT OF RANGE
MULTIPLE SYMBOLS
INVALID MNEMONIC
BAD REFERENCE
INVALID TERM
INVALID REGISTER
SYMBOL TRUNCATION
UNDEFINED SYMBOL
COM TABLE OVERFLOW
END ASSUMED
COPY ERROR

Es gibt weitere Fehlernachrichten, die jedoch mit Programmfehlern zusammenhängen, welche in diesem Kurs nicht auftreten können. Eine Beschreibung dieser Fehler würde den Rahmen dieses Kurses sprengen.

Nach diesen Fehlern bricht der Assembler nicht ab, sondern zeigt die EDITOR-Zeile mit dem Fehler an und fährt fort. Wenn ein Fehler auftritt, notieren Sie sich diesen, um ihn später zu korrigieren.

Die Fehlernachricht END ASSUMED ist ohne große Bedeutung. In diesem Fall haben Sie als letzte Zeile in Ihrem Programm lediglich 'END' vergessen.

Wenn keine Fehler gefunden werden, endet der Prozess mit der Nachricht:

```
0000 ERRORS
PRESS ENTER TO CONTINUE
```

Wenn Sie ENTER drücken, gelangen Sie zur Hauptwahlliste zurück. Jetzt können Sie mit LOAD AND RUN Ihr Programm ausprobieren.

Erneut gibt es einen Eingabedialog:

```
FILE NAME?
```

Sie geben ein: DSK1.OPROGRAM <ENTER>

Die Nachricht erscheint ein weiteres Mal, für den Fall, daß Sie weitere Assembler-Files laden möchten. Da dies nicht der Fall ist, drücken Sie einfach <ENTER>.

PROGRAM NAME?
Sie geben ein: PROG1 <ENTER>

PROG1 ist die Eingangsstelle, die Sie in der DEF-Anweisung spezifiziert haben. Hier startet das Programm. Sie können in einem Programm mehrere solcher Eingangsstellen angeben (mit DEF).

Jetzt startet sofort Ihr Maschinenprogramm. Und siehe da...

...ein 'A' erscheint in der oberen linken Ecke des Bildschirms!

Bevor Sie in das Kapitel Bildschirmausgabe einsteigen, möchten wir Ihnen sagen, weshalb dieses 'A' auf dem Bildschirm erscheint. Es ist die Voraussetzung für alle nachfolgenden Kapitel und Beispiele.

Dazu gehen wir das Programm Zeile für Zeile durch. Zunächst zum Zeilenaufbau:

```
0001            DEF  PROG1
```

Links sehen Sie die Pseudo-Zeilennummer. Wir haben sie in allen Beispielen angegeben, um das Abzählen bei der Eintippkontrolle zu vermeiden. Die eigentliche Eingabe beginnt hinter der Zeilennummer.

Die ersten sechs Positionen werden LABEL-Feld genannt. Hier befinden sich die Symbole, vordefinierten Symbole, Eingangsadressen und ähnliches. Ein solches Symbol ist zum Beispiel LOOP. Auf das Symbol folgt mindestens ein Leerzeichen. Der besseren Übersichtlichkeit wegen sollten Sie immer mit den Mnemonics in Spalte 8

beginnen. Wenn also wie oben kein Symbol vorhanden ist, drücken Sie TAB, und der Cursor befindet sich an der richtigen Stelle, um einen Mnemonic einzugeben.

DEF ist ein solcher Mnemotechnischer Code, also eine Abkürzung für etwas, welches den Sinn der Abkürzung verständlich macht. DEF bedeutet DEFine und definiert eine Eingangsstelle im Programm. In unserem Fall ist dies PROG1. Um auch hier eine gewisse Übersicht zu bewahren, beginnen wir mit den Operanden ab Spalte 13. Operanden sind alles, was hinter einem Mnemonic steht (mit Ausnahme von Kommentaren), also auch PROG1 im Fall der Zeile 1 (DEF). Die meißten Mnemonics benötigen Operanden, die sie verarbeiten, manche davon einen, andere zwei oder mehr. Im Fall von DEF kann dies unterschiedlich sein.

```
0001          DEF  PROG1
```

aber auch z.B.:

```
0001          DEF  PROG1,EING,START
```

Operanden werden durch Kommata getrennt.

DEF muß in jedes Programm eingeschlossen werden, da Sie es sonst nicht starten können.

```
0002          REF  VSBW
```

lautet die nächste Zeile. REF ermöglicht es, bestimmte vordefinierte Symbole, welche die Eingangsstellen von eingebauten (ROM-) Unterprogrammen bezeichnen, in den Assemblierungsprozess einzubinden. VSBW ist

ein solches Unterprogramm. Es steuert unter anderem die Bildschirmausgabe.

VSBW bedeutet VIDEO SINGLE BYTE WRITE und schreibt, wenn es aufgerufen wird, ein Zeichen in das VDP-RAM. Dieser Speicherbereich ist von großer Bedeutung für Bildschirmausgabe, Farben, Zeichendefinition, Arbeiten mit Dateien und vielem mehr.

Wie solch ein Unterprogramm benutzt wird, werden Sie noch sehen.

```
0003 PROG1  LI   R0,>0000
```

Hier haben wir eine Typische Assembler-Zeile mit Label, Mnemonic und zwei Operanden. PROG1 kennen wir bereits. Es ist unsere Programmeintrittsstelle, die wir mit DEF definiert haben.

Es folgt der Mnemonic LI. Mit LI können wir einen bestimmten Wert in ein Register laden. Was Register sind, erfahren Sie im nächsten Kapitel. Wichtig ist, daß der Wert >0000 in das Register geladen wird, also eine 16-Bit Hexadezimalzahl. Die Register werden WS oder Workspace Register genannt.

```
0004          LI   R1,>4100
```

Wieder ein Ladebefehl. Diesmal wird >41, also Dezimal 65, in Register 1 geladen. Das Symbol Feld ist diesmal nicht besetzt.

```
0005          BLWP §VSBW
```

Mit BLWP rufen wir ein Unterprogramm auf. Dies kann ein eingebautes sein wie VSBW, aber auch solche, die wir selbst definieren. Das Unterprogramm VSBW benötigt in Bestimmten Regitern Werte für die Exekution.

In Register 0 die Adresse, in die im VDP-RAM geschrieben werden soll. Daher LI R0,>0000, weil wir in Adresse >0000 schreiben wollen. Die Adressen ab >0000 im VDP-Ram sind für die Bildschirmausgabe vorgesehen.

In Register 1 der Wert, der geschrieben werden soll. Da Register immer 16-Bit enthalten, ein zu schreibender Wert aber stets aus 8 Bit besteht, muß dieser in die ersten 8 Bits des Registers geladen werden. In unserem Beispiel:

```
BYTES...:   Erstes     Zweites
WERT....:   Hochwert.  Niedrigwert.
REGISTER:   ---- ----  ---- ----
WERT....:   0100 0001  0000 0000    (= >4100)
```

Nach LI steht der Wert im Register.

Wenn VSBW aufgerufen wird, schreibt es den Wert aus den ersten 8 Bit im Register 1 in den VDP-Speicherplatz, der im Register 0 gegeben ist.

Damit das Programm nicht abstürzt, beenden wir es mit einer Warteschleife. Sie ist Endlos und kann nur durch Abschalten des Computers beendet werden.

```
0006 LOOP   NOP
0007        JMP  LOOP
```

In Zeile 6 finden Sie die Eingangsstelle der Schleife. Dort steht als Mnemonic NOP. NOP benötigt Speicherplatz, führt aber ansonsten nichts aus. NOP = No OPeration.

In Zeile 7 springen Sie zur Eingangsstelle der Schleife zurück. JMP können wir in etwa mit GOTO im Basic vergleichen, allerdings gibt es gewisse Beschränkungen, die bei den Erläuterungen zu den Mnemonics aufgeführt sind. Hinter JMP steht sie Sprungadresse, zu der gesprungen werden soll. Im Basic haben wir statt der Sprungadressen Zeilennummern.

```
0008        END
```

Jedes Assemblerprogramm sollte mit END abgeschlossen werden. Andernfalls erscheint die Fehlernachricht END ASSUMED beim Assemblieren.

Damit haben Sie das erste Kapitel bereits geschafft. Üben Sie den Umgang mit dem Assembler, indem Sie das Programm wie folgt verändern. Beachten Sie, daß Sie nach jedem Laufenlassen den Rechner aus- und wieder einschalten müssen, bevor Sie weitermachen können.

AUFGABEN:

1. Bezeichnen Sie die Eingangsstelle des Programms nicht mit PROG1, sondern mit START.

2. Drucken Sie das 'A' eine Position weiter rechts.

3. Versuchen Sie, das 'A' eine Position weiter unten zu drucken.

4. Schreiben Sie statt des 'A' ein 'X' auf den Bildschirm.

5. Schreiben Sie mithilfe des Unterprogrammes VSBW das Wort 'AXT' in die obere linke Ecke des Bildschirms.

6. Schreiben Sie das Wort 'AXT' in die Mitte des Bildschirms.

Und wenn es nicht klappt? Dann sollten Sie Kapitel I.1. nochmals aufmerksam lesen. Arbeiten Sie keinesfalls weiter, bevor Sie nicht alle Aufgaben dieser Seite selbständig lösen können. Die Lösungen finden Sie am Ende dieses Buches im Anhang. Aufgabe 6 ist auch auf Diskette vorhanden.

Eine Zeile hat 32 Spalten

Position = Zeilen x 32 + Spaltenzahl

z.B. 332 ≙ 10. Zeile, Spalte 12

deswegen R0 mit Hex laden 20 ≙ 32 recht einfach

1.2

2. Bildschirmausgabe auf dem TI

Wenn Sie anfangen, BASIC zu lernen, ist das erste, eine Nachricht auf den Bildschirm zu bringen. Wir wissen nicht warum, aber bei den Einführungen für Assembler verfährt man normalerweise anders, obwohl auch Sie als angehender Assemblerprogrammierer gern das Ergebnis Ihrer Mühen auf dem Bildschirm sehen würden. Deshalb beginnen wir damit, eine Zeichenkette auf den Bildschirm zu bringen.

Wie dies in TI-Basic funktioniert, wissen Sie. Mit 'PRINT "Guten Tag"' schreibt der Computer diese Zeichenkette an den unteren Rand des Bildschirms. Warum?

Der TI-Computer besitzt einen sogenannten Interpreter, der das Wort 'PRINT' so in für den Computer lesbare Zeichen umformt, daß er diese als Befehl erkennt, eine Zeichenkette ans untere Ende des Bildschirms zu drucken.

Überlegen wir also einmal, was der Computer alles tun und wissen muß.

Zunächst einmal benötigt er die Zeichenkette, die an einer bestimmten Stelle im Speicher abgelegt ist. Bei Basic funktioniert dies zum Beispiel mit Variablen, bei Assembler durch eine Speicheradresse. Diese Adresse muß dem Computer ebenfalls bekannt sein. Weiterhin ist die Bildschirmposition erforderlich, an der die Zeichenkette gedruckt werden soll. Damit der TI weiß, wieviele Zeichen er ausdrucken soll, muß Ihm diese Anzahl ebenfalls mitgeteilt werden. Zu guter letzt

müssen alle diese erforderlichen Daten an bestimmte Stellen plaziert werden, um das gewünschte Ergebnis zu erhalten. Die Informationen nützen nichts, wenn sie falsch plaziert sind.

Zur Erinnerung noch ein Beispiel aus dem Basic:

```
10 Goto 30
20 Print "Nein"
30 Goto 10
```

Sooft wir diese Routine auch mit 'Run' aufrufen - das Wort 'Nein' wird uns der TI niemals ausdrucken, denn die Informationen sind zwar vorhanden, aber an falscher Stelle.

In Assembler benutzen wir anstelle der Zeilennummern bestimmte Speicheradressen. Für die Informationen, die wir zum Beispiel für den Ausdruck einer Zeichenkette benötigen, benutzen wir den Accumulator des TI. Er verarbeitet allein die Informationen durch ein Assembler-Programm. Um den Print-Effekt wenigstens halbwegs zu erreichen, müssen wir folgendes Programm schreiben und assemblieren:

```
0001          DEF  PRINT
0002          REF  VMBW
0003 SATZ     TEXT 'GUTEN TAG!'
0004 PRINT    LI   R0,>2E0
0005          LI   R1,SATZ
0006          LI   R2,>A
0007          BLWP @VMBW
0008 STOP     NOP
0009          JMP  $STOP
0010          END
```

So funktioniert es:

Wie Sie später noch sehen werden, verfügt der TI-99/4a über eine Reihe von Unterprogrammen, die die Arbeit mit Assembler sehr vereinfachen. VSBW haben Sie bereits kennengelernt. Man bringt damit ein Zeichen auf den Bildschirm. Mit einem weiteren dieser Unterprogramme wollen wir uns nun beschäftigen:

VMBW bedeutet 'Video Multi Byte Write'

und bewirkt nichts anderes, als eine Anzahl von Bytes, also eine Zeichenkette, in den Speicherbereich des TI-99/4a zu schreiben, der für die Videoausgabe zuständig ist. Wollen wir dieses oder ein anderes Unterprogramm verwenden, so muß am Anfang des Assemblerprogramms ein Bezug dazu hergestellt werden. Dies geschieht mit Hilfe der REF-Anweisung (REFerence).

In diesem Fall bewirkt REF VMBW, daß später bei der Umwandlung der Mnemo-Codes in ein Maschinenprogramm das Unterprogramm VMBW in den Assemblierungsprozess mit einbezogen wird und so für unsere Verwendung zur Verfügung steht.

Natürlich können auch mehrere Unterprogramme einbezogen werden. Der Assembler des TI kennt eine ganze Reihe mit verschiedensten Aufgaben. Die wichtigsten werden Sie noch kennenlernen.

Zurück zu unserem Programm: Mit REF VMBW haben wir dem Computer gesagt, daß später irgendwann einmal (vielleicht) Bytes, daß heißt Daten, in den Videospeicher des

TI-99/4a geschrieben werden sollen. Das wollen wir jetzt tun. Dazu bedarf es einiger Vorbemerkungen, betreffend den Accumulator des TI.

Der Accu besteht aus einer Reihe von Registern, deren Aufgabe es ist, Daten zu verarbeiten. Das ist bei jedem Computer so, nur ist der TI-Accu ein wenig eigenwillig.

Wie der Accumulator des TI aufgebaut ist, werden Sie noch erfahren und können Sie auch im Editor/Assembler Handbuch nachlesen. An dieser Stelle nur folgendes:

Jedes Register im Accumulator kann mit einem bestimmten Wert geladen werden. Einige Register erledigen Sonderaufgaben; sie liefern zum Beispiel die Parameter für die eingebauten Unterprogramme. In unserem Fall sind dies die Register 0,1 und 2. Sie enthalten die Daten, die das Unterprogramm für die Erledigung seiner Aufgabe braucht.

Das Unterprogramm VMBW benutzt die Register folgendermaßen:

- 0 mit der Adresse des Videospeichers, an der mit dem Beschreiben begonnen werden soll.

- 1 mit der Adresse, an der sich die Daten befinden, welche geschrieben werden sollen.

- 2 mit der Menge der zu übertragenden Daten (Bytes).

VMBW benötigt noch weitere Register, die Sie an dieser Stelle aber nicht interessieren sollten. Wichtig sind zunächst einmal die

Register 0 bis 2.

Wenn Sie sich nun das gezeigte Programm ansehen, werden Sie warscheinlich schnell entdecken, wie die Register zu ihren Daten kommen. Mit der LI-Instruktion. LI ist die Abkürzung - der Mnemonic - für LOAD IMMEDIATE.

Die LI-Instruktion bewirkt, daß ein Register mit einem Wert geladen wird. Dieser Wert kann unterschiedliche Bedeutungen haben, wie Sie gleich sehen werden.

LI R0,>2E0 lautet die nächste Zeile unseres kleinen Programms. LI sagt dem Computer, daß in das nachfolgend genannte Register der darauffolgende Wert geladen werden soll. Der Buchstabe R kennzeichnet einen Wert als Register, das Zeichen > eine Zahl als Hexadezimal. Falls Sie mit dem Hexadezimalen Zahlensystem noch nicht vertraut sind, lesen Sie bitte in den Erläuterungen der Fachbegriffe nach.

Die Zeile sagt dem Computer also, daß er die hexadezimale Zahl 2E0 (Dezimal 736) in das Register Nr. 0 des Accumulators laden soll.

Weshalb soll er das tun?

Erinnern Sie sich noch einmal an die Aufgabe, deren Lösung das Programm liefern soll. Wir wollen eine Zeichenkette an den unteren Rand des Bildschirms schreiben. Um nas tun zu können, müssen wir das Unterprogramm VMBW benutzen, und dieses wiederum erwartet in Regiter 0 (R0) die Adresse, an die bestimmte Daten geschrieben werden sollen. Die Zahl >2E0 sagt also aus,

daß im Unterprogramm VMBW Daten an die Speicherstelle >2E0 im Videospeicher des TI geschrieben werden sollen. Eventuelle folgende Daten belegen dann die folgenden Speicherstellen (>2E1,>2E2...). Was aber ist >2E0 ???

Vom Basic her wissen wir, daß der Bildschirm des TI-99/4a in 24 Zeilen und 32 Spalten aufgeteilt ist, insgesamt also 768 Kästchen, in denen jeweils ein bestimmtes Zeichen stehen kann. In der Maschinensprache des TI sind diese Kästchen nummeriert, und zwar beginend in der oberen linken Ecke mit der Zahl 0. Das Kästchen rechts daneben trägt die Zahl 1, das nächste die 2 und das übernächste die 3. Jenes unterhalb des ersten hat die Bezeichnung 32, darunter 64 und so weiter. Das Kästchen in der rechten unteren Ecke schließlich wird mit 767 bezeichnet.

In Assembler verwenden wir Hexadezimalzahlen. Zwar erlaubt LI auch die Benutzung von Dezimalzahlen. Sie sollten sich aber die Benutzung des 'anderen' Zahlensystems angewöhnen.

Wenn wir >2E0 in eine Dezimalzahl umwandeln, so erhalten wir die Zahl 736. Damit wird das erste Zeichen der 23. Bildschirmzeile angesprochen (vgl. obige Beschreibung).

LI R0,>2E0 lädt also zur weiteren Verarbeitung im Unterprogramm VMBW die Position auf dem Bildschirm, an der mit dem Beschreiben begonnen werden soll.

Doch es fehlen noch weitere Daten.

Wir benötigen wie bereits erwähnt eine Adresse, an der sich die zu schreibenden Daten zur Zeit befinden.

LI R1,SATZ lädt in Register 1 (R1) die Adresse, an der sich die Daten für den Schreibevorgang befinden. Da wir nicht wissen, wo der TI-Assembler die Daten ablegt, benutzen wir Pseudo-Adressen (=Label). Diese ersetzen für den Anwender die echte Adresse.

In Adresse SATZ befindet sich die Anweisung TEXT 'GUTEN TAG!'. R1 wird also später im Unterprogramm VMBW auf diese Nachricht zeigen und sie benutzen, um an die Stelle im Video-Speicher, den R0 bezeichnet, zu schreiben. Wir kopieren also einfach die Nachricht 'GUTEN TAG!' von Adresse START in den Video-Speicher, Adresse >2E0. Wenn das geschieht, wird der alte Inhalt von >2E0 vom Computer vergessen. Der Inhalt von Adresse START wir dadurch nicht verändert.

Zum Schluß benötigt der TI noch die Information, wieviele Zeichen er übertragen (kopieren) soll. 'GUTEN TAG!' hat, das Leerzeichen mitgezählt, 10 Character, die zu übertragen sind. Wir müssen also in R2 die Zahl 10 laden, um alle Informationen komplett zu haben. Der Befehl lautet also:

```
LI R2,10   oder
LI R2,>A   (denn >A ist Dezimal 10).
```

Damit enthalten die Register 0 bis 2 die

benötigten Werte. Nun müssen wir lediglich ins Unterprogramm VMBW verzweigen, um die besprochenen Resultate zu erhalten. Dies geschieht mit

```
BLWP §VMBW
```

und bedeutet Branch (Verzweige) and Link (verkette) Workspace Pointer (Arbeitsspeicher). Das Unterprogramm wird also unter Verwendung der genannten Register aufgerufen, wobei sich der Computer die Programmstelle, an der die Verzweigung stattfindet, merkt.

Das §-Zeichen vor VMBW besagt, daß es sich um eine Eingangs-Adresse des Assembler-Programs handelt; in diesem besonderen Fall um ein eingebautes Unterprogramm.

DEF, NOP, JMP und END wurden bereits erläutert.

Wenn Sie das Programm assemblieren und laufen lassen, wird der TI, beginnend mit der 1.Reihe der 23. Zeile des Bildschirms, die Nachricht 'GUTEN TAG!' ausdrucken.

Übungsaufgaben zum Kapitel 2:

1. Schreiben Sie mithilfe des Unterprogramms VMBW den Satz 'DRUECKEN SIE EINE TASTE' in die oberste Zeile des Bildschirms.

2. Versuchen Sie einmal, selbst ein Menü wie in der Editor-Auswahlliste gezeigt, auf den Bildschirm zu bringen. Farben brauchen Sie nicht zu ändern. Bezeichnen Sie die Programmeingangsstelle mit PROG2 und die FILES mit FILE1 bzw. OFILE1.

Die zweite Übungsaufgabe befindet sich auf der Diskette. Alle Aufgaben werden am Ende dieses Buches aufgelöst.

Nach TEXT
an EVEN denken wenn die
Zeichenkette ungerade ist.

1.3

3. Zeichen vom Bildschirm lesen

Sie wissen jetzt, wie man ein Zeichen oder eine ganzen Bildschirm ausdruckt und können sich, wenn Sie die Aufgabe 2 des vorangegangenen Kapitels gelöst haben, auch eine Vorstellung von der Arbeitsgeschwindigkeit machen. Am Ende dieses Kurses sollten Sie in der Lage sein, ein Programm in Maschinensprache zu schreiben, welches PRINT oder DISPLAY aus dem Basic ersetzt. Doch soweit ist es noch nicht. Kümmern wir uns erst noch einmal um den Bildschirm.

Wie Sie Zeichen darstellen können, wissen Sie jetzt. Wie aber verhält es sich mit dem Lesen von Zeichen? Im Basic machen wir dies mit INPUT oder ACCEPT bei manueller Eingabe oder GCHAR bei der selbständigen Abfrage.

In Assembler haben wir zwei eingebaute Unterprogramme, die in ihrer Art den bereits bekannten entsprechen:

VSBR - liest ein einzelnes Zeichen vom Bildschirm. Es ist also die Umkehrung von VSBW.

VMBR - liest eine Zeichenkette oder auch einen ganzen Bildschirm. Das Äquivalent dazu ist VMBW.

Ein Beispiel:

```
0001            DEF   PROG3
0002            REF   VMBW,VMBR
0003  SATZ       TEXT  'ICH LERNE ASSEMBLER!'
0004  BUFF1      BSS   >14
0005  PROG3      LI    R0,>0000
0006            LI    R1,SATZ
0007            LI    R2,>14
0008            BLWP  @VMBW
0009            LI    R1,BUFF1
0010            BLWP  @VMBR
0011            LI    R0,>0020
0012            LI    R2,>0009
0013            BLWP  @VMBW
0014  LOOP      NOP
0015            JMP   LOOP
0016            NOP END
```

Dieses Programm schreibt, wie Sie wissen, den Satz 'ICH LERNE ASSEMBLER' in die erste Zeile des Bildschirms. Danach liest es diesen Satz vom Bildschirm in einen Buffer, den wir mit dem BSS (Begin Block with Symbol) Befehl gebildet haben. BUFF1 ist ein 20 Byte Buffer und nimmt genau den Satz auf. Wäre der Buffer größer, so käme der Satz in die ersten 20 Bufferstellen. Wäre der Buffer zu klein, so würden nachfolgende (eventuell Programm-) Teile gelöscht, und es könnte zu einem Systemabsturz kommen.

VMBR benutzt die Register folgendermaßen:

0 - Die VDP-Stelle, von der gelesen werden soll.

1 - Eine Adresse, in die der gelesene Wert geschrieben werden soll.

2 - Die Anzahl der zu lesenden Bytes

(Zeichenzahl).

BUFFER1 hat jetzt folgenden Inhalt:

```
BUFFER1 +1 +2 +3 +4 +5 +6 +7  u.s.w.
>     49 43 48 20 4C 45 52 4E  ...
=      I  C  H     L  E  R  N  ...
```

Sie werden später noch sehen, wie Sie innerhalb eines Buffers bestimmte Stellen ansprechen können.

Zurück zu unserem kleinen Programm: Nachdem die Zeichen aus der ersten Zeile in den Buffer kopiert sind, können wir diese natürlich in unserem nächsten Schreibebefehl verwenden, indem wir die Eingangsstelle des Buffers BUFF1 als Hinweisadresse verwenden, wo die zu schreibenden Zeichen stehen. Der zweite VMBW-Aufruf wird also die ersten 9 Zeichen aus unserem Buffer in die zweite Zeile des Bildschirms schreiben.

Wir ändern natürlich immer nur die Register um, die wir für den Unterprogramm-Aufruf benötigen. So steht beim zweiten Aufruf von VMBW in Register 1 immer noch BUFFER1, so daß wir dieses Register nicht neu zu laden brauchen.

Den Abschluß unseres Programms bildet wie immer die Endlosschleife und der END-Befehl.

Wenn Sie ein einzelnes Zeichen lesen wollen, ist VSBR besser geeignet. Die Registerbelegung ist wie folgt:

0 - Adresse im VDP-Ram, die gelesen werden soll.

1 - Hier steht anschließend der gelesene Wert.

Bevor Sie VSBR benutzen, sollten Sie Register 1 löschen. Sie können dies mit LI R1,>0000 tun, aber es gibt auch eine elegantere Möglichkeit:

CLR R1 löscht Register 1. Merken Sie sich diesen Befehl!

Übungsaufgaben:

1. Schreiben Sie ein möglichst kurzes Programm, welches das Wort 'HALLELUJA' zehnmal untereinander schreibt. Kopieren Sie dann darunter zehnmal das Wort 'HALLE'.

2. Versuchen Sie einmal, ein Laufschriftprogramm zu schreiben. Benutzen Sie das Wort 'HAGERA' und lassen sie es von rechts nach links über die Bildschirmmitte laufen. Benutzen Sie dabei die bekannten Befehle. Programmieren Sie so, daß das Programm immer weiter läuft und verzichten Sie auf die Endlossschleife LOOP.

Aufgabe 2 befindet sich auf der Diskette. Zu allen Aufgaben finden Sie die Lösungen am Ende dieses Buches.

1.4

4. Definieren von Sonderzeichen

Sie können jetzt schon etwas auf den Bildschirm schreiben und auch davon lesen. Wenn Sie die letzte Aufgabe des vorangegangenen Kapitels gelöst haben, müssten Sie auch Zeichen auf dem Bildschirm bewegen können. Für viele Programme ist es aber auch erforderlich, seine eigenen Zeichen definieren zu können. In Basic geht das, wie Sie warscheinlich wissen, sehr einfach.

```
CALL CHAR(42,"FF818181818181FF")
```

verwandelt das Sternchen (*), also den ASCII-Character Nr. 42, in ein kleines Quadrat. Um dieses Problem in Assembler zu lösen, bedarf es wieder eines Blickes hinter die Kulissen, nämlich in den Speicher.

Bisher kennen wir im Video-Speicher, dem VDP-RAM, den Bereich >0000 bis >02FF. Er steht uns für die Bildschirmausgabe zur Verfügung. Es gibt jedoch andere Speicherbereiche mit anderen Aufgaben. Für die Charactermuster benutzt der TI wieder einen bestimmten Speicherbereich im VDP-RAM, dem Video-Speicher. Beginnend bei Adresse >0800 befinden sich hier die Muster der verschiedenen ASCII-Zeichen, wobei die Standartcharacter (ASCII 32-95) die Adressen >0900 bis >0AFF belegen. Jedes Muster benötigt 8 Bytes Speicherraum (also 4 2-Byte Worte), wie wir im Basic-Beispiel (FF181818181818FF) sehen.

Wollen Sie ASCII-Character 42 verändern, muß der entsprechende Speicherbereich, der die Daten für das Muster enthält, geändert werden. Wo aber befinden sich die Daten für ASCII-42?

Rechnen wir nach: Character 0 belegt, da der Bereich für diese Daten ja bei >0800 beginnt, die Speicherstellen >0800 bis >0807; der nächste >0808 bis >080F und der übernächste die Adressen >0810 bis >0817. Wenn Sie diese Tabelle bis Character 42 erweitern, können Sie sehen, daß das Muster dieses ASCII-Zeichens die Speicherstellen >0950 bis >0957 Belegen muß, also:

```
>0950 >FF          >0954 >18
>0951 >18          >0955 >18
>0952 >18          >0956 >18
>0953 >18          >0957 >FF
```

Das müssen wir dem Computer mitteilen. Es ist fast schon selbstverständlich, daß dafür unser Universal-Unterprogramm VMBW herhalten muß. Bevor Sie weiterlesen, überlegen Sie selbst einmal, wie VMBW die Daten wohl verarbeitet, wenn es um die Definition von Sonderzeichen geht.

Richtig???

```
0001            DEF   QUADER
0002            REF   VMBW
0003 KETTE      TEXT  '*-*-*-*-*-*'
0004 PATT       BYTE  >FF,>18,>18,>18
0005            BYTE  >18,>18,>18,>FF
0006 QUADER     LI    R0,>0950
0007            LI    R1,PATT
0008            LI    R2,8
```

```
0009            BLWP  §VMBW
0010 PRINT      LI    R0,>0000
0011            LI    R1,KETTE
0012            LI    R2,>B
0013            BLWP  §VMBW
0014 LOOP       NOP
0015            JMP   LOOP
0016            END
```

Erklärung:

In Zeile 0001 definieren wir unser Programm als 'QUADER' und stellen in Zeile 0002 den Bezug zu unserem Unterprogramm her. In Zeile 0003 haben wir wieder die uns schon bekannte TEXT-Anweisung.

Zeile 0004 und 0005 beinhalten die Daten für unseren Quader als Byte-Daten. Entsprechend zu TEXT und DATA haben wir den Befehl BYTE, um einzelne Bytes als Konstante in eine Speicherstelle zu schreiben. Hintereinander steht ab Speicherstelle PATT:

FF181818181818FF.

Bei QUADER startet das eigentliche Programm. Es werden zunächst die 8 Bytes aus PATT in den VDP-Speicher kopiert, und zwar ab Speicherstelle >0950, wo sich normalerweise das Sternchen befindet.

Danach erfolgt der Ausdruck im Teilprogramm PRINT in die obere linke Ecke des Bildschirms. Den Abschluß mit LOOP und END kennen Sie bereits.

Assemblieren sie jetzt das Programm und lassen Sie es ablaufen. Wenn Sie alles richtig eingegeben haben, müsste das Programm in die linke obere Ecke des Bildschirms sechs Quadrate drucken, verbunden durch fünf Bindestiche.

Falls nicht, haben Sie leider etwas falsch gemacht und sollten sich Kapitel I.1 und I.2 nocheinmal genau ansehen.

ÜBUNGSAUFGABEN

1. Sie haben bestimmt schon einmal in Basic einen Charactersatz neu definiert. Nehmen Sie die dort entwickelten Hexadezimalcodes der Charactermuster, definieren Sie die Zeichen in Assembler genauso um und schreiben Sie den Charactersatz auf den Bildschirm.

2. Versuchen Sie einmal, ein Programm zu schreiben, welches die Character 160 bis 255 auf den Bildschirm bringt, nachdem Sie diese umgeformt haben.

Beide Lösungen sind auf der Diskette enthalten. Die Lösungen befinden sich auch am Ende dieses Buches.

1.5

5. Farben bringen Leben

In diesem Kapitel wollen wir uns mit den Farbmöglichkeiten des TI beschäftigen. Wie Sie wissen, verfügt der TI-99/4a über 16 Farben, die er einmal als Hintergrundfarbe für den Bildschirm und dann als Vorder- oder Hintergrundfarbe für die Character-Sets verwenden kann, wobei ein Character-Set immer 8 aufeinanderfolgende ASCII-Codes beinhaltet.

In Basic sind die Farben von 1 bis 16 durchnummeriert. Die Farbwahl geschieht dort durch zwei Unterprogramme:

CALL SCREEN(f) stellt die Bildschirmfarbe auf f ein. Ist f=2, so ist der Bildschirm schwarz.

CALL COLOR(c,vf,hf) wählt für den Zeichensatz c die Vordergrundfarbe vf und die Hintergrundfarbe hf aus.

In Assembler sind die Farben von >0 bis >F durchnummeriert. Folgende Tabelle liefert einen direkten Vergleich:

Farbtabelle:

Farbe	Basic	Assembler
transparent	1	>0
schwarz	2	>1
mittelgrün	3	>2
hellgrün	4	>3
dunkelblau	5	>4
hellblau	6	>5
dunkelrot	7	>6
kornblumenblau	8	>7
mittelrot	9	>8
hellrot	10	>9
dunkelgelb	11	>A
hellgelb	12	>B
dunkelgrün	13	>C
magentarot	14	>D
grau	15	>E
weiß	16	>F

Beschäftigen wir uns zunächst mit der Bildschimfarbe. Wie Sie bereits wissen, verfügt der Accu des TI über eine Reihe von Registern, die verschiedene Aufgaben übernehmen können. In unseren vorangegangenen Beispielen lieferten die Register 0 bis 2 die Parameter für das Unterprogramm VMBW.

Außer diesen 'RAM'-Registern, die beliebig verändert und gelesen werden können, verfügt der TI aber auch über spezielle VDP-Register, die Sie zwar verändern, aber nicht lesen können. Diese Register haben unterschiedliche, feste Aufgaben und sind im Assembler-Manual auführlich erläutert. Uns soll es an dieser Stelle nur um unsere Bildschirmfarbe gehen.

Für die Bildschirmfarbe ist VDP-Register Nr. 7 zuständig. Es handelt sich dabei um ein 8-Bit-Register; hat also 8 Speicherstellen, die je eine 1 oder eine 0 beinhalten können. Jeweils die ersten vier und die letzten vier Bits fassen wir zu einem 'Nybble' zusammen. Jedes Nybble kann genau 16 Zustände ausdrücken, je nachdem, welche Bits ein- und welche ausgeschaltet sind. Damit lässt sich der Wert ausgezeichnet in eine Hexadezimalzahl umwandeln.

Enthält VDP-Register 7 beispielsweise die Binärzahl '00000111', so können wir diese aufteilen in '0000' und '0111' oder an deren Stelle zwei Hex-Zahlen schreiben, nämlich >0 und >7. Der Inhalt von VDP-R7 ist also >07.

Was bedeutet das nun für unsere Bildschirmfarbe?

Die Bildschirmfarbe des TI wird mit den Bits 4-7 des VDP-R7 festgelegt. Die Bits 0-3 sollen uns hier nicht interessieren. Sie haben eine andere Aufgabe, deren Erläuterung die Aufgaben dieses Einleitungskurses aber sprengen würde. Sie sollten auf '0000' stehen.

Die Bits 4-7 des VDP-R7 enthalten den Wert '0111' oder '>7'. Nach unserer Farb-Tabelle steht diese Zahl für kornblumenblau. Wollen wir die Farbe ändern, so müssen wir einen neuen Wert in VDP-R7 schreiben. Dies geschieht mit einem weiteren Unterprogramm:

VWTR bedeutet Video WriTe only Register.

Sie ahnen warscheinlich bereits, wie es nun weitergeht.

Das Unterprogramm benötigt die Werte, die es verarbeiten soll, wieder an Bestimmten Adressen. Die Werte, die VWTR verarbeitet, befinden sich in RAM-Register 0 (R0), welches im Gegensatz zu den VDP-Registernein 16-Bit Register ist und damit doppelt soviel Informationen beinhalten kann wie VDP-R7.

Sie können also, wie auch schon aus den voranggangenen Kapiteln bekannt, vier Hex-Zahlen in R0 laden. Um R0 auf die Benutzung des Unterprogramms VWTR vorzubereiten, muß es wie folgt programmiert werden:

Bits 0-7: Nummer des VDP-Registers, welches Sie ansprechen wollen. In diesem Fall also >07 für VDP-Register 7.

Bits 8-15: Wert, der übertragen werden soll. In diesem Fall einen Farbcode aus der bekannten Farbtabelle, zum Beispiel >C für dunkelgrün.

Um Dunkelgrün als Bildschirmfarbe zu erhalten, müssen wir in Assembler also folgendes programmieren:

```
0001          DEF  COLOR
0002          REF  VWTR
0003 COLOR    LI   R0,>070C
0004          BLWP @VWTR
0005 LOOP     NOP
0006          JMP  LOOP
0007          END
```

Noch einmal:
Um die Bildschirmfarbe zu verändern, belegt man Register 0 mit der Nummer des anzusprechenden VDP-Registers (<07) und der Farbbezeichnung. Danach wird das Unterprogramm VWTR mit der BLWP-Instruktion aufgerufen.

Übungsaufgaben:

1. Ändern Sie das Programm so ab, daß die Bildschirmfarbe in hellrot wechselt.

2. Schreiben Sie ein Programm, welches die Farbe ständig zwischen rot und grün wechselt und achten Sie dann auf den Bildschirmeffekt!

Die Lösungen zu den Listings finden Sie am Ende dieses Buches. Aufgabe 2 befindet sich auch auf der beiliegenden Diskette.

Wenn Sie die Farben für Character-Sets ändern wollen, wird die Sache wieder ein wenig komplizierter. Sie benötigen dazu wieder eines der Unterprogramme VSBW oder VMBW, dessen Funktionsweisen inzwischen bekannt sein dürften. Noch einmal zur Erinnerung: Wenn Sie VMBW benutzen, muß R0 die zu beschreibende Adresse im VDP-Speicher beinhalten, R1 die Adresse, an der sich die zu schreibenden Daten befinden und in R2 die Anzahl der Daten. Bei VSBW beinhaltet R1 den zu schreibenden Character, R2 ist ohne Bedeutung.

Der TI-99/4a speichert die Vorder- und Hintergrundfarben der Character-Sets im Bereich >0380 bis >039F. Das sind genau 32 Speicherstellen für 16 Charactersets mit je einer Vorder- und Hintergrundfarbe.

Wie sie vielleicht bereits wissen, verfügen wir in Assembler über weitaus mehr Character als in Basic. Begonnen wird mit Charactersatz 0 an Speicherstelle >0380 und >0381. Die Charactersets sind in Assembler jedoch anders organisiert als in Basic. Auf der nächsten Seite finden Sie eine übersicht über die Speicherbelegung der Farben.

Speicherbelegung der Farben:

Basic	Assem.		ASCII-Codes / bzw			Adr.
	0		>0 - >7	/ 0 - 7		>0380
	1		>8 - >F	/ 8 - 15		>0381
	2		>10- >17	/ 16- 23		>0382
	3		>18- >1F	/ 24- 31		>0383
1	4		>20- >27	/ 32- 39		>0384
2	5		>28- >2F	/ 40- 47		>0385
3	6		>30- >37	/ 48- 55		>0386
4	7		>38- >3F	/ 56- 63		>0387
5	8		>40- >47	/ 64- 71		>0388
6	9		>48- >4F	/ 72- 79		>0389
7	10	>A	>50- >57	/ 80- 87		>038A
8	11	>B	>58- >5F	/ 88- 95		>038B
9	12	>C	>60- >67	/ 96-103		>038C
10	13	>D	>68- >6F	/104-111		>038D
11	14	>E	>70- >77	/112-119		>038E
12	15	>F	>78- >7F	/120-127		>038F
13	16	>10	>80- >87	/128-135		>0390
14	17	>11	>88- >8F	/136-143		>0391
15	18	>12	>90- >97	/144-151		>0392
16	19	>13	>98- >9F	/152-159		>0393
	20	>14	>A0- >A7	/160-167		>0394
	21	>15	>A8- >AF	/168-175		>0395
	22	>16	>B0- >B7	/176-183		>0396
	23	>17	>B8- >BF	/184-191		>0397
	24	>18	>C0- >C7	/192-199		>0398
	25	>19	>C8- >CF	/200-207		>0399
	26	>1A	>D0- >D7	/208-215		>039A
	27	>1B	>D8- >DF	/216-223		>039B
	28	>1C	>E0- >E7	/224-231		>039C
	29	>1D	>E8- >EF	/232-239		>039D
	30	>1E	>F0- >F7	/240-247		>039E
	31	>1F	>F8- >FF	/248-255		>039F

Um beispielsweise die Farbe der Buchstaben H bis O auf dunkelblau zu setzen, müssen wir folgendes tun:

a) Welcher Character-Satz ist angesprochen?
b) Welcher Speicherbereich muß geändert werden?
c) Welche Farbnummer muß in diesen Speicherbereich geschrieben werden?

Die Buchstaben H bis O umfassen die ASCII-Character 72 bis 79 (>48->4F). Die Vorder- und Hintergrundfarbe wird demnach laut unserer Tabelle in Byte >0389 gespeichert. Um nun die Vordergrundfarbe (= die Farbe der Buchstaben) auf dunkelblau zu setzen, müssen wir in >0389 den Wert >41 laden. >4 setzt die Farbe der Buchstaben, >1 den Hintergrund (hier schwarz).

Für die Farbgebung dient, wie bereits erwähnt, unser VSBW-Unterprogramm:

```
0001          DEF   COLOR1
0002          REF   VSBW
0003 COLOR1   LI    R0,>0389
0004          LI    R1,>4100
0005          BLWP  @VSBW
0006 LOOP     NOP
0007          JMP   LOOP
0008          END
```

Das Ergebnis können Sie noch nicht sehen. Lesen Sie erst einmal weiter.

Wenn wir mehrere Farbcodes übertragen wollen, können wir natürlich VMBW benutzen, dessen Funktionsweise Ihnen ja

inzwischen bekannt ist.

Jetzt möchten wir die Farbveränderung sichtbar machen. Wir ändern das Programm so, daß es das Wort 'MILLION' in den neuen Farben auf den Bildschirm schreibt.

Unser Programm sähe dann folgendermaßen aus:

```
0001          DEF  COLOR1
0002          REF  VSBW,VMBW
0003 WORT     TEXT 'MILLION'
0004 COLOR1   LI   R0,>0389
0005          LI   R1,>4100
0006          BLWP @VSBW
0007          LI   R0,>018B
0008          LI   R1,WORT
0009          LI   R2,>7
0010          BLWP @VMBW
0011 LOOP     NOP
0012          JMP  LOOP
0013          END
```

Versuchen Sie einmal, die einzelnen Zeilen des Programms zu erklären, bevor Sie weiterlesen. Dann Vergleichen Sie:

01 Programm-Eintrittsstelle
02 Bezug auf die Unterprogramme.
03 Text-Daten für Schrift.
04 Adresse für Farbe laden.
05 Farbkombination laden.
06 Screenfarbe an VDP übergeben.
07 Bildschirmposition laden.
08 Textanfangsstelle in R1 laden.
09 Wieviele Bytes schreiben?
10 Daten an VDP übergeben.
11 Beginn Endlosschleife.
12 Rücksprung.

13 Programmende.

Wir haben also wieder ein richtiges kleines Assemblerprogramm geschrieben. Probieren Sie doch anhand der folgenden Übungsaufgaben einfach einmal andere Farben etc. aus, damit Sie Übung im Umgang mit diesen Unterprogrammen bekommen. Sie sind nach unserer Meinung der Kern der TI-Assemblersprache, da mit ihnen fast alles geändert werden kann: Farben, Formen, Bildschirm und Musikuntermalung.

Den Umgang mit Sprites wollen wir in diesem Einleitungskurs nicht behandeln, da hier eine tiefere Kenntnis der Assemblersprache erforderlich ist, die dieser Kurs weder vermitteln soll und kann. Sprites benutzen aber ebenfalls diese Unterprogramme.

Übungsaufgaben

1. Schreiben Sie ein Programm, welches die Farben aller Charactersätze und die Bildschirmfarbe ändert; geben Sie gleichzeitig einen beliebigen Text auf dem Bildschirm aus.

2. Geben Sie allen Charactersätzen eine andere Farbe (Sätze für Buchstaben und Zahlen) und definieren Sie Zeichen in verschiedenen Charactersätzen um. Geben Sie diese zum gegenseitigen Vergleich auf dem Bildschirm aus.

3. Probieren Sie doch einmal mit verschiedenen Farben aus, was geschieht, wenn man die Bildschirmfarbe blitzschnell hintereinander wechselt.

4. Definieren Sie die Character so um, daß alle Farbworte aus dem Handbuch in der Farbe erscheinen, die sie beschreiben.

Beispiel: DUNKELBLAU in dunkelblau.
MAGENTA in Magenta

Die Lösungen zu den Aufgaben finden Sie am Ende dieses Buches. Aufgabe 4 befindet sich auch auf der beiliegenden Diskette.

1.6

6. Abfrage der Tastatur

Ein weiteres Unterprogramm möchten wir Ihnen an dieser Stelle zeigen, obwohl Sie einige der damit verbundenen, notwendigen Befehle erst später kennenlernen werden, nämlich im Kapitel Verzweigungen und Sprünge.

Es ist jedoch so wichtig, daß Sie es unbedingt in seiner Wirkungsweise kennen sollten.

```
0001          DEF   PROG5
0002          REF   VSBW,KSCAN
0003 PROG5    CLR   R5
0004          CLR   R1
0005          MOVB  R5,@>8374
0006 TASTE    BLWP  @KSCAN
0007          MOVB  @>837C,R5
0008          JEQ   TASTE
0009          MOVB  @>8375,R1
0010          LI    R0,>000F
0011          BLWP  @VSBW
0012          JMP   PROG5
0013          END
```

So Funktioniert das Programm:
Zunächst wir es initialisiert und die Verbindung zu den eingebauten Unterprogrammen hergestellt. KSCAN fragt im Beispiel oben die Tastatur ab, ob eine Taste gedrückt worden ist. Wenn keine Taste gedrückt wurde, steht in Adresse >837C eine Null. Zuvor müssen wir spezifizieren, daß wir mit KSCAN die

6. Abfrage der Tastatur

Ein weiteres Unterprogramm möchten wir Ihnen an dieser Stelle zeigen, obwohl Sie einige der damit verbundenen, notwendigen Befehle erst später kennenlernen werden, nämlich im Kapitel Verzweigungen und Sprünge.

Es ist jedoch so wichtig, daß Sie es unbedingt in seiner Wirkungsweise kennen sollten.

```
0001          DEF   PROG5
0002          REF   VSBW,KSCAN
0003 PROG5    CLR   R5
0004          CLR   R1
0005          MOVB  R5,@>8374
0006 TASTE    BLWP  @KSCAN
0007          MOVB  @>837C,R5
0008          JEQ   TASTE
0009          MOVB  @>8375,R1
0010          LI    R0,>000F
0011          BLWP  @VSBW
0012          JMP   PROG5
0013          END
```

So Funktioniert das Programm:
Zunächst wir es initialisiert und die Verbindung zu den eingebauten Unterprogrammen hergestellt. KSCAN fragt im Beispiel oben die Tastatur ab, ob eine Taste gedrückt worden ist. Wenn keine Taste gedrückt wurde, steht in Adresse >837C eine Null. Zuvor müssen wir spezifizieren, daß wir mit KSCAN die

Tastatur - und nicht die linke oder rechte Hälfte oder Joysticks - abfragen wollen. Dies tun wir, indem wir ein Byte >00 in Adresse >8374 schreiben. Mit JEQ testen wir, ob eine Taste gedrückt worden ist und falls nein, springt das Programm zur Tastaturabfrage zurück.

Wenn eine Taste gedrückt wurde, steht der ASCII-Code in Adresse >8375. Alle diese Adressen gehören zu einem bestimmten Speicherbereich, der CPU RAM PAD genannt wird. Die Bedeutung dieser Speicherstellen finden Sie im EA-Handbuch ab Seite 404.

MOVB überträgt ein einzelnes Byte von einer Speicherstelle zur anderen, zu oder von einem Register oder von Register zu Register. Es wird jeweils das höherwertige erste Byte des Wortes übertragen. Nehmen wir an, es wird Taste 'A' gedrückt, so befindet sich nach KSCAN in >8375 eine >41. Wir übertragen diese mit MOVB in Register 1, also R1=>4100.

MOVB verschiebt Bytes von OPERAND 1 in OPERAND 2.

Danach geben wir über VSBW das Zeichen an den Bildschirm weiter. Es wird etwa in der Mitte der obersten Zeile ausgedruckt.

Damit haben wir ein kleines Programm geschrieben, welches immer das Zeichen auf den Bildschirm bringt, welches auf der Tastatur gedrückt wird. Über MOVB und andere Befehle werden Sie später noch mehr erfahren.

Übungsaufgabe:

Schreiben Sie ein Programm, welches zu Beginn die Nachricht 'DRÜCKEN SIE EINE TASTE' anzeigt - ja, mit deutschem Umlaut... das können Sie!!! - und auf Tastendruck in Bildschirmmitte das Wort 'TASTE GEDRUECKT'.

Beim nächsten Tastendruck sollen beide Nachrichten verschwinden und der Bildschirm rot aufleuchten.

Wenn erneut eine Taste gedrückt wird, soll das Programm wieder von vorne starten.

Dieses Programm befindet sich auf der Diskette und als Listing am Ende dieses Buches!

1.7

7. Töne Programmieren

Neben seinen hervorragenden grafischen Eigenschaften verfügt der TI über Sound Prozessoren, die perlende Musikabläufe ermöglichen. Besonders bei der Programmierung von Spielen sollte auf 'guten Ton' Acht gegeben werden.

Bevor wir Ihnen zeigen, wie man in Assembler Töne programmiert, bringen wir wieder ein Beispiel aus dem Basic, um die Sache zu verdeutlichen. Wie programmiert man in Basic Töne oder einen einzelnen Ton?

```
10 FOR I=1 TO 10
20 CALL SOUND(200,10,0)
30 NEXT I
```

Diese Zeilen produzieren einen Ton. Wie nun erreichen wir dies in Assembler? Greifen wir dazu einfach einmal auf das Beispiel mit dem Print-Befehl zurück.

Sie haben gesehen, daß der TI, um den Bildschirm zu beschreiben, ganz bestimmte Daten an ganz bestimmten Stellen (Adressen) benötigt. Mit dem Sound verhält es sich nicht anders. Mit VMBW sprechen wir nämlich außer der Bildschirmausgabe, den Charactermuster und Farben auch den Musikprozessor an. Wie Sie vielleicht wissen, verfügt der TI über mehrere Tonprozessoren. Wir möchten uns auf einen beschränken, um den Einstieg zu erleichtern.

Wie gezeigt, werden z.B. beim Bildschirmausdruck die Adressen >0000 bis

>02FF Benutzt, um jeweils den Character abzulegen, der an einer bestimmten Position des Bildschirms ausgegeben werden soll. Für Töne benutzen wir die Adressen ab >1000. In unserem Beispiel sprechen wir nur einen Soundgenerator an der TI besitzt derer drei.

Um einen Ton zu produzieren, sind eine ganze Reihe von Informationen erforderlich. Benötigt werden Dauer, Frequenz, Lautstärke sowie die bereits vom 'Print' her bekannten Daten über Schreibeadresse, Tabelle mit den zu schreibenden Daten und Anzahl der zu schreibenden Daten.

Beginnen wir mit der Vorbereitung des Unterprogramms VMBW für die Sounderzeugung. Entsprechend der Bildschirmausgabe müssen wir wieder die Register 0 bis 2 mit bestimmten Daten laden. Erinnern Sie sich noch, wie wir das angestellt haben?

```
0001          DEF   MUSIK
0002          REF   VMBW
0003 KLANG    BYTE  >03,>8C,>1F,>91,>A
0004          BYTE  >03,>83,>15,>91,>F
0005          BYTE  >03,>8E,>0F,>91,>14
0006          BYTE  >03,>8A,>0C,>91,>19
0007          BYTE  >03,>8E,>0F,>91,>14
0008          BYTE  >03,>83,>15,>91,>F
0009          BYTE  >03,>8C,>1F,>91,>A
0010 MUSIK    LI    R0,>1000
0011          LI    R1,KLANG
0012          LI    R2,>23
0013          BLWP  @VMBW
  .
  .
```

Mit REF/DEF machen wir die bekannte Initialisierung unseres Programms. Danach folgt die Datenübetragung der KLANG-Bytes in das VDP-RAM ab Adresse 1000.

Die Sound-Erzeugung funktioniert also ganz ähnlich wie das Beschreiben des Bildschirms, nur das anstelle von Texten Bytes übertragen werden. Natürlich könnten auch Texte als Byte-Daten übertragen werden, aber diese Methode ist sehr umständlich und fehlerträchtig und daher nicht zu empfehlen.

Beschäftigen wir uns jetzt mit den Daten, die übertragen werden sollen.

Zunächst muß der Computer über die Art des Tones informiert werden. Da der TI mehrere Generatoren zur Sonderzeugung besitzt, müssen wir Töne in einer ganz bestimmten Form eingeben, damit der Computer weiß, auf welche Art er sie abzuspielen hat. Zum Beispiel ist

```
CALL SOUND(40,110,15)::CALL SOUND(40,440,0)
```

etwas ganz anderes als etwa

```
CALL SOUND(40,110,15,440,0).
```

Sie müssen dem Computer also sagen, wie lang ein Ton ist, d.h., wieviel Datenbytes er umfasst. Das erste Byte in einer Soundzeile gibt die Anzahl der zu

benutzenden Bytes an.

Als nächstes geben wir die Frequenz an, die gespiel werden soll. Die Frequenzdanten bestehen aus jeweils 2 Byte; Sie finden Sie im Handbuch zum Editor/Assembler ab Seite 318.

Beispiel >893F = Frequenz 110.

Das nächste Byte der Soundzeile bestimmt die Dämpfung. Eine Dämpfung liegt im Bereich von 0 bis 28 Dezibel und wir folgendermaßen berechnet:

Die ersten vier Bits im Dämpfungsbyte sind 1001, also >9, wenn Sie Sondgenerator 1 ansprechen.

Die zweiten vier Bits bestimmen die Dämpfung. Dabei werden die Bits 4-7 so behandelt, als hätten sie an 8. Stelle eine zusätzliche binäre 0 stehen. Wenn die Bits 4-7 also beispielsweise den Inhalt 0001 haben, wird dies als 00010 gerechnet und spezifiziert eine Dämpfung von 2 Dezibel. >91 spezifiziert also eine Dämpfung von 2db auf Soundgenerator Nummer 1.

Wollen Sie einen Generator abschalten, so sind die Bits 4-7 des Dämpfungsbyte auf 1 zu setzen. >9F im Dämpfungsbyte schaltet Generator 1 aus.

Das letzte Byte der Soundzeile bestimmt die Tondauer. Es wird bei der Angabe der

Anzahl der Sounddaten nicht mitgerechnet! Beachten Sie das obige Beispiel!

Die Tondauer berechnet sich in 60stel Sekunden. Sie liegt im Bereich von >00 bis >FF (ca. 4.25 Sekunden).

Noch ein Beispiel:

>03,>8A,>2F,>94,>78 bestimmt einen Ton der Frequenz 146 mit einer Dämpfung von 8 Dezibel auf Generatot 1. Der Ton hat eine Dauer von ca. 2 Sekunden (>78/60 = 120/60 = 2).

Wenn Sie mehrere Töne abspielen wollen, müssen mehrere dieser Soundzeilen programmiert werden. Diese ergeben eine Soundliste und müssen mittels VMBW ins VDP-Ram ab Adresse >1000 geladen werden. Als Beispiel soll unser Assembler-Programm (s.o.) dienen.

Um die Töne abzuspielen, müssen wir jedoch noch mehr tun, als nur die Daten zu laden. Der Sound unterliegt nämlich einer automatischen Ablaufkontrolle, die es uns ermöglicht, Sound gleichzeitig zum übrigen Geschehen auf dem Bildschirm ablaufen zu lassen. Erweitern Sie zunächst das Programm wie folgt:

```
0014 LOOP1  LIMI 0
0015        LI   R9,>0100
0016        LI   R10,>1000
0017        MOV  R10,@>83CC
0018        SOCB R9,@>83FD
```

```
0019          MOVB R9,@>83CE
0020          LIMI 2
0021 LOOP2    MOVB @>83CE,@>83CE
0022          JEQ  LOOP1
0023          JMP  LOOP2
```

Zunächst wieder ein paar neue Assembler Befehle:

LIMI steuert die VDP-Unterbrechung. Jede 60stel Sekunde macht der TI einen VDP-Interrupt, d.h., er durchläuft ein Kontrollprogramm zur Videosteuerung, auf welches Sie normalerweise keinen Einfluß haben. An diesen VDP-Interrupt ist auch die Tonerzeugung angeschlossen. LIMI steht für LOAD INTERRUPT MASK IMMEDIATE und ermöglicht es uns, diesen VDP-Interrupt zu unterdrücken. Für den oben genannten Programmteil LOOP1 muß der VDP-Interrupt unterdrückt sein. Solange dann später der Sound abläuft, muß ein VDP-Interrupt alle 60stel Sekunde ermöglicht werden. Der VDP-Interrupt kann Adressen im VDP-Bereich verändern, sodaß Interrupts immer dort liegen sollten, wo Sie selbst den VDP-Bereich nicht ansprechen. Eine Tastaturabfrage - wie im vorangegangenen Kapitel gezeigt - ist eine geeignete Stelle, um einen Interrupt zu ermöglichen.

Der Interrupt wird mit LIMI 0 unterdrückt und mit LIMI 2 wieder ermöglicht.

MOV und MOVB müssten Ihnen bekannt sein. Es werden damit Worte bzw. Bytes in einen anderen Speicher kopiert; sei es von und zu Register, von Speicherstelle

zu Speicherstelle, oder wie im obigen Fall von Register zu Speicherstelle (oder umgekehrt). Wenn man in die gleiche Adresse verschiebt, aus der die Daten stammen, wird nichts verändert. Der Inhalt der Adresse wird aber mit 0 im Statusbyte verglichen, sodaß eine 'JEQ'-Verzweigung möglich wird. Es ist eine Mögliche Art, etwas zu vergleichen.

SOCB vergleicht die Bytes zweier Operanden und setzt im zweiten Operanden diejenigen, welche in mindestens einem Operanden oder in beiden gesetzt sind. Der Befehl heißt daher auch SET ONES CORRESPOND., BYTE.

Beispiel für SOCB siehe Mnemonic-Tabellen.

Verwandte Befehle sind SOC und COC, aber auch SZCB, SZC und CZC (alle in der Mnemonic-Tabelle erläutert).

Zum Programm:
Zunächst schalten wir den VDP-Interrupt aus, um Daten übertragen zu können. Register 9 laden wir mit >0100, sodaß wir im Führungsbyte einen Wert >01 haben. Diesen brauchen wir später noch.

Register 10 laden wir mit der Hinweisadresse zur Soundliste, also mit >1000, und geben diesen Wert mit MOV an Adresse >83CC weiter. Adresse >83CC gehört zum CPU PAD RAM und muß für die automatische Ablaufkontrolle der Soundliste die Startadresse der

Soundtabelle beinhalten.

Danach setzen wir eine VDP-Markierung mittels SOCB in ein weiteres Byte des CPU PAD RAM, Adresse >83FD. Register 9 enthält dabei unser Byte >01, welches wir mit LI geladen haben.

Um die automatische Sounderzeugung zu starten, muß ein Byte >01 in Adresse >83CE geladen werden. >83CE verringert sich automatisch bei jeder VDP-Unterbrechung und kontrolliert so die zu spielenden Soundbytes. Wenn wir später >83CE zu 0 vergleichen, wissen wir, ob alle Bytes der Soundtabelle abgespielt wurden oder nicht.

Bevor wir unseren Sound-Ablauf kontrollieren, ermöglichen wir mit LIMI 2 den VDP-Interrupt.

Abschließend wird Byte >83CE auf Ende der Soundliste überprüft. Ist es erreicht, beginnt die Melodie von vorne. Ist es nicht erreicht, wird die nächste Soundzeile gespielt.

Wenn Sie das Programm vollständig eingeben und laufen lassen, wird der TI einige Töne abspielen.

Übungsaufgaben:

1. Versuchen Sie einmal selbst, einen Klang zu programmieren. Beispiel: Sirene, Big Ben, Glockenspiel.

2. Trauen Sie es sich zu, auch eine kleine Musik, zum Beispiel die Anfangsmelodie für ein Spiel, zu komponieren?

Ein Beispiel für Musikprogrammierung finden sie im Spielausschnitt von 'PARTISAN VILLAGE' auf der Diskette, welches auch als Listing in diesem Kurs enthalten ist.

1.8

8.Etwas Mathematik und Programmtechnik

Einige unter Ihnen, die vielleicht schon Kenntnisse in Assembler haben, werden uns warscheinlich anklagen, das Pferd am Schwanze aufgezäumt zu haben; fangen wir doch mit Bildschirmausgabe und Tonerzeugung an, wo andere gerade das binäre Einmaleins zu vermitteln versuchen.

Es ist pure Absicht von uns, daß wir Sie zuerst mit der Bildschirmausgabe vertraut gemacht haben und erst jetzt zu einigen grundlegenden Dingen kommen. Wer die Absicht hat, trotzdem erst Binärzahlen zu addieren, der kann im entsprechenden Kapitel nachlesen, da wir mit diesem Kurs praktische Tips und keine großartigen Theorien vermitteln wollen.

Selbstverständlich ist es dem TI auch möglich, in Assembler Rechnungen durchzuführen. Man kann auch Schleifen und Sprünge programmieren oder Funktionen und Unterprogramme. Wir setzen an dieser Stelle voraus, daß Sie die einführenden Kapitel über VDP-Programmierung mit ihren Unterprogrammen begriffen haben. Falls nicht, so empfehlen wir eine Wiederholung dieser Kapitel, da die nachfolgenden auf dem bereits Bekannten aufbauen ohne dieses erneut zu erläutern.

Wie Sie bemerkt haben, ist Assembler keine Programmiersprache, bei der wildes 'drauflos programmieren' zum Ziel führt.

Vielmehr ist es erforderlich, sich vorher genau Gedanken über die Leistungsfähigkeit eines Programms zu machen. Wenn wir uns an dieser Stelle auch noch nicht mit Programmentwicklung und Struktur befassen wollen, so möchten wir doch eine gewisse Ordnung in unsere Programme bringen, die sich nur mit strukturierter Programmierung erreichen lässt.

Ein bißchen Struktur brachten wir bereits in verschiedene Beispiele. Hier stellten wir zum Beispiel die Daten an den Anfang des Programms. Wir wollen uns auch in Zukunft an folgendes Schema halten, sofern dies programmtechnisch vertretbar ist.

a) Definieren der Startadressen (DEF).
b) Bezüge auf Unterprogramme (REF).
c) Sonstige Bezüge und Buffer (EQU).
d) Datenzeilen (DATA,BYTE,TEXT,etc).
e) Hauptprogramm
f) Eigene Unterprogramme

Zur Unterscheidung der einzelnen Teile können wir im TI-Assembler Kommentarzeilen einfügen. Als Kommentar gilt alles hinter einem Sternchen (*), es sei denn daß Sternchen steht zwischen den Anführungszeichen einer Textanweisung oder vor einer Registerbezeichnung (Sonderfunktion; vgl. 'EA-Handbuch'). Jedes Programm auf Diskette wurde auf diese Weise dokumentiert.

Beim TI-Assembler steht das Sternchen entweder anstelle eines LABELS oder hinter einer vollständigen Anweisung.

Wenn wir also in Zukunft programmieren, so wollen wir unsere Programme durchschaubarer machen, damit auch noch nach einigen Wochen ihre Funktion direkt erkennbar ist.

Die folgenden Kapitel sollen Ihnen weitere Programmierbefehle der Assemblersprache vermitteln. Alle werden von kurzen Beispielprogrammen begleitet sein, die Sie direkt am TI-Assembler ausprobieren können. Dabei nehmen wir in allen Fällen die bekannten VDP-Routinen zur Hilfe, um das Ergebnis visuell oder akustisch wahrnehmbar zu machen. Denn die meißten Assembler-Anweisungen verursachen 'lediglich' eine Veränderung von Speicherstellen, die man nicht erkennt. Erinnern Sie sich an Basic:

```
10 A=5
20 B=N*A-8
30 IF B<0 THEN 50
40 IF B>0 THEN 80
50 A=A-1
60 N=N+1
70 GOTO 20
80 END
```

Lassen wir dieses Programm laufen, so wird es sich nach kurzer Zeit mit 'READY' zurückmelden, ohne daß für uns sichtbar eine Veränderung stattgefunden hat. Erst wenn wir

```
25 PRINT A;B;N
```

einfügen, sehen wir, was das Programm bewirkt.

Um diesen Effekt in Assembler zu erreichen, haben wir die wichtigsten VDP-Unterprogramme vorweg erklärt und benutzen diese, um unsere Programmierergebnisse sichtbar zu machen.

Genauso, wie Ihnen der Basic-Befehl PRINT ein Begriff ist, sollten in Assembler die Unterprogramme VMBW, VSBW und VWTR von Ihnen verstanden worden sein.

1.9

9. Schleifen, Verzweigungen und Sprünge

Eine der wichtigsten Eigenschaften aller Programmiersprachen ist es, daß man den Programmablauf beeinflussen kann. Aus dem Basic kennen Sie die FOR-NEXT-Schleife, die IF-THEN-Abfrage und den GOTO-Sprung. In Assembler haben wir ähnliche Anweisungen, die allerdings ein bißchen anders arbeiten.

Beginnen wir mit dem unbedingten Sprung, der einer GOTO-Anweisung beim Basic entspricht. Wir haben diesen Befehl bereits kennengelernt. Er lautet JMP.

JMP verzweigt immer zu der genannten Adresse, im TI-Assembler auch ein Ausdruck, der eine Adresse bezeichnet.

JMP ENDE verzweigt also im TI-Assembler zu einer Eingangsstelle, die mit ENDE bezeichnet ist. Der Vergleich mit Basic:

```
10 A=5
20 GOTO 40
30 A=3
40 END
```

```
0001          DEF   JUMP
0002 *
0003 JUMP     LI    R6,>41
0004          JMP   STOP
0005          LI    R6,>42
0006 STOP     NOP
0007          JMP   STOP
0008          END
```

Wichtig ist der unbedingte Sprung nach STOP, wenn einer Speicherstelle ein Wert

zugewiesen worden ist, bevor sich der Wert erneut ändert. Das Programm könnte so beschrieben werden:

a) Wertzuweisung
b) Sprung zum Ende
c) Wertzuweisung
d) Ende

Die zweite Wertzuweisung soll nicht stattfinden. Deshalb JMP STOP.

Der Befehl wird oft benötigt, um Programmteile zu trennen und diese gegenseitig anzuspringen. Wenn die Entfernung zur Sprungadresse größer als >100 ist, müssen Sie statt JMP den Befehl BRANCH verwenden.

```
0004          B     §STOP
```

Wichtiger als JMP sind jedoch die Befehle für einen bedingten Sprung, die ähnlich wie die IF-THEN-Anweisung im Basic funktionieren.

Der TI-Assembler verfügt über eine ganze Reihe von bedingten Sprungbefehlen. Eine kürzere Erläuterung finden Sie im Kapitel mit den Mnemotechnischen Abkürzungen.

Hier die Beschreibung aller Sprungbefehle:

JEQ Jump if Equal
Sprung wenn Gleich

JNE Jump if not Equal
Sprung wenn ungleich

JGT Jump if Greater Than
Sprung wenn größer als

JLT Jump if Less Than
Sprung wenn kleiner als

JHE Jump if Higher or Equal
Sprung wenn höher oder gleich

JH Jump if logical High
Sprung wenn logisch höher

JL Jump if logical Less
Sprung wenn logisch niedriger

JLE Jump if logical Less or Equal
Sprung wenn logisch niedriger
oder gleich

JNC Jump if no Carry
Sprung wenn kein Übertrag

JOC Jump on Carry
Sprung wenn Übertrag

JNO Jump if no Overflow
Sprung wenn kein Überlauf

JOP Jump on Odd Parity
Sprung wenn Parität ungerade

Es ist natürlich zu fragen, was der TI-Assembler denn nun kontrolliert, bevor er einen bedingten Sprung durchführt, beispielsweise JEQ START.

Der TI verfügt, wie Sie bereits wissen, über verschiedene Register, die bestimmte Aufgaben erfüllen sollen. Zwei Gruppen haben Sie kennengelernt:

a) Workspace-Register: Mit ihnen können Sie in Ihren Programmen arbeiten; z.B. auch Parameter an ein Unterprogramm (VMBW, etc.) weitergeben. Sie können gelesen und beschrieben werden. Sie werden noch einige Assemblerbefehle kennenlernen, mit denen Sie Registerinhalte ändern können.

b) VDP-Register: Sie enthalten Informationen für den Bildschirmspeicher. VDP-Register können nur beschrieben werden (VWTR). Denken Sie an das Beispiel mit der Bildschirmfarbe.

Für die Verzweige-Operationen lernen Sie jetzt ein weiteres wichtiges Register kennen:

c) Das Status-Register: Es kann nicht direkt beschrieben werden, sondern enthält immer Informationen über eine durchgeführte Operation. Fast jede Assembler-Anweisung verändert oder benutzt das Status-Register.

Das Status-Register ist ein 16-Bit-Register, entspricht also der größe nach einem Workspace-Register (WR). Von den 16 Bits, die es für verschiedene

Informationen benutzt, intressieren uns hier lediglich die höchstwertigen 6, also Bits 0-5. Diese Bits tragen besondere Bezeichnungen wegen der Informationen, die sie enthalten.

Bit 0: L> oder Logisch Größer Bit.
Bit 1: A> oder Arithmetisch größer Bit.
Bit 2: EQ oder Equal Bit.
Bit 3: C oder Carry (Übertrags-) Bit.
Bit 4: OV oder Overflow (Überfluß-) Bit.
Bit 5: OP oder Odd Parity (Ungerades Paritäts-) Bit.

Verglichen wird nach jeder Operation das Ergebnis der Operation mit >0. Hier ein Beispiel:

Die LI-Instruktion verändert den Inhalt des Statusregisters. Welche Bits können überhaupt verändert werden. Erinnern Sie sich: LI lädt einen Wert in ein Register.

L> wird beeinflußt, denn der Operand der LI-Instruktion wird mit >0 verglichen. Lautet die Operation z.B. LI R3,>5, so wird das L>-Bit gesetzt, da 5 logisch größer Null ist.

Logisch größer heißt, daß beim Bit-für-Bit Vergleich des LI-Operanden dessen Wert ohne Berücksichtigung des ersten (führenden) Bits als Vorzeichen größer war als eine binäre 0.

In diesem Beispiel wird L> gesetzt.

A> wird ebenfalls beeinflußt, denn >5 ist tatsächlich größer als >0. Das heißt, das Führungsbit wird als Vorzeichen

gewertet. Ist es gesetzt, so gilt die Zahl als negativ. Im Unterschied zu L> bedeutet dies, daß zum Beispiel >F3 logisch größer, aber arithmetisch kleiner (als Null) ist. (vgl. hierzu auch Kapitel 'Binärzahlen').

Im obigen Beispiel wird A> gesetzt.

EQ wird ebenfalls beeinflußt, da es ja möglich ist, daß der LI-Operand gleich >0 ist. Falls ja, wird das EQ-Bit gesetzt, falls nein, wird es (auf BIN(0)) zurückgesetzt.

Im Beispiel wird EQ zurückgesetzt, da der LI-Operand nicht gleich >0 ist.

C wird nicht beeinflußt, da ein Bit-Übertrag beim Laden von Daten nicht stattfindet. Ein Übertrag kann nur bei Rechenoperationen oder Verschiebungen (siehe Kapitel Vergleiche) erfolgen.

C behält bei LI seinen alten Wert bei.

OV wird ebenfalls nicht beeinflußt, da kein Überfluß stattfinden kann; ein Überfluß entsteht nur bei einer Rechenorperation, wenn der Speicherbereich der Speicherstelle überschitten wird, nicht aber beim einfachen laden von Direktdaten.

OV behält ebenfalls seinen alten Wert bei.

OP wird nicht gesetzt. ODD PARITY testet die Anzahl der Eins-Bits. Ist sie ungerade, wird OP gesetzt.

Programmieren wir:

```
0001          DEF   PROG5
0002          REF   VSBW
0003 PROG5    LI    R5,>7
0004          LI    R0,>0000
0005          LI    R1,>5A00
0006 NEXT     BLWP  §VSBW
0007          INC   R0
0008          DEC   R1
0009          DEC   R5
0010          JEQ   LOOP
0011          JMP   NEXT
0012 LOOP     NOP
0013          JMP   LOOP
0014          END
```

Zunächst zu den neuen Programmierbefehlen, DEC und INC. Sie werden Ihnen bei den folgenden Beispielen noch öfters begegnen.

Mit DEC vermindern Sie den Inhalt eines Registers oder einer Speicherstelle um 1. Also, wenn R5=>7 ist, so vermindert sich der Wert nach DEC R1 in >6. Wenn §ZAHL=>10 ist, so vermindert sich der Wert nach DEC §ZAHL auf >0F.

Mit INC verhält es sich umgekehrt. INCrement heißt hochzählen und addiert 1 zum augenblicklichen Wert. Wenn R0=>0 ist, so enthält R1 nach INC R1 den Wert >1. Ist §ZAHL=>10, so ist es nach INC R1 = >11.

Zwei weitere Befehle, die hier noch nicht auftauchen, ermöglichen demgemäß das addieren bzw. subtrahieren von 2. Die Befehle lauten dann DECT und INCT

(DECrement und INCrement by Two).

Das Programm arbeitet wie folgt:

Zunächst wird in R5 eine >7 geladen. Danach bereiten wir R0 und R1 für die Videoausgabe vor. Als erstes wird der Buchstabe 'Z' in die obere linke Ecke geschrieben. Danach werden folgende Werte geändert:

R0 wird 1 größer und zeigt auf Bildschirmstelle 1.

R1 wird 1 kleiner und beinhaltet jetzt den Buchstaben 'Y'.

R5 wird 1 kleiner und beinhaltet jetzt den Wert >6.

Sobald der Inhalt von R5 gleich >0 ist, springt danach die Programmkontrolle zu LOOP (daher JEQ LOOP), andernfalls zurück zu NEXT, wo der nächste Buchstabe ausgegeben wird.

Das Programm ist also eine Schleife, die wir in etwa mit folgendem Basic-Programm vergleichen können:

```
100 FOR I=90 TO 81 STEP -1
102 CALL HCHAR(1,91-I,I)
104 NEXT I
106 END
```

Wir wollen uns noch ein wenig mit den bedingten Sprüngen beschäftigen.

Gelöst werden soll folgendes Problem: Der Bildschirm soll in sichtbaren Abständen

die Farbe wechseln und die Farbe als Wort in der oberen linken Bildschirmecke anzeigen.

Programmstruktur:

a) Bezug zu VWTR,VMBW,VSBW
b) Textdaten
c) Vorbereitung, Startprogramm
d) Farben und Text anzeigen
e) Pause
f) Farben und Text ändern
g) Wenn nicht alle Farben, nächste!
h) Neustart des Programms.

Verwenden wollen wir den Sprungbefehl JNE. Er verursacht einen Sprung, wenn im Statusbyte das EQ-Bit nicht gesetzt ist.

```
0001          DEF   PROG6
0002          REF   VWTR,VMBW,VSBW
0003 *
0004 FARBEN   TEXT  'TRANSPARENT    '
0005          TEXT  'SCHWARZ        '
0006          TEXT  'MITTELGRUEN    '
0007          TEXT  'HELLGRUEN      '
0008          TEXT  'DUNKELBLAU     '
0009          TEXT  'HELLBLAU       '
0010          TEXT  'DUNKELROT      '
0011          TEXT  'KORNBLUMENBLAU'
0012          TEXT  'MITTELROT      '
0013          TEXT  'HELLROT        '
0014          TEXT  'DUNKELGELB     '
0015          TEXT  'HELLGELB       '
0016          TEXT  'DUNKELGRUEN    '
0017          TEXT  'MAGENTAROT     '
0018          TEXT  'GRAU           '
0019          TEXT  'WEISS          '
0020 *
```

```
0021 PROG6  LI   R0,>0384
0022        LI   R1,>1F00
0023        LI   R5,>E
0024 NEXT1  BLWP @VSBW
0025        INC  R0
0026        DEC  R5
0027        JNE  NEXT1
0028 *
0029        LI   R6,>0700
0030        LI   R7,FARBEN
0031        LI   R8,>000F
0032 NEXT2  MOV  R6,R0
0033        BLWP @VWTR
0034        LI   R0,>0000
0035        MOV  R7,R1
0036        LI   R2,>E
0037        BLWP @VMBW
0038        LI   R5,>000E
0039 NEXT3  INC  R7
0040        DEC  R5
0041        JNE  NEXT3
0042        INC  R6
0043        LI   R9,>FFFF
0044 NEXT4  DEC  R9
0045        JNE  NEXT4
0046        DEC  R8
0047        JNE  NEXT2
0048
0049        JMP  PROG6
0050 *
0051        END
```

Zunächst zum Befehl MOV. MOV verschiebt - im Gegensatz zu MOVB - ein 2-Byte-Wort, also 16 Bits auf einmal.

Mit MOV können also Speicherinhalte oder Registerinhalte kopiert werden. Verschoben wird von OPERAND 1 zum OPERAND 2.

Dies ist das erste komplexe Assembler Programm, welches Sie kennenlernen. Versuchen Sie einmal selbst herauszufinden, was in jeder Zeile geschieht und lesen Sie dann erst weiter.

1. Programmteil:
Hier stehen wieder die Definitionen REF/DEF.

2. Programmteil:
Text-Daten für die Wortanzeige.

3. Programmteil:
Laden der Farben schwarz/weiß für die Standartcharacter mithilfe einer Schleife:

```
0021 Farbspeicheradresse laden
0022 Farbcodes laden
0023 Zähler auf 14 (14 Sets werden geändert)
0024 übergabe an VDP-Speicher
0025 Erhöhen der Farbspeicherstelle um 1.
0026 Zähler minus eins
0027 Wenn Zähler nicht gleich Null dann NEXT1
```

4. Programmteil:
Wechseln der Farben, der Texte und Verzögerungsschleife:

```
0029 R6 mit VDP-7 und Transparent laden.
0030 R7 mit Hinweisadresse FARBEN-Texte
     laden.
0031 R8 mit  Zähler  laden  (16  Farben).
0032 R6 nach R0 kopieren (Inhalt).
0033 An VWTR übergeben.
0034 Bildschirmstelle >0 laden
0035 R7 nach          R1          kopieren
(Hinweis->Farben).
0036 R2 mit Textlänge (14) laden.
0037 Text schreiben.
0038 Zähler R5 mit 14 laden.
0039 R7 um eins erhöhen.
0040 Zähler verringern.
0041 Wenn Zähler  nicht  gleich  Null  ->
NEXT3
0042 Farbcode um eins erhöhen.
0043 Verzögerung mit >FFFF laden
0044 Vom Verzögerungszähler 1 abziehen.
0045 Wenn   Verzögerung   ungleich   Null
->NEXT4
0046 R8 zurückzählen
0047 Wenn nicht letzte Farbe dann NEXT2.
```

5. Programmteil:
Rücksprung zum Anfang des Programms.

Das Programm kann nur durch Abschalten abgebrochen werden.

Mit den anderen Sprungbefehlen können Sie genauso arbeiten. Sie müssen sich einfach bei jeder Operation fragen, wie das Statusbyte verändert wird. Mit den JUMP-Befehlen testen Sie bestimmte Bits dieses Bytes und verzweigen, wenn der Test positiv ausfällt.

Bei den Übungsaufgaben auf der nächsten Seite kommen Sie mit JMP und JEQ nicht mehr aus. Verwenden Sie verschiedene Sprungbefehle und erkunden Sie so ihre Wirkungsweise.

Übungsaufgaben:

1. Schreiben Sie ein Programm, welches wie das gezeigte alls Farben mit ihren Namen auf den Bildschirm bringt. Schreiben Sie anstelle der Verzögerungsschleife einen Programmteil, welcher erst auf Tastendruck weiterläuft.

2. Versuchen Sie einmal, ein Programm zu schreiben, mit dem Sie den Bildschirm vollschreiben können. Wenn der Bildschirm voll ist, soll die Eingabe wieder an der ersten Bildschirmposition beginnen.

3. Entwerfen Sie gemäß Aufgabe 2 ein Eingabefenster für 10 Character in der Mitte des Bildschirms. Geben Sie dabei auch einen Cursor aus!

Alle Aufgaben sind am Ende dieses Buches aufgelistet. Aufgabe 3 ist zusätzlich auf der beiliegenden Diskette enthalten.

SOUND UTILITIES

Alle reden von den Grafikeigenschaften des TI-99/4a.

Daß dieser Computer auch hervorragende Soundmöglichkeiten hat, glaubt keiner! Mit diesem Programm,

SOUND UNTILITIES

können Sie

- Sounds testen;
- Lieder Komponieren;
- Musiken speichern;
- diese in DATA-Zeilen umwandeln und
- in jedes Programm einbauen.

Ein LINE/LINE-Editor unterstützt Ihre Arbeit!

Tun Sie etwas für Ihre Ohren! Holen Sie sich die SOUND-UTILITIES ins Haus!!!

Diskette für Extended Basic nur DM 39.90

Rausch & Haub
Vertriebsgesellschaft dbR
Postfach 32 03 13
5300 BONN 3

1.10

JNE - JUMP IF NOT EQUAL

Syntax:

<Label> b JNE b exp b <Kommentar>

Beispiel:

```
INPUT  JNE  OVER
```

verzweigt nach OVER, wenn das EQ bit (auf 0) zurückgesetzt ist.

Definition:

Verzweigung wenn EQ=0.

Status Bit Benutzung:

Test von EQ. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn EQ=0 dann disp=> PC.
b) Wenn EQ=1 keine Veränderung.

10. VERSCHIEBEN UND VERGLEICHEN

Zwei weitere Funktionen sind für das Programmieren in Assembler unerlässlich. Wir haben sie bereits des öfteren verwendet, möchten aber an dieser Stelle nochmals ausführlich darauf eingehen. Bevor Sie daran gehen, kleine Probleme selbst zu lösen, wollen wir diese also näher Erklären. Befassen wir uns zunächst mit dem Verschieben.

Sie können Daten direkt in ein Register laden. Im TI-Assembler heißt dieser Befehl wie bereits gesehen LI (Load Immediate). Da Registern aber die vielfältigsten Aufgaben zukommen, sollten Sie es sich von vornherein angewöhnen, sämtliche Werte in bestimmten Speicherstellen abzulegen. Hierzu dienen Ihnen die Befehle MOV und MOVB.

MOV verschiebt ein 2-Byte-Wort von einem Register in ein anderes; von einer Speicherstelle in eine andere; von einem Register zu einer Speicherstelle und umgekehrt. Ein Beispiel:

```
0001 PROG   LI   R0,>0380
0002        MOV  R0,§ADRESS
```

Der Wert >0380 befindet sich jetzt sowohl in R0 als auch an Speicherstelle ADRESS. Das Zeichen § bedeutet 'Der Inhalt von...'.

Verschieben wir weiter:

```
0003          MOV  §ADRESS,§BUFF1
```

Jetzt befindet sich >0380 auch in BUFF1. Es gibt eine ganze Reihe unterschiedlicher Adressierungsarten, wohin ein Wort (oder Byte) verschoben werden soll.

ohne Zeichen: Der Hexcode der Speicherstelle.

Führendes * : Der Inhalt des Wortes an der Adresse im nachfolgenden Register, z.b. *R3.

Nachfolgendes + : Automatische Incrementierung. Bei Bytebefehelen wird um 1 erhöht, bei Wortbefehlen um 2. Beispiel: R3+.

Klammern ( ): Der Inhalt wird zum Ausdruck vor der Klammer Addiert und danach der Ausdruck behandelt.

Terme: z.B. ADR+8/2 bedeutet die Adresse ADR vermehrt um 8 und dividiert durch 2. Wenn ADR gleich 1000 ist, so ist das Ergebnis des Ausdrucks 0504, nicht etwa 1004.

```
0004          MOV  *R1+,§ADR+8(R6)
```

verschiebt das Wort an der Adresse, die in R1 steht an die Adresse, die sich ergibt, wenn man den Inhalt von R6 zur Summe von ADR+8 hinzuaddiert. Der Inhalt von R1 wird automatisch um 2 erhöht.

Gegeben seien R1, R6 und ADR:

Adresse/Register	Vorher	Nachher
R1	>1000	>1002
R6	>0004	>0004
§ADR+8	>A008	>A008
>1000	>2422	>2422
>A00C	>7234	>2422

Entsprechend ist die Möglichkeit, einzelne Bytes zu verschieben. Ein Beispiel:

```
0005        CLR  R1
0006        MOVB §>A00C,R1
```

Nach Durchführung steht in R1 der Wert >2400.
Nur das erste (führende) Byte wird bei Byteoperationen verschoben. Mit SWPB können Sie dann das Byte in eine Position bringen, welche die Weiterverwendung von Wortbefehlen ermöglicht:

```
0007        SWPB R1       *SWAP BYTE R1
```

In R1 steht jetzt der Wert >0024.

Alle unter MOV angegebenen Adressierungsarten sind anwendbar.
ACHTUNG: Der Automatische Inkrementierungs Befehl zählt hier nur um eins weiter, also entsprechend INC.

Wenn Sie also ab jetzt irgendwo etwas zählen möchten, tun Sie es doch mit dem automatischen Increment-Befehl.

Versuchen Sie einmal, Ihre bisherigen Übungsprogramme unter zuhilfenahme von MOV und MOVB neu zu schreiben.

Kommen wir nun zu den Vergleichsbefehlen. Sie dienen der Möglichkeit, Vergleiche anzustellen und das Ergebnis im Statusbyte festzustellen (mit einem Sprungbefehl wird entsprechend des Inhalts im Statusbyte weiter verfahren). Drei Vergleichsbefehle stehen zur Auswahl:

C - Compare vergleicht 2 Worte
CB - Compare Bytes vergleicht 2 Bytes
CI - Compare Immediate vergleicht mit Direktdaten.

Die Syntax von C und CB entspricht der von MOV und MOVB, die von C dem Befehl LI. Man sieht deutlich die 'Verwandtschaft'.

Eine kleine Routine:

```
0001          DEF   PROG6
0002          REF   VSBW
0003 PROG6    LI    R0,>0000
0004          LI    R1,>41
0005 NEXT     BLWP  @VSBW
0006          INC   R0
0007          CI    R0,>0300
0008          JLT   NEXT
0009 LOOP     NOP
0010          JMP   LOOP
0011          END
```

Was geschieht?
Das Programm schreibt den Bildschirm voll mit dem Buchstaben 'A'. In Zeile 0007 wird kontrolliert, ob die letzte Bildschirmposition beschrieben wurde. Falls nicht, wird das nächste 'A' geschrieben.

Im Wesentlichen sind also beim TI-Assembler drei Befehlsgruppen zur Schleifenprogrammierung erforderlich: Verschiebungen, Vergleiche und Sprünge. Wie Sie sehen, ist es möglich, in Verbindung mit den VDP-Unterprogrammen bereits kleine, nützliche Routinen zu schreiben.

Übungsaufgaben:

1. Wiederholen Sie die Aufgaben aus dem Kapitel 'Sprünge' und verwenden Sie dabei die Vergleichsoperationen.

2. Schreiben Sie ein Programm, welches jeweils fünf Zahlen miteinander vergleicht und die jeweils niedrigste und höchste auf dem Bildschirm ausgibt.

Die Aufgaben sind am Ende dieses Buches aufgelistet. Aufgabe 2 befindet sich zusätzlich auf der beiliegenden Diskette.

1.11

11. ARITHMETISCHE UND LOGISCHE BEFEHLE

Sie kennen jetzt die wichtigsten Programm Kontrollbefehle. Es gibt jedoch auch Befehle, mit denen Sie Werte, aber nicht den Programmablauf als solchen ändern können. Diese sind verschiedene mathematische Routinen und logische Bitoperationen.

Zunächst das Addieren. Es gibt hier wieder drei Befehle, von denen Sie nach der Lektüre des vorangegangenen Kapitels eigentlich schon ahnen könnten, was sie bedeuten:

AI - Add Immediate addiert Direktdaten
A - Add (Words) addiert Worte
AB - Add Bytes addiert Bytedaten

Beispiel:

```
0001            DEF   START
0002 START      LI    R0,>1522
0003            LI    R1,>3654
0004            LI    R2,>21B0
0005            AI    R0,>2310      *R0=>3832
0006            AB    R1,R0         *R0=>6E32
0007            A     R1,R2         *R0=>8FE2
0008 LOOP       NOP
0009            JMP   LOOP
0010            END
```

Während AI sich immer auf ein Register bezieht, können bei A und AB auch Adressen in der Form angegeben werden, die Ihnen von den Verschiebe- und Vergleichsinstruktionen bekannt sein dürften.

```
Beispiel:     A     §ZAHL,§SUMME
              AB    §BYTADR+2,§BYTSUM(R4)
```

Ähnlich funktioniert es mit dem Subtrahieren:

S - Subtract (Words) subtrahiert Worte
SB - Subtract Bytes   subtrahiert Bytes

SI gibt es jedoch nicht, da dies mit AI ohne weiteres simuliert werden kann.

Beispiel:

```
0001          DEF   START
0002 ZAHL     BSS   2
0003 START    LI    R1,>125B
0004          MOV   R1,§ZAHL
0005          LI    R1,>013A
0006          S     R1,§ZAHL      *§ZAHL=1121
0007          SB    R1,§ZAHL+1    *§ZAHL=1120
0008 LOOP     NOP
0009          JMP   LOOP
0010          END
```

Außerdem verfügt der TI-Assembler über Multiplikation und Division, was nicht bei jedem Assembler selbstverständlich ist. Oftmals muß man sich diese Programme selbst schreiben.

MPY - Multiply Multiplikation
DIV - Divide   Division

Mit diesen Programmen können Sie nur Register behandeln, und zwar in der folgenden Art und Weise:

MPY §MULTI,R3

Multipliziert wird der Inhalt des Wortes zum Inhalt des angegebenen Registers. Das Ergebnis steht im angegebenen Register und im darauffolgenden, ist also eine 32-Bit-Zahl.

Beispiel:

```
0001            DEF   START
0002 ZAHL       DATA  >0723
0003 START      LI    R1,>0335
0004            MPY   §ZAHL,R1
0005 LOOP       NOP
0006            JMP   LOOP
0007            END
```

In R1 steht danach >0000;
in R2 steht danach >156F;
denn >0723 x >0335 = >0000156F

Bei der Division werden die beiden Register anders genutzt. Dividiert werden die beiden Worte im angegebenen und darauf folgenden Register durch den Inhalt der beiden Worte ab der Angegebenen Adresse. Im ersten steht nach der Durchführung das Ergebnis als Ganzzahl, im zweiten der Teilungsrest.

```
0001            DEF   START
0002 ZAHL       DATA  >0002,>0422
0003 START      CLR   R1
0004            LI    R2,>0014
0005            DIV   @ZAHL,R1
0006 LOOP       NOP
0007            JMP   LOOP
0008            END
```

In R1 steht danach >019C
In R2 steht danach >000E
denn >00020422 / >0014 = >019C REST >000E

Übungsaufgaben:

Sie wissen bereits, wie man Daten vom Bildschirm liest und auf den Bildschirm bringt und sollten daher in der Lage sein, ein paar kleine Rechenprogramme zu schreiben:

Programmablauf:

1. Von einem Menü soll die gewünschte Grundrechenart gewählt werden.

2. Eingeben der beiden Operanden über ein Bildschirmfenster.

3. Ausgabe des Ergebnises in einem dritten Fenster.

4. Warten auf Tastendruck

5. Rückkehr zum Menü.

Ein Operand darf maximal 3stellig sein.

Die Lösung dieser Aufgabe befindet sich am Ende dieses Buches. Sie ist außerdem auf der beiliegenden Diskette enthalten.

Neben den Instruktionen, mit denen wir Worte und Bytes behandeln, gibt es jedoch auch solche, die es uns ermöglichen, Bits innerhalb eines Registers zu verschieben. Die Verschiebung von Bits um eines Stelle nach links entspricht einer Multiplikation um 2, um zwei Stellen einer Multiplikation um 4 und so weiter. Wir können also auch durch Bitverschiebungen Multiplikationen bzw. Division (Verschieben nach rechts) simulieren.

Die Bitverschiebe-Instruktionen lauten:

SRA - Shift Right Arithmetic
SRL - Shift Right Logical
SLA - Shift Left Arithmetic
SRC - Shift Right Circular

Diese Instruktionen werden in anderen Kursen gerne zu Anfang erläutert. Wir haben bewußt darauf verzichtet, da ohne die Kenntnis der übrigen Befehle kaum sichtbare Ergebnisse erzielt werden können.

Verschoben wird immer innerhalb eines Registers um eine anzugebende Zahl von Bits. Es gibt dabei noch einige Sonderregelungen, die im Anhang bei den Mnemonics erläutert sind. Als Einführung soll folgendes genügen.

SRA - Arithmetisch nach rechts verschieben

Beispiel: SRA R1,4 verschiebt den Inhalt des Registers 1 um vier Bitstellen nach

rechts. Freiwerdende Bitstellen werden mit dem Führungsbit, dem sogenannten Vorzeichenbit, aufgefüllt.

```
Inhalt von R1: >1036
= Binär .....: 0001000000110110
1. Rechtshift: 0000100000011011
2. Rechtshift: 0000010000001101
3. Rechtshift: 0000001000000110
4. Rechtshift: 0000000100000011
Neuer  Inhalt: >0103
```

SRL - Logisch nach rechts verschieben

Beispiel: SLR R1,4 verschiebt den Inhalt des Registers 1 um vier Bitstellen nach rechts. Freiwerdende Bitstellen werden mit binären Nullen aufgefüllt, nicht mit dem Führungsbit. Solange das Führungsbit=0 ist, sind SRA und SRL identisch. Das Beispiel von oben kann also in diesem Fall übernommen werden.

SLA - Arithmetisch nach links verschieben

Der Befehl ist die Umkehrung von SRA. Nach der Durchführung befindet sich im Übertragstatusbit der Wert des verschobenen Führungsbits. Freiwerdende Bitstellen werden durch binäre Nullen ersetzt.

Beispiel: SLA R1,4

```
Inhalt von R1: >1036
= Binär .....: 0001000000110110
1. Linksshift: 0010000001101100
2. Linksshift: 0100000011011000
```

3. Linksshift: 1000000110110000
4. Linksshift: 0000001101100000
Neuer Inhalt: >0360

Das Übertrag-Status-Bit ist auf Null zurückgesetzt, da beim ersten Linksshift eine Null hinausgeschoben wurde.

SRC - Zyklisch nach rechts verschieben

Bei anderen Assemblern heißt dieser Befehl zumeist ROTATE. Die Bits werden dabei nach rechts verschoben, wobei die jeweils freiwerdende Bitstelle mit dem Inhalt des hinausgeschobenen Bits gefüllt wird. Für das Übertrag-Statusbit gelten die Regeln von SLA.

Beispiel: SRC R1,4

Inhalt von R1: >B42E
= Binär .....: 1011010000101110
1. Linksshift: 0101101000010111
2. Linksshift: 1010110100001011
3. Linksshift: 1101011010000101
4. Linksshift: 1110100101000010
Neuer Inhalt: >E942

Das Übertrag-Statusbit ist gesetzt.

Anwendungsbeispiel:

Berechnung des Ausdrucks (nx8+4)/y
n sei >4, y sei >2.
Das Ergebnis steht in ZAHL3.

```
0001            DEF   START
0001   ZAHL1    DATA  >0004
0002   ZAHL2    DATA  >0002
0003   ZAHL3    BSS   2
0004   START    LI    R1,>0001
0005            SLA   R1,3
0006            MPY   §ZAHL1,R1
0007            AI    R2,>0004
0008            SRA   R2,1
0009            MOV   R2,§ZAHL3
```

Übungsaufgabe:

Ergänzen Sie das Programm um eine Bildschirmausgabe des Ergebnisses.

Das Listing dazu finden Sie am Ende dieses Buches.

1.12

12. SCHLUSSWORT ZUM KURS

Die in diesem Einleitungskurs behandelten Mnemonics sind nicht alle auf dem TI-99/4a verfügbaren. Weitere können Sie im Kapitel IV nachlesen. Die hier erläuterten Befehle sollten jedoch einen problemlosen Einstieg in die Assemblersprache ermöglichen, da sie alle Probleme damit lösen können.

Wir haben mit Absicht alles Komplizierte aus diesem Kurs herausgehalten, so etwa den Aufruf von Maschinenprogrammen aus dem Basic, das Arbeiten mit externen Dateien und ähnliches. Darüber gibt das Handbuch zum Editor/Assembler dem Fortgeschrittenen, der Sie ja nach Absolvierung dieses Kurses sind, genügende Auskunft.

Ein weiterführender Kurs, der auf dem Inhalt dieses Buches aufbaut, wird in Kürze erscheinen!

Ihre Rausch & Haub
Vertriebsgesellschaft

Hans-Georg Rausch
Der Autor.

2

II. Problemlösungen

Wenn Sie nun anhand des vorangegangenen Kurses mit dem TI-Assembler einigermaßen vertraut sind, können Sie damit beginnen, komplexere Probleme zu lösen. In diesem Kapitel wollen wir fünf unterschiedliche Probleme zu lösen versuchen.

Zu jedem Problem gibt es mehrere Lösungen. Versuchen Sie die Aufgaben eigenständig zu lösen, bevor Sie nochmals im Einführungskurs nachschlagen.

Beachten Sie bitte auch Punkt 6 am Ende dieses Kapitels!

1. Balkendiagramme

Als Basicprogramm finden wir dieses Problem mittlerweile in jedem Computerheft. Wir wollen an unser Programm gar keine größeren Ansprüche stellen. Folgendes soll es leisten:

Wir wollen den Absatz zweier Waren im Verlauf eines Jahres grafisch darstellen. Die Jahre sollen nebeneinander stehen, die Verkaufsmengen durch die Balkenhöhe ausgedrückt werden. Die Verkaufszahlen bewegen sich zwischen Null und Fünfhundert.

Die Farben für unsere Grafik wollen wir so gestalten, daß alles gut lesbar und nicht anstrengend für das Auge ist: Hintergrund schwarz, Koordinatenwinkel grau, Schrift gelb, Balken dunkelrot und dunkelgrün.

Die Verkaufszahlen sollen nacheinander in der untersten Zeile eingegeben werden können. Dazu benötigen wir ein Eingabe-Unterprogramm mit folgenden Funktionen:

Akzeptieren von ganzen Zahlen zwischen 0 und 500.
Löschen einer nicht beendeten Eingabezeile.
Einschieben und Löschen einzelner Zeichen in der Zeile.
Als Cursor soll ein feststehender Block dienen, die Farbe sei hellblau.

Zum Ablauf des Programms: Nach dem Start

soll sich die Eingabezeile mit der Nachricht 'Ware 1: Ware 2: ' melden und in jedem Feld die Eingabe einer Zahl zwischen 0 und 500 verlangen. Bei einer falschen Eingabe soll diese wiederholt werden. Sind 12x2 Eingaben korrekt erfolgt, sollen die Balken entsprechend den Werten so genau wie möglich als Grafik dargestellt werden. Sodann wird auf Tastendruck die Grafik gelöscht und der Bildschirm steht für die Eingabe neuer Daten bereit.

2. Klaviertastatur

Dieses Programm soll es ermöglichen, auf dem TI-99/4a zu komponieren. Dazu wandeln wir die Tastatur in ein Klavier um. Gespielt werden soll in allen 4 Oktaven.

Der Bildschirm soll jeweils anzeigen, welchen Ton wir gerade spielen. Außerdem soll dieser Wert gespeichert werden. Auf Knopfdruck muß der Computer die gespeicherten Werte benutzen, um eine Melodie zu spielen. Wir benötigen also auch Funktionen zur Speichersteuerung und zum Löschen des Bildschirms.

Die Darstellung soll rot auf grau erfolgen; die Eingabe von Tönen durch Tastendruck mit dem KSCAN Unterprogramm.

3. Textbearbeitung

An diese Aufgabe sollten Sie sich nur heranwagen, wenn Sie bereits etwas mehr über den Assembler wissen, als dieser Kurs vermitteln soll und kann. Sie benötigen dazu Kenntnisse von Dingen, die in diesem Kurs nicht behandelt werden, die aber im Handbuch zum Editor/Assembler ausführlich beschrieben werden.

Die Hauptaufgabe soll es diesmal sein, einen PAB (Peripheral Acess Block) zu bilden, der es uns ermöglicht, Daten an ein externes Gerät weiterzugeben. Für die Lösung dieser Aufgabe ist ein Drucker erforderlich. Sollten Sie keinen Drucker besitzen, so können Sie auch ein Diskettenlaufwerk benutzen. Sie müssten sich dann allerdings ein kleines Programm zur überprüfung der gespeicherten Daten schreiben.

Nach dem Starten des Programms soll der Bildschirm gelöscht werden. In der oberen linken Ecke wird ein einfarbiger gründer Cursor auf schwarzem Grund positioniert. Mit diesem können wir Zeile für Zeile des Bildschirms vollschreiben. Der Cursor soll Bildschirmorientiert sein - also mit den Steuertasten an jede beliebige Stelle gebracht werden.

Folgende Sonderfunktionen muß das Programm besitzen:

Delete: Löschen einzelner Zeichen mit zurechtrücken des Textes.
Insert: Einfügen einzelner Zeichen mit

zurechtrücken des Textes.
Erase: Löschen der Zeile, in der sich der Cursor befindet.
Clear: Löschen des gesamten Bildschirmes.
Begin: Positionierung des Cursors am Testanfang.
Proceed: Starten des Ausdruckvorganges.

Bei Proceed soll der Ausdruck im 28-Zeichen-Format auf einem Drucker erfolgen. Ein entsprechender PAB muß geschrieben werden.

Nach dem Druckvorgang soll der Bildschirm gelöscht und die Startsituation hergestellt weden.

4. Etwas Mathematik

Mit diesem Programm wollen wir ein wenig die Rechenfähigkeiten des TI ausnutzen. Das Programm soll uns helfen, kleine Kalkulations und kaufmännische Rechnungen durchzuführen. Folgendes soll gelöst werden:

Eine Ware X kostet ab Lager einen Betrag Y. Dazu kommen Fuhrgeld zum Bahnhof, Fracht und Fuhrgeld zu unserem Betrieb. Wir berechnen außerdem unsere Geschäftskosten nach einem Prozentsatz, Gewinn und Mehrwertsteuer. Unsere Kunden verlangen Rabatt und Skonto.

Programmablauf: Der Computer erfragt nacheinander alle Angaben und berechnet die Zwischensummen und das Endergebnis. Nach der letzten Berechnung geht es vom Start auf Knopfdruck weiter.

Farben und Bildschirmaufbau seien Ihnen überlassen. Haben Sie eine originelle Idee?

5. Grafik Total

Mit diesem Programm wollen wir es ermöglichen, grafische Darstellungen auf dem TI zu zeichnen. Dazu definieren wir verschiedenartige Linien, Winkel und Muster in den hohen Zeichensatzbereich (Character über 159). Die gesamte Darstellung soll zweifarbig in schwarz und grau erfolgen.

Die unteren zwei Zeilen sollen für unsere Auswahl zur Verfügung stellen. Eingegeben werden sollen:

a) Characternummer
b) Zeile und Spalte
c) Anzahl der Wiederholungen
d) Wiederholungsrichtung vertikal oder horizontal

Außerdem soll die Möglichkeit gegeben sein, mittels eines Cursors direkt auf dem Bildschirm zu zeichnen.

Sonderfunktionen:

Clear: Löschen des Bildschirms
Bildschirmorientierter Cursor.

3

III. Ein Spiel in Assembler

Eigentlich müsste es Ihnen jetzt möglich sein, selbst ein Maschinenprogramm zu schreiben. Damit dies am Anfang nicht zu schwierig wird, haben wir einen Ausschnitt des HAGERA(R)-Programms 'PARTISAN VILLAGE' hier abgedruckt. Dieser Ausschnitt ist auf der beiliegenden Diskette sowohl im Assemblercode als auch im reinen Maschinencode (lauffähig) vorhanden.

In diesem Spielteil werden keine Sprites verwendet. Ihre Aufgabe soll es sein, eigene Ideen in diesem Spiel zu realisieren. Doch schauen Sie sich zunächst einmal das Spiel an.

Partisan Village ist ein Spiel in drei Phasen. Lediglich die erste wird in diesem Kurs verwendet. Ziel ist es, die Panzer von der rechten Seite auf die linke zu bringen, ohne über eine Mine zu fahren. Minen können von den Panzern (in diesem Ausschnitt) nicht geortet werden. Merken Sie sich also die gefahrenen Wege, um sicher ans Ziel zu kommen.

Das Programm ist Zeile für Zeile im Listing kommentiert und läuft im Wesentlichen folgendermaßen ab:

1. Vordefinitionen
2. Initialisierung
3. Bildschirmaufbau
4. Spielstart auf Tastendruck
5. Panzersteuerkontrolle
6. Gültigkeit der Fahrtrichtung

7. Mine befahren?
8. Spielende und Abfrage ob neues Spiel

Ihre Aufgabe soll es sein, das Spiel um folgende Positionen zu erweitern und gegebenenfalls alte Routinen zu ersetzen. Gehen Sie Schritt für Schritt vor und testen Sie nach jeder Änderung das neue Programm. Wenn Sie nicht weiterkommen, schlagen Sie nochmals im Einführungsteil nach.

1. Zufällige Festlegung der Minenfelder unter der Vorgabe, daß mindestens ein Weg Minenfrei ist.

2. Schußmöglichkeit des Panzers in Fahrtrichtung auf ein Gebäude, welches dadurch (teilweise) zerstört wird, so daß sich der Panzer einen Weg freischießen kann.

3. Akustische Ortungsmöglichkeit für die Minen.

4. Änderung der Tastaturbelegung für Panzerbewegungen nach Ihren persönlichen Wünschen nach Starten des Programms.

5. Änderung des Punktebewertungssystems:
Bisher: 1 Punkt für jeden Panzer im Ziel.

Änderungen:
100 Punkte für jeden Panzer im Ziel.
10 Punkte Abzug für das Befahren einer Mine, jedoch nur, wenn Punkte vorhanden sind.

6. Die Anzeige ist für Bevölkerung und Feinde vorgesehen. Benutzen Sie diese Anzeige und zeigen Sie an:
a) Eine Einwohnerzahl, die sich bei jedem Gebäudetreffer vermindert.
b) Eine Feindzahl, die sich bei der totalen Venichtung eines Gebäudes verringert.

Am Ende einer Runde sollen Zusatzpunkte vergeben werden: Anzahl der noch lebenden Einwohner multipliziert mit der Anzahl der vernichteten Feinde.

7. Beschränken Sie die Schußzahl eines Panzers, aber geben Sie die Möglichkeit, den Munitionsvorrat zu ergänzen.

8. Beschränken Sie den Treibstoffvorrat eines Panzers, aber geben Sie die Möglichkeit, den Vorrat zu ergänzen.

9. Machen Sie die entsprechenden 'Balkenanzeigen' lauffähig.

10. Erweitern Sie das Spiel um verschiedene Schwierigkeitsgrade durch unterschiedliche Anzahl von Minen, Bevölkerung, Feinden, Schußzahl, Treibstofftankgröße etc.

11. Programmieren Sie selbst für jeden Schwierigkeitsgrad ein neues Bild (den Grundriß einer neuen Stadt).

12. Untermalen Sie die verschiedenen Aktionen und Spielsituationen mit Geräuschen und Musik.

Wenn Sie alle Änderungen wie beschrieben vornehmen und die Erläuterungen im ersten Teil dieses Kurses berücksichtigen, befinden Sie sich im Besitz eines Spiels, welches unserem PARTISAN VILLAGE sehr ähnlich sein dürfte.

Zu diesen Aufgaben haben wir absichtlich keine Musterlösungen abgedruckt. Die Gefahr ist doch groß, daß man allzu schnell aufgibt und nachschlagen möchte, wie 'es geht'.

Wenn Sie jedoch wirklich nicht weiterkommen, dann schreiben Sie uns einfach (bitte einen frankierten und adressierten Rückantwortumschlag beifügen). Wir helfen Ihnen dann ein wenig.

Auf den nachfolgenden Seiten finden Sie nun das Listing zum Spiel sowie zu den Übungsaufgaben aus Kapitel 1.

```
**************************************************************
**************************************************************
**                                                          **
**                                                          **
** PARTISAN VILLAGE (C)1985 BY HAGERA  HANS-GEORG RAUSCH **
**                                                          **
**       Auszug aus dem gleichnamigen Action-Game           **
**                                                          **
**       von  HAGERA                                        **
**            Rausch & Haub                                 **
**            Vertriebsgesellschaft d.h.R.                  **
**            Postfach 320313                               **
**            5300 BONN 3                                   **
**                                                          **
**                                                          **
**************************************************************
**************************************************************
*
*
      IDT  'PARTVIL'          *Programmidentifizierung
*                             *(siehe Mnemonic-Tabellen)
      DEF  BEGIN1             *Eingangsstellen-Definitionen
      REF  VMBW,VMBR,VWTR     *Eingeschlossene ROM-Routinen
      REF  VSBW,VSBR          *dto.
      REF  KSCAN              *dto.
*
*
*
*********************************************
*                                           *
*     VORDEFINITIONEN, BUFFER, VARIABLE ETC *
*                                           *
*********************************************
*
*
*
SCRN1 TEXT ''' <ENTER> START ' <E,S,D,X> ''''    *SCREEN
      TEXT '''''''''''''''''''''''''''''''''''
```

```
       TEXT '``jjjjjdeefjjjjjdeefjjjjjd````'
       TEXT '``jabbcjdeefjabbcjdeefjabbcjd````'
       TEXT '``jdeefjdeefjdeefjdeefjdeefjd````'
       TEXT '``jdeefjghhijdeefjghhijdeefjd````'
       TEXT '``jdeefjjkjjjdeefjjkjjjdeefjd````'
       TEXT '``jdeefjabbcjdeefjabbcjdeefjgh```'
       TEXT '``jghhijdeefjdeefjdeefjghhijjj```'
       TEXT '``jyjkjjdeefjdeefjdeefjjjkjjsj```'
       TEXT '``jabbcjdeefjdeefjdeefjabbcjjj```'
       TEXT '``jdeefjdeefjdeefjdeefjdeefjab```'
       TEXT '``jdeefjghhijdeefjghhijdeefjd````'
       TEXT '``kdeefjjjjjjdeefjjjjjjdeefjd````'
       TEXT '``jdeefjabbcjdeefjabbcjdeefjd````'
       TEXT '``jghhijdeefkghhijdeefkghhijd````'
       TEXT '``jjjjjdeefjjjjjdeefjjjjjd````'
       TEXT '''''''''''''''''''''''''''''''''''
       TEXT '''''''''''''''''''''''''''''''''''
       TEXT 'POPULARS: 000000  ENEMYS: 00000 '
       TEXT 'FUEL....:                       '
       TEXT 'AMORCES.:                       '
       TEXT 'POINTS..:         BONUS.:       '
       TEXT 'PARTISAN VILLAGE z1984 BY HAGERA'
*
NACHR  TEXT ' GAME OVER '              *Spielende
       EVEN                            *Adresse geradzahlig
*
CHAR   DATA >F00F,>F00F,>F00F,>F00F    *' Spielfeldrand
       DATA >0000,>0000,>0000,>0001    *a Gebaeude, Ecke
       DATA >0000,>0000,>0000,>00FF    *b  " oberer Rand
       DATA >0000,>0000,>0000,>0080    *c  " oben rechts
       DATA >0101,>0101,>0101,>0101    *d  " linker Rand
       DATA >8142,>2418,>1824,>4281    *e  " Mitte
       DATA >8080,>8080,>8080,>8080    *f  " rechter Rand
       DATA >0100,>0000,>0000,>0000    *g  " unten links
       DATA >FF00,>0000,>0000,>0000    *h  " unterer Rand
       DATA >8000,>0000,>0000,>0000    *i  " unten rechts
       DATA >0000,>0000,>0000,>0000    *j  Strasse
       DATA >0000,>0000,>0000,>0000    *k  versteckte Mine
       DATA >0000,>0000,>0000,>0000    *l  ... Frei
```

```
       DATA >0000,>0000,>0000,>0000   *m  ... fuer
       DATA >0000,>0000,>0000,>0000   *n  ... Ihre
       DATA >0000,>0000,>0000,>0000   *o  ... Ergaenzungen
       DATA >8199,>99FF,>FFFF,>8181   *p  Panzer, nach oben
       DATA >FF38,>383E,>3E38,>38FF   *q   " nach links
       DATA >8181,>FFFF,>FF99,>9981   *r   " nach unten
       DATA >FF1C,>1C7C,>7C1C,>1CFF   *s   " nach rechts
       DATA >0000,>0000,>0000,>0000   *t  ... Frei
       DATA >0000,>0000,>0000,>0000   *u  ... fuer
       DATA >0000,>0000,>0000,>0000   *v  ... Ihre
       DATA >0000,>0000,>0000,>0000   *w  ... Ergaenzungen
       DATA >8138,>FF7C,>FF38,>99FF   *x  Panzer zerstoert
       DATA >C0E0,>F0E0,>C080,>8080   *y  Zielfaehnchen
       DATA >3C42,>99A1,>A199,>423C   *z  Copyright-Zeichen
*
COLOR  DATA >A8A8,>A8A8,>A8A8,>A8A8   *Standartcharacter
       DATA >1E1E,>5E6E               *Sonderzeichen
* .
ZERO   DATA >0000                     *Festzahlen fuer Vergleiche
ONE    DATA >0001                     *und Verschiebungen
ZEHN   DATA >000A
*
POS1   DATA >013C                     *Startposition Panzer
SP     DATA >0000                     *Buffer fuer neue Position
SPALTE DATA >0001                     *Spaltenoffset zur Position
ZEILE  DATA >0020                     *Zeilenoffset zur Position
PANZER DATA >0002                     *Anzahl der Bonus-Panzer
PUNKTE DATA >0000                     *Punktestand
*
TOENE  BYTE >03,>8C,>1F,>91,>15       *Soundliste fuer VDP >1000
       BYTE >03,>8C,>1A,>91,>15
       BYTE >03,>8C,>1F,>91,>15
       BYTE >03,>8D,>11,>91,>15
       BYTE >03,>8C,>1A,>91,>15
       BYTE >03,>8C,>1F,>91,>15
       BYTE >03,>8D,>11,>91,>15
       BYTE >03,>83,>15,>91,>15
       BYTE >03,>83,>15,>91,>15
       BYTE >03,>8E,>0B,>91,>15
```

```
      BYTE >03,>85,>1C,>91,>15
      BYTE >03,>8D,>17,>91,>15
      BYTE >03,>83,>15,>91,>15
      BYTE >03,>8C,>1A,>91,>15
      BYTE >01,>9F,>0
*
BYTE1 BYTE >01                    *Byte-1 fuer Musikstart
      EVEN                        *Adresse geradzahlig
*
POS   BSS  >2                     *Buffer fuer Panzerposition
*
*
*
***********************************************
*                                             *
*     INITIALISIERUNG DES SPIELS              *
*                                             *
***********************************************
*
*     BILDSCHIRMAUFBAU
*
BEGIN1 NOP                  *SPIELSTART
      CLR  R0               *R0 = Screen im VDP-Ram >0000
      LI   R1,SCRN1         *R1 = Startadresse Titelbild
      LI   R2,768           B Bildschirmpositonen
      BLWP @VMBW            *an VDP-RAM uebergeben
*
      LI   R0,>0B00         *R0 = Sonderzeichen (ASCII 96)
      LI   R1,CHAR          *R1 = Startadresse Musterbeschr.
      LI   R2,216           *Bytes = Muster schreiben
      BLWP @VMBW            *an VDP-RAM uebergeben
*
      LI   R0,>0384         *R0 = Farben im VDP-Ram (Adr.)
      LI   R1,COLOR         *R1 = Startadresse Chr.-Farben
      LI   R2,12             Bytes (6x2) schreiben
      BLWP @VMBW            *an VDP-RAM uebergeben
*
      LI   R0,>0701         *R0 = Bildschirmfarbe
      BLWP @VWTR            *An VDP-Register uebergeben
```

```
*
      LI   R0,>0002          *Panzerzahl laden
      MOV  R0,@PANZER        *und uebertragen
      CLR  R9                *Flagregister loeschen
      MOV  @POS1,@POS        *Startposition laden
*
*
*
*
***********************************************
*                                             *
*     HAUPTTEIL I: START UND PANZERBEWEGUNG   *
*                                             *
***********************************************
*
*     TASTATURABFRAGE START
*
BEGIN2 NOP                   *Eingangsstelle Spielsteuerung
      CLR  R5                *R5 = >0000
      MOVB R5,@>8374         *Byte fuer KSCAN = Tastatur
SOUND  LI   R0,>1000         *Soundadresse im VDP laden
      LI   R1,TOENE          *Hinweisadresse zu Toenen laden
      LI   R2,>49            *>49 Bytes
      BLWP @VMBW             *ins VDP-RAM schreiben
TAST1A CI   R9,>0001         *Start gedrueckt?
      JEQ  LOS               *Wenn ja, Start!
      LIMI 2                 *VDP-Interrupt-Umschaltung
      LIMI 0                 *VDP-Interrupt-Umschaltung
      LI   R6,>1000          *Soundadresse im VDP laden
      MOV  R6,@>83CC         *Tabellenhinweis-Adresse laden
      SOCB @BYTE1,@>83FD     *VDP-Markierung, siehe
*                            *Mnemonic-Erlaeuterungen SOCB
      MOVB @BYTE1,@>83CE     *Musik starten
TAST1B LIMI 2                *VDP-Interrupt-Umschaltung
      LIMI 0                 *VDP-Interrupt-Umschaltung
      MOVB @>83CE,@>83CE     *Ueberpruefen, ob alle Toene
*                            *gespielt
      JEQ  TAST1A            *Falls ja, von vorne beginnen
      LIMI 2
```

```
      LIMI 0
*
      BLWP @KSCAN          *Tastaturabfrage
      MOVB @>837C,R5       *GPL-Status-Byte in R5 schieben
      JEQ  TAST1B          *Wenn keine Taste Ruecksprung
      LI   R9,>0001        *Flag Taste gedrueckt
      JMP  TAST1B          *Ruecksprung
*                          *Ruecksprung zur Kontrolle
*
LOS   CLR  R9              *Flag-Register Spielende loeschen
BEGIN3 NOP                 *s.o.
      LI   R8,>1FFF        *Verzoegerung laden
VERZ  DEC  R8              *vermindern
      JNE  VERZ            *Wenn nicht beendet Ruecksprung
      LI   R5,>FF00        *Zeichen fuer Keine Taste laden
      MOVB R5,@>8375       *und uebertragen
      CLR  R5              *R5= >0000
      MOVB R5,@>8374       *s.o.
      BLWP @KSCAN          *s.o.
*                          *BEGINN DER UEBERUEFUNG:
      CLR  R5              *R5 loeschen
      MOVB @>8375,R5       *Gegrueckte Taste in R5
      SWPB R5              *Bytes vertauschen
      CI   R5,>0045        *Ist Taste = <E>
      JEQ  OBEN            *Wenn ja nach oben fahren
      CI   R5,>0044        *Ist Taste = <D>
      JEQ  RECHTS          *Wenn ja nach rechts fahren
      CI   R5,>0058        *Ist Taste = <X>
      JEQ  UNTEN           *Wenn ja nach unten fahren
      CI   R5,>0053        *Ist Taste = <S>
      JEQ  LINKS           *Wenn ja nach links fahren
      CI   R5,>000D        *Ist Taste = <ENTER>
      JEQ  BEGIN1          *Wenn ja neues Spiel
      JMP  BEGIN3          *Ruecksprung
*
*
OBEN  NOP                  *nach oben fahren
      LI   R6,>7000        *Zeichen laden
      MOV  @POS,@SP        *Panzerposition holen
```

```
        S    @ZEILE,@SP        *1 Zeile abziehen
        BL   @CONTR            *Ist die Richtung moeglich?
        B    @SPENDE           *Verzweigung - Ist Ende?
*
RECHTS  NOP                    *Nach rechts fahren
        LI   R6,>7100          *Zeichen laden
        MOV  @POS,@SP          *Panzerposition holen
        A    @SPALTE,@SP       *1 Spalte addieren
        BL   @CONTR            *Ist die Richtung moeglich?
        B    @SPENDE           *Verzweigung - Ist Ende?
*
UNTEN   NOP                    *Nach unten fahren
        LI   R6,>7200          *Zeichen Laden
        MOV  @POS,@SP          *Panzerposition holen
        A    @ZEILE,@SP        *1 Zeile addieren
        BL   @CONTR            *Ist die Richtung moeglich?
        B    @SPENDE           *Verzweigung - Ist Ende?
*
LINKS   NOP                    *Nach links fahren
        LI   R6,>7300          *Zeichen laden
        MOV  @POS,@SP          *Panzerposition holen
        S    @SPALTE,@SP       *1 Spalte abziehen
        BL   @CONTR            *Ist die Richtung moeglich?
        B    @SPENDE           *Verzweigung - Ist Ende?
*
*
*
*****************************************************
*                                                   *
*                                                   *
*     UNTER- UND HILFSPROGRAMME SPIELKONTROLLE / ENDE *
*                                                   *
*                                                   *
*****************************************************
*
*
*
CONTR   MOV  @SP,R0            *Position in R0
        CLR  R1                *R1 loeschen
```

```
        BLWP @VSBR              *Was ist da?
        SWPB R1                 *Bytes vertauschen
        CI   R1,>006A           *Ist es die Strasse?
        JEQ  SETP               *Wenn ja Panzer setzen
        CI   R1,>0079           *Ist es das Faehnchen?
        JEQ  SIEG               *Wenn ja naechste Runde
        CI   R1,>006B           *Ist es eine Mine?
        JEQ  ENDE               *Wenn ja dann Ende
        RT
*
*
SETP    NOP                     *Panzer setzen
        MOV  @POS,R0            *Alte Position laden
        LI   R1,>6A00           *Strassensymbol laden
        BLWP @VSBW              *und damit Panzer loeschen
        MOV  @SP,@POS           *Neue Position sichern
        MOV  @SP,R0             *Position in R0
        MOV  R6,R1              *Zeichen gewinnen
        BLWP @VSBW              *Zeichen schreiben
        RT                      *Rueckkehr hinter BL!
*
*
SIEG    NOP                     *Faehnchen erreicht
        MOV  @POS,R0            *Alte Position laden
        LI   R1,>6A00           *Strassensymbol laden
        BLWP @VSBW              *und damit Panzer loeschen
        LI   R0,>013D           *Startposition laden
        MOV  R0,@POS            *und uebertragen
        LI   R1,>7300           *Panzer Richtung links
        BLWP @VSBW              *An Start setzen
        A    @ONE,@PUNKTE       *Punkte+1
        C    @PUNKTE,@ZEHN      *Sind 10 Punkte erreicht?
        JEQ  HYMNE              *Wenn ja Siegeshymne
        MOV  @PUNKTE,R1         *Punkte in R1
        AI   R1,>0030            Addieren = Character
        SWPB R1                 *Bytes vertauschen
        LI   R0,720             *Punktausgabe laden
        BLWP @VSBW              *Punkte ausgeben
        RT                      *Rueckkehr hinter BL!
```

```
*
ENDE   NOP                  *Auf eine Mine gefahren
       MOV  @POS,R0         *Alte Position laden
       LI   R1,>6A00        *Strassensymbol laden
       BLWP @VSBW           *und damit Panzer loeschen
       LI   R3,>0008        *Zaehler laden
LOOP0  LI   R2,>0FFF        *Zaehler laden
       MOV  @SP,R0          *Neue Position laden
       LI   R1,>7800        *Explosionszeichen laden
       BLWP @VSBW           *und ausgeben
LOOP1  DEC  R2              *R2 vermindern
       JNE  LOOP1           *Wenn nicht=0 dann Schleife
       LI   R1,>7300        *Panzersymbol laden
       BLWP @VSBW           *und ausgeben
       LI   R2,>0FFF        *Zaehler laden
LOOP2  DEC  R2              *vermindern
       JNE  LOOP2           *wenn nicht=0 dann Schleife
       DEC  R3              *vermindern
       JNE  LOOP0           *wenn nicht=0 dann Schleife
       LI   R1,>6A00        *Strassensymbol laden
       BLWP @VSBW           *und ausgeben
       DEC  @PANZER         *Panzer vermindern
       C    @PANZER,@ZERO   *Keine Panzer mehr?
       JNE  ENDOUT          *Wenn doch weiter
       LI   R9,>0001        *R9 als Flag setzen
ENDOUT LI   R0,>0130        *Startposition laden
       MOV  R0,@POS         *und uebertragen
       LI   R1,>7300        *Panzersymbol laden
       BLWP @VSBW           *und ausgeben
       RT                   *Zurueck nach BL!
*
*
HYMNE  NOP                  *Hier Siegesmelodie einfuegen
       LI   R0,>0012        *R0 mit 1. Zeile Anfang laden
       LI   R1,NACHR        *Auf NACHR zeigen
       LI   R2,>B            Bytes
       BLWP @VMBW           *schreiben
       LI   R1,>000F        *Pausenzaehler laden
VERZ1A LI   R0,>7FFF        *Pausenzaehler laden
```

```
VERZ1   DEC   R0              *vermindern
        JNE   VERZ1           *wenn nicht zuende Ruecksprung
        DEC   R1              *vermindern
        JNE   VERZ1A          *wenn nicht zuende Ruecksprung
        MOV   @ZERO,@PUNKTE   *Punkte loeschen
        B     @BEGIN1         *Neustart
*
*
SPENDE  NOP                   *Ueberpruefung auf Spielende
        CI    R9,>0001        *Ist Spielende erreicht?
        JEQ   ENDALL          *Wenn ja Spielende
        B     @BEGIN3         *Ansonsten weiter
*
*
ENDALL  NOP                   *Hier Schlussmusik einfuegen
        LI    R0,>0012        *Unterste Zeile laden
        LI    R1,NACHR        *Game Over laden
        LI    R2,>A           *=10 BYTES
        BLWP  @VMBW           *an VDP uebergeben
        LI    R1,>000F        *Pausenzaehler laden
NPAUS1  LI    R0,>7FFF        *Pausenzaehler laden
NPAUS   DEC   R0              *vermindern
        JNE   NPAUS           *wenn nicht beendet Ruecksprung
        DEC   R1              *vermindern
        JNE   NPAUS1          *wenn nicht beendet Ruecksprung
        B     @BEGIN1         *Neustart
*
*****************************************************************
*
        END
```

PARTISAN VILLAGE

Die Diskette mit dem kompletten Spiel aus diesem Buch!

Tolle Action, Sound, Grafik in reiner Maschinensprache.

Befreien Sie die Kleinstadt von den Partisanen...

Lassen Sie sich zum General befördern...

PARTISAN VILLAGE...

für TI-99/4a, Speichererweiterung, Diskettenlaufwerk und Joysticks mit

- Extended Basic Modul

Mit deutscher Anleitung nur DM 39.90

oder

- Editor Assembler Modul

Mit deutscher Anleitung nur DM 34.90

4

1. Lösungen zum Kapitel 1:

1. Aufgabe:

Umbenennung eines Programms.

```
0001          DEF   START
0002          REF   VSBW
0003 START    LI    R0,>0000
0004          LI    R1,>4100
0005          BLWP  §VSBW
0006 LOOP     NOP
0007          JMP   LOOP
0008          END
```

Der Name muß als Eingangsadresse, aber auch in der DEF-Anweisung geändert werden.

2. Aufgabe:

Veränderung der Bildschirmposition horizontal.

```
0001          DEF   START
0002          REF   VSBW
0003 START    LI    R0,>0001
0004          LI    R1,>4100
0005          BLWP  §VSBW
0006 LOOP     NOP
0007          JMP   LOOP
0008          END
```

Die Bildschirmposition wird in Register 1 geändert. Wir erreichen dies durch Änderung des Ladebefehls für dieses Register in Zeile >0003.

3. Aufgabe:

Veränderung der Bildschirmposition vertikal.

```
0001          DEF   START
0002          REF   VSBW
0003 START    LI    R0,>0020
0004          LI    R1,>4100
0005          BLWP  @VSBW
0006 LOOP     NOP
0007          JMP   LOOP
0008          END
```

Noch eine Änderung. Diesmal eine Zeile tiefer (wir erhöhen um 32 = >20 Positionen).

4. Aufgabe:

Schreiben von verschiedenen Zeichen.

```
0001          DEF   START
0002          REF   VSBW
0003 START    LI    R0,>0000
0004          LI    R1,>5800
0005          BLWP  @VSBW
0006 LOOP     NOP
0007          JMP   LOOP
0008          END
```

>58 ist der ASCII-Code von 'X' (=Dez. 88). Um dieses Zeichen zu schreiben, müssen wir Register 1 ändern. Mit den Zahlen zwischen >41 und >5A können wir alle Großbuchstaben erreichen.

5. Aufgabe:

Schreiben eines Wortes mit Hilfe von VSBW.

```
0001          DEF  START
0002          REF  VSBW
0003 START    LI   R0,>0000
0004          LI   R1,>4100
0005          BLWP §VSBW
0006          LI   R0,>0001
0007          LI   R1,>5800
0008          BLWP §VSBW
0009          LI   R0,>0002
0010          LI   R1,>5400
0011          BLWP §VSBW
0012 LOOP     NOP
0013          JMP  LOOP
0014          END
```

Nacheinander werden die Buchstaben 'A', 'X' und 'T' am oberen Rand ausgegeben. Nach jeder Ausgabe wird die Bildschirmposition neu festgelegt und dem Register 2 ein anderes Zeichen zugewiesen. Sie werden noch andere Möglichkeiten kennenlernen, etwas um eins zu erhöhen!

6. Aufgabe:

Ein Wort in Bildschirmmitte ausgeben.

```
0001          DEF   START
0002          REF   VSBW
0003 START    LI    R0,>0170
0004          LI    R1,>4100
0005          BLWP  §VSBW
0006          LI    R0,>0171
0007          LI    R1,>5800
0008          BLWP  §VSBW
0009          LI    R0,>0172
0010          LI    R1,>5400
0011          BLWP  §VSBW
0012 LOOP     NOP
0013          JMP   LOOP
0014          END
```

Dieses Programm ist ausführlich auf der Diskette dorkumentiert.

2. Lösungen zum Kapitel 2:

1. Aufgabe:

Schreiben einer Zeichenkette mit VMBW.

```
0001         DEF   START
0002         REF   VMBW
0003 SATZ    TEXT  'DRUECKEN SIE EINE TASTE'
0004         EVEN
0005 START   LI    R0,>0000
0006         LI    R1,SATZ
0007         LI    R2,>17
0008         BLWP  @VMBW
0009 LOOP    NOP
0010         JMP   LOOP
0011         END
```

Die Register werden wie erforderlich geladen. Die EVEN-Direktive unter SATZ veranlasst den Assembler, mit einer geradzahligen Speicherstelle fortzufahren, wenn er SATZ abgelegt hat. EVEN sollten Sie immer dann verwenden, wenn die Zeichenkette in TEXT ungerade ist.

2. Aufgabe:

Beschriften eines Bildschirms:

```
0001         DEF   START
0002         REF   VMBW
*
```

Die Text-Anweisungen im Folgenden müssen durch Leerzeichen so aufgefüllt werden, daß jeweils 32 Zeichen in jeder Zeile stehen!

```
*
0003 BILD  TEXT 'DIES IST EIN MENUE:'
0004       TEXT '                   '
0005       TEXT 'DRUECKEN SIE:      '
0006       TEXT '                   '
0007       TEXT '                   '
0008       TEXT '<1>  PROGRAMM 1    '
0009       TEXT '<2>  PROGRAMM 2    '
0010       TEXT '<3>  PROGRAMM 3    '
0011       TEXT '<4>  PROGRAMM 4    '
0012       TEXT '<5>  PROGRAMM 5    '
0013       TEXT '                   '
0014       TEXT '                   '
0015       TEXT '(C)  HAGERA  1985  '
0016       TEXT '                   '
```

weitere 10 Leerzeilen (wie 0016) anfügen!

```
0027       EVEN
0028 PROG2 LI   R0,>0000
0029       LI   R1,BILD
0030       LI   R2,768
0031       BLWP §VMBW
0032 LOOP  NOP
0033       JMP  LOOP
0034       END
```

Richtig! Wir lesen die 768 Bytes nach Speicherstelle BILD, die wir ab Bildschirmposition >0000 ausgeben. Das Programm ist auf der Diskette!

3. Lösungen zum Kapitel 3:

1. Aufgabe:

Wiederholung von Ausgaben, Ausgabe von Teilen einer Zeichenkette.

```
0001          DEF   START
0002          REF   VMBW,VMBR
0003 SATZ     TEXT  'HALLELUJA'
0004 BUFF1    BSS   >9
0005          EVEN
0006 START    LI    R0,>0000
0007          LI    R1,SATZ
0008          LI    R2,>9
0009          BLWP  @VMBW
0010          LI    R1,BUFF1
0011          BLWP  @VMBR
0012          LI    R0,>0020
0013          BLWP  @VMBW
0014          LI    R0,>0040
0015          BLWP  @VMBW
0016          LI    R0,>0060
0017          BLWP  @VMBW
0018          LI    R0,>0080
0019          BLWP  @VMBW
0020          LI    R0,>00A0
0021          BLWP  @VMBW
0022          LI    R0,>00C0
0023          BLWP  @VMBW
0024          LI    R0,>00E0
0025          BLWP  @VMBW
0026          LI    R0,>0100
0027          BLWP  @VMBW
0028          LI    R0,>0120
0029          BLWP  @VMBW
0030          LI    R2,>5
0031          LI    R0,>0140
0032          BLWP  @VMBW
```

```
0033          LI    R0,>0160
0034          BLWP  §VMBW
0035          LI    R0,>0180
0036          BLWP  §VMBW
0037          LI    R0,>01A0
0038          BLWP  §VMBW
0039          LI    R0,>01C0
0040          BLWP  §VMBW
0041          LI    R0,>01E0
0042          BLWP  §VMBW
0043          LI    R0,>0200
0044          BLWP  §VMBW
0045          LI    R0,>0220
0046          BLWP  §VMBW
0047          LI    R0,>0240
0048          BLWP  §VMBW
0049          LI    R0,>0260
0050          BLWP  §VMBW
0051 LOOP     NOP
0052          JMP   LOOP
0053          END
```

Das Programm ist, wie Sie sehen, sehr lang geraten. Wenn es einen Befehl gäbe, mit dem man eine Schleife programmieren könnte...
Auch solche Befehle werden Sie noch kennenlernen!

Aufgabe 2:

Ein Laufschriftprogramm.

```
0001          DEF   START
0002          REF   VMBW,VMBR
0003 LOESCH   TEXT  '          '
0004 WORT     TEXT  ' HAGERA '
0005 BUFF1    BSS   >8
0006 START    LI    R0,>0177
0007          LI    R1,WORT
```

```
0008          LI    R2,>8
0009          BLWP  §VMBW
0010          LI    R1,BUFF1
0011          BLWP  §VMBR
0012 NEXT     LI    R0,>0176
0013          BLWP  §VMBW
0014          LI    R0,>0175
0015          BLWP  §VMBW
0016          LI    R0,>0174
  .
  .           und so weiter
  .
0037          LI    R0,>0160
0038          BLWP  §VMBW
0039          LI    R1,LOESCH
0040          BLWP  §VMBW
0041          LI    R1,WORT
0042          JMP   NEXT
0043          END
```

So schnell ist Maschinensprache...!
Diese Aufgabe finden Sie auch auf der beiliegenden Diskette.

4. Lösungen zum Kapitel 4:

Die Lösungen zu diesem Kapitel befinden sich auf der Diskette. Wir haben daher darauf verzichtet, das Listing abzudrucken. Hier ein paar erklärende Worte zu den Aufgaben:

1. Aufgabe:

Zunächst wieder die Initialisierung des Programms mit REF/DEF. Danach bauen wir eine Liste mit Byte-Daten auf und rufen diese durch VMBW auf. Zuletzt wird das ganze auf den Bildschirm ausgegeben.

2. Aufgabe:

Im Prinzip genauso zu lösen wie Aufgabe 1. Lediglich die zu verändernden Zeichen sind andere - und die Adresse muß verändert werden. Tatsächlich funktioniert es mit den Charactern 160 bis 255, denn der TI-Assembler verfügt über die Möglichkeit, diese Zeichen auszugeben.

5. Lösungen zum Kapitel 5:

1. Aufgabe:

Ändern der Bildschirmfarbe.

```
0001          DEF   COLOR
0002          REF   VWTR
0003 COLOR    LI    R0,>0709
0004          BLWP  @VWTR
0005 LOOP     NOP
0006          JMP   LOOP
0007          END
```

Das niedrigwertige Byte des Registers 0 muß mit >9 geladen werden, bevor Sie VWTR aufrufen.

2. Aufgabe:

Der Mischfarben-Effekt.

```
0001          DEF   COLOR
0002          REF   VWTR
0003 COLOR    LI    R0,>0709
0004          BLWP  @VWTR
0005          LI    R0,>0703
0006          BLWP  @VWTR
0007          JMP   COLOR
0008          END
```

Erstaunlich? Keineswegs. Die Farben wechseln so schnell, daß man es nicht sieht! Einfach die beiden Wechsel hintereinander schreiben und nach der zweiten Farbe zur ersten mit JMP zurückspringen.

6. Lösungen zum Kapitel 6:

1. Aufgabe:

Schreiben einer Farbänderungsroutine. Im Beispiel verwenden wir unser Menü-Beispiel aus Kapitel I.2.

```
0001        DEF  START
0002        REF  VMBW
*
Die Text-Anweisungen im Folgenden müssen
durch Leerzeichen so aufgefüllt werden,
daß jeweils 32 Zeichen in jeder Zeile
stehen!
*
0003 BILD   TEXT 'DIES IST EIN MENUE:'
0004        TEXT '                   '
0005        TEXT 'DRUECKEN SIE:      '
0006        TEXT '                   '
0007        TEXT '                   '
0008        TEXT '<1>  PROGRAMM 1    '
0009        TEXT '<2>  PROGRAMM 2    '
0010        TEXT '<3>  PROGRAMM 3    '
0011        TEXT '<4>  PROGRAMM 4    '
0012        TEXT '<5>  PROGRAMM 5    '
0013        TEXT '                   '
0014        TEXT '                   '
0015        TEXT '(C)  HAGERA  1985  '
0016        TEXT '                   '

weitere 10 Leerzeilen (wie 0016) anfügen!

0027        EVEN
0028 COLOR  DATA >F4F4,>F4F4,>F4F4
0029        DATA >F4F4,>F4F4,>F4F4
0030        DATA >F4F4,>F4F4,>F4F4
0031        DATA >F4F4,>F4F4,>F4F4
0032 PROG2  LI   R0,>0384
```

```
0033          LI    R1,COLOR
0034          LI    R2,24
0035          BLWP  §VMBW
0036          LI    R0,>070E
0037          BLWP  §VWTR
0038          LI    R0,>0000
0039          LI    R1,BILD
0040          LI    R2,768
0041          BLWP  §VMBW
0042 LOOP     NOP
0043          JMP   LOOP
0044          END
```

Auch hier könnte man sich mit einer Schleife einige Arbeit ersparen. Doch dazu später.

Aufgabe 2:

Als Beispiel sollen die Lösungen aus Kapitel I.4. gelten. Laden Sie den File von Diskette und überlegen Sie nocheinmal, was Sie alles ändern müssen. Wir wollen nicht immer gleich die Lösung verraten!

Aufgabe 3:

Diese Aufgabe haben Sie bereits gelöst - in diesem Kapitel nach den Erläuterungen zur Bildschirmfarbe. Probieren Sie es einmal mit anderen Farben!

Aufgabe 4:

Diese Aufgabe befindet sich auf der Diskette und wird dort ausreichend dokumentiert!

7. Lösungen zum Kapitel 9:

1. Aufgabe:

Warten auf einen Tastendruck, bis etwas geschieht.

```
0001          DEF   PROG7
0002          REF   VWTR,VMBW,VSBW,KSCAN
0003 GPLSTA   EQU   >837C
0004 FARBEN   TEXT  'TRANSPARENT    '
0005          TEXT  'SCHWARZ        '
0006          TEXT  'MITTELGRUEN    '
  .
  .
  .
0019          TEXT  'WEISS          '
0020 *
0021 PROG7    LI    R0,>0384
  .
  .
  .
0043          CLR   R5
0044 NEXT4    INC   R7
0045          MOVB  R5,@>8374
0046 TASTE    BLWP  @KSCAN
0047          MOVB  @GPLSTA,R5
0048          JEQ   TASTE
0050          DEC   R8
0051          JNE   NEXT2
0052          JMP   PROG7
0053          END
```

In Zeile 3 sehen Sie eine neue Anweisung, EQU. Mit EQU können Sie ein Symbol einer Speicherstelle zuordenen. Wenn Sie wie im Beispiel GPLSTA EQU >837C schreiben, können Sie fortan anstelle von >837C auch GPLSTA schreiben. Dies hat den Vorteil,

daß Sie sich keine Hexadezimaladressen zu merken brauchen, sonderen einfach ein sinnvolles Symbol dieser Adresse gleichsetzen. VMBW,KSCAN etc. sind ebenfalls solche vordefinierten Symbole, die bereits vom ROM her gleichgesetzt sind.

Im übrigen dürfte das Problem wohl keine Schwierigkeiten aufgegeben haben. Wenn nicht, sollten Sie sich nocheinmal das Kapitel TASTATURABFRAGE ansehen.

2. Aufgabe:

Eingeben von Zeichen auf den Bildschirm über die Tastatur.

```
0001          DEF   START
0002          REF   VSBW,KSCAN
0003 GPLSTA   EQU   >837C
0004 START    LI    R0,>0000
0005          LI    R5,768
0006 LOOP1    CLR   R6
0007          MOVB  R6,@>8374
0008 TASTE    BLWP  @KSCAN
0009          MOVB  @GPLSTA,R6
0010          JEQ   TASTE
0011          MOVB  @>8375,R1
0012          BLWP  @VSBW
0013          INC   R0
0014          DEC   R5
0015          JGT   LOOP1
0016          LI    R0,768
0017          LI    R1,>2000
0018 LOOP2    BLWP  @VSBW
0019          DEC   R0
0020          JGT   LOOP2
0021          JMP   START
```

```
0022          END
```

Klare Sache?! Zuerst kommt die Tastaturabfrage. Die gedrückte Taste wird in Register 1 kopiert. VSBW gibt dann den Wert aus Register 1 in die Bildschirmposition, die Register 0 beinhaltet. Zu Anfang ist dies >0000, mit jedem Tastendruck erhöht sich diese um 1. Gleichzeitig wird ein Zähler verringert, um nach voller Beschriftung des Bildschirms diesen zu löschen. Dies geschieht mit den Zeilen 0016 bis 0020. Das Leerzeichen wird geladen und 768 Mal ausgegeben. Ist der Bildschirm leer, springt das Programm zum Anfang zurück.

3. Aufgabe:

Wir brauchen Lösung 2 nur geringfügig zu ändern:

```
0001          DEF   START
0002          REF   VSBW,KSCAN
0003 GPLSTA   EQU   >837C
0004 START    LI    R0,>0168
0005          LI    R5,>000A
0006 LOOP1    CLR   R6
0007          MOVB  R6,@>8374
0008 CURSOR   LI    R1,>1E00
0009          BLWP  @VSBW
0010 TASTE    BLWP  @KSCAN
0011          MOVB  @GPLSTA,R6
0012          JEQ   TASTE
0013          MOVB  @>8375,R1
0014          BLWP  @VSBW
0015          INC   R0
0016          DEC   R5
0017          JGT   LOOP1
```

```
0018          LI    R5,>000A
0019          LI    R0,>0168
0020          LI    R1,>2000
0021 LOOP2    BLWP  @VSBW
0022          INC   R0
0023          DEC   R5
0024          JGT   LOOP2
0025          JMP   START
0026          END
```

Sie haben sich warscheinlich Gedanken darüber gemacht, wie man den Cursor zum Blinken bringt. Dazu müsste man, neben einer geeigneten Verzögerung, noch eine Routine programmieren, welche den Cursor zwischenzeitlich löscht. Für die Simulation eines Bildschirm-Accept soll jedoch obige Routine genügen.

8. Lösungen zum Kapitel 10:

1. Aufgabe:

Wiederholung der Aufgaben aus Kapitel 9 mit den Vergleichs-Instruktionen.

a) Warten auf einen Tastendruck, bis etwas geschieht. Wir haben hier keine Musterlösung abgedruckt. Versuchen Sie im Gegenteil einmal selbst, durch indirekte Adressierung und Vergleichsoberationen das Programm zu kürzen. Wenn Sie schwierigkeiten haben, schauen Sie sich die Befehle MOV und MOVB nocheinmal genau an!

b) Eingeben von Zeichen auf den Bildschirm über die Tastatur.

```
0001          DEF   START
0002          REF   VSBW,KSCAN
0003 GPLSTA   EQU   >837C
0004 KONST    DATA  >02FF
0005 START    LI    R0,>0000
0006 LOOP1    CLR   R6
0007          MOVB  R6,@>8374
0008 TASTE    BLWP  @KSCAN
0009          MOVB  @GPLSTA,R6
0010          JEQ   TASTE
0011          MOVB  @>8375,R1
0012          BLWP  @VSBW
0013          DEC   R0
0014          C     R0,@KONST
0015          JLE   LOOP1
0016          LI    R0,>0000
0017          LI    R1,>2000
0018 LOOP2    BLWP  @VSBW
```

```
0019          INC  R0
0020          CI   R0,>02FF
0020          JLE  LOOP2
0021          JMP  START
0022          END
```

Das Programm benutzt die Befehle C und CI. Dafür wird der Zähler für die Bildschirmposition eingespart. KONST ist eine DATA-Konstante mit dem Wert >02FF, was der letzten Bildschirmposition entspricht.

c) Für die Cursorausgabe brauchen wir Lösung b) nur geringfügig zu ändern:

```
0001          DEF  START
0002          REF  VSBW,KSCAN
0003 GPLSTA   EQU  >837C
0004 KONST    DATA >0172
0005 START    LI   R0,>0168
0006 LOOP1    CLR  R6
0007          MOVB R6,@>8374
0008 CURSOR   LI   R1,>1E00
0009          BLWP @VSBW
0010 TASTE    BLWP @KSCAN
0011          MOVB @GPLSTA,R6
0012          JEQ  TASTE
0013          MOVB @>8375,R1
0014          BLWP @VSBW
0015          INC  R0
0016          C    R0,@KONST
0017          JGT  LOOP1
0018          LI   R0,>0168
0019          LI   R1,>2000
0020 LOOP2    BLWP @VSBW
0021          INC  R0
0022          CI   R0,>0172
0023          JLE  LOOP2
```

```
0024          JMP  START
0025          END
```

Alles Klar? Falls nein, sollten Sie das Kapitel Verschieben und Vergleichen unbedingt wiederholen.

2. Vergleichen von Zahlen und Ausgabe der jeweils größten und kleinsten.

Eine Auflösung dieser Aufgabe wollen wir nicht geben. Aber vielleicht helfen Ihnen folgende Hinweise weiter:

1. Die Zahlen sollen Bytekonstante sein.
2. Benutzen Sie eine Schleife mit indirekter Adressierung.
3. Halten Sie die jeweils größte und kleinste gefundene Zahl in einem Buffer fest, zum Beispiel KLEIN und GROSS.
4. Nehmen wir an, die Zahlen im Buffer sind >19 und >46. Das entspricht Dezimal 25 und 70. Um sie umzuwandeln, muß ein entsprechendes Programm geschrieben werden. Denken Sie an die Sprung- und Vergleichsbefehle.

Übrigens: Nach der Lektüre des 11. Kapitels sollten die Schwierigkeiten, vor denen Sie im Augenblick noch stehen, keine mehr sein.

9. Lösung zum Kapitel 11:

Aufgabe:

Ergänzen eines Rechenprogrammes um eine Bildschirmausgabe.

Lösung: Wandeln Sie das Ergebnis wie in Aufgabe 2 des vorangegangenen Kapitels in eine Dezimalzahl um. Geben Sie dann die einzelnen Zeichen dieser Zahl mit VSBW an den Bildschirm ab.

Fragen zu den Lösungen?

Schreiben Sie uns!

5

V. Die TI-Assembler Mnemonics

In diesem Kapitel finden Sie eine kurze Übersicht über die TI-Assembler Mnemonics und deren Syntax. Soweit erforderlich, wurden kurze Beispiele in Form kleinerer Programme hinzugefügt.

Das Kapitel ist nach Befehlsgruppen unterteilt. Eine Übersicht über die Befehlsgruppen finden Sie am Ende dieses Kapitels.

Es werden jedoch nur die Mnemonics erwähnt, die durch den Einführungskurs bereits vermittelt wurden. Einige TI-Mnemonics, die für den fortgeschrittenen Programmierer sicher von Bedeutung sind, fehlen daher und können im TI-Assembler Handbuch nachgelesen werden.

Dieses Kapitel ersetzt in keinem Fall das Handbuch zum Editor/Assembler, sondern soll lediglich eine Zusammenfassung des Erlernten sein.

Die Abkürzungen in den Syntax-Beschreibungen stimmen mit denen des Handbuchs zum Editor / Assembler überein, um die Verständlichkeit zu erhöhen.

Abkürzungen in der Syntax:

<LABEL> markiert das Label-Feld
<KOMMENTAR> markiert das Kommentarfeld

b	ein oder mehrere Blanks
gas	Allgemeine Quelloperanden-Adresse
gad	Allgem. Bestimmungsoper.-Adresse
wa	Workspace-Registeradresse
iop	Direktoperand/Direktdaten
wad	Workspace Register Bestimmungs-Adr.
disp	Distanzadresse CRU-Zeile - CRU Basis-Register
exp	Ausdruck, der eine Befehlsstelle angibt
cnt	Zählen von Bits für CRU-Übertrag.
scnt	Verschiebt zähler

5.1

1. Arithmetische Instruktionen

Übersicht

A	Add words	Worte addieren
AB	Add Bytes	Bytes addieren
ABS	Absolute Value	Absolutwert
AI	Add Immediate	Unmittelb. add.
DEC	Decrement	Verringern um 1
DECT	Decrement by Two	Verringern um 2
DIV	Divide	Dividieren
INC	Increment	Erhöhen um eins
INCT	Increment by Two	Erhöhen um zwei
MPY	Multiply	Multiplizieren
NEG	Negate	Negieren
S	Subtract Words	Worte abziehen
SB	Subtract Bytes	Bytes abziehen

A - ADD WORDS

Syntax:

<Label> b A b gas,gad b <Kommentar>

Beispiel:

```
MAKEIT A    R1,§ORT2
```

addiert den Inhalt von Register 1 zum Inhalt von Adresse ORT2 und speichert das Ergebnis an Adresse ORT2.

Definition:

Addition von zwei Worten, wobei das Ergebnis der Addition das zweite Wort ersetzt.

Status Bit Benutzung:

Verglichen wird das Additionsergebnis (Summe) mit 0. Veränderungen sind daher möglich bei den Bits L>, A>, EQ, C und OV.

Ergebnis der Durchführung:

gas + gad => gad

AB = ADD BYTES

Syntax:

<Label> b AB b gas,gad b <Kommentar>

Beispiel:

MAKEIT AB R1,R4

addiert das linke Byte des Registers R1 zum linken Byte des Registers R4 und speichert das Ergebnis im linken Byte von Register R4.

Definition:

Addiert zwei Bytes, wobei die Summe an der Adresse gespeichert wird, an der sich das zweite Byte befindet.

Status Bit Benutzung:

Wie ADD WORDS. Benutzt werden L>, A>, EQ, C, OV und OP.

Ergebnis der Durchführung:

gas + gad => gad

DEC - DECREMENT

Syntax:

<Label> b DEC b gas b <Kommentar>

Beispiel:

MAKEIT DEC R1

subtrahiert vom Inhalt R1 eine binäre 1.

Definition:

Verminderung eines Wertes um eins.

Status Bit Benutzung:

Verglichen wird das Ergebnis der Subtraktion mit 0. Benutzt werden L>, A>, EQ, C und OV.

Ergebnis der Durchführung:

gas-1 => gas

DECT - DECREMENT BY TWO

Syntax:

<Label> b DECT b gas b <Kommentar>

Beispiel:

MAKEIT DECT §ZEIT

Subtrahiert vom Inhalt der Speicherstelle ZEIT zwei.

Definition:

Subtraktion von Zwei vom Inhalt einer Adresse.

Status Bit Benutzung:

Wie DECREMENT.

Ergebnis der Durchführung:

gas-2 => gas

DIV - DIVIDE

Syntax:

<LABEL> b DIV b gas,wad

Beispiel:

```
DIVIS1 DIV §DORT,R7
```

Teilt den Inhalt des Wortes in Register 7 und 8 durch den Inhalt des Wortes an Adresse DORT und liefert das Resultat in Register 7. Der Teilungsrest steht in Register 8.

Definition:

Division eines Registerinhaltes durch einen Wortinhalt. Das Resultat ist ganzzahlig im ersten Register, der Teilungsrest im zweiten (angegebenes+1).

Status Bit Benutzung:

OV.

Ergebnis der Durchführung:

wad UND wad+1 / gas => wad
Teilungsrest in wad+1

INC - INCREMENT

Syntax:

<Label> b INC b gas b <Kommentar>

Beispiel:

```
MAKEIT INC  >CA04
```

addiert den Inhalt von Adresse >CA04 um 1. Wenn z.B. in >CA04 der Wert >A3C2 gespeichert ist, so erhöht sich >A3C2 um 1 auf >A3C3.

Definition:

Addition eines Speicherinhaltes um 1.

Status Bit Benutzung:

Wie DECREMENT.

Ergebnis der Durchführung:

gas+1 => gas

INCT = INCREMENT BY TWO

Syntax:

<Label> b INCT b gas b <Kommentar>

Beispiel:

MAKEIT INCT R3

erhöht den Inhalt von R3 um 2.

Definition:

Addition eines Speicherinhaltes um 2.

Status Bit Benutzung:

Wie DECREMENT.

Ergebnis der Durchführung:

gas+2 => gas.

MPY - MULTIPLY

Syntax: <LABEL> b MPY b gas,gad

Beispiel:

```
MULTI MPY §FIRST,R4
```

multipliziert den Inhalt von FIRST zum Inhalt von Register 4. Das Ergebnis steht rechtsbündig in den 32 Bit von Register 4 und 5.

Definition:

Multiplizieren von Wort-Ausdrücken. Das Produkt ergibt einen 32-Bit-Ausdruck (2 Worte).

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

gas * wad => wad UND wad+1

NEG - NEGATE

Syntax:

<Label> b NEG b gas b <Kommentar>

Beispiel:

```
MAKEIT NEG  R4
```

ändert das Vorzeichen des Inhaltes von R4. Wenn in R4 der Wert >A342 Steht, so wird dieser auf >5CBE geändert (2er-Komplement):

```
Alter Wert:1010001101000010
Neuer Wert:0101110010111101
Plus  eins:0000000000000001
Negierung :0101110010111110
```

Definition:

Bildung des 2er-Komplements eines Speicher Inhaltes.

Status Bit Benutzung:

Verglichen wird das Negationsergebnis mit 0. Benutzt werden L>, A>, EQ und OV.

Ergebnis der Durchführung:

-gas => gas.

S - SUBTRACT WORDS

Syntax:

<Label> b S b gas,gad b <Kommentar>

Beispiel:

```
MAKEIT S    R4,R7
```

Subtrahiert den Inhalt von Register R4 vom Inhalt des Registers R7 und speichert das Ergebnis in Register R7.

Definition:

Subtraktion zweier Worte, wobei das Ergebnis das zweite Wort ersetzt.

Status Bit Benutzung:

Wie ADD WORDS.

Ergebnis der Durchführung:

gad - gas => gad

SB - SUBTRACT BYTES

Syntax:

<Label> b SB b gas,gad b <Kommentar>

Beispiel:

MAKEIT SB R4,R7

Subtrahiert das Linke Byte von Register R4 vom linken Byte in Register R7 und speichert das Ergebnis vom linken Byte in Register R7.

Definition:

Subtraktion von Bytes, wobei das zweite Byte durch das Subtraktionsergebnis ersetzt wird.

Status Bit Benutzung:

Wie ADD BYTES.

Ergebnis der Durchführung:

gad - gas => gad.

5.2

2. Sprünge und Verzweigungen

Übersicht

B	Branch	Verzweigung
BL	Branch and Link	Verzw./Verkettung
BLWP	Branch Load WP	Verzw./WP laden
JEQ	Jump if Equal	Sprung wenn gleich
JGT	Jump if Greater than	Sprung wenn größer als
JHE	Jump if High or Equal	Sprung wenn Höher oder gleich
JH	Jump if logical high	Sprung wenn logisch größer
JL	Jump if logical low	Sprung wenn logisch kleiner
JLE	Jump if Low or Equal	Sprung wenn kleiner oder gleich
JLT	Jump if Less than	Sprung wenn kleiner als
JMP	unconditional Jump	unbedingter Sprung
JNC	Jump if no Carry	Sprung wenn kein Übertrag
JNE	Jump if not Equal	Sprung wenn nicht gleich
JNO	Jump if no Overflow	Sprung wenn kein Überfluß
JOP	Jump if odd Parity	Sprung wenn Parität ungerade
JOC	Jump on Carry	Sprung wenn Übertrag
RTWP	Return with WP	Zurück mit WP

EX - EXecute und XOP - eXtended OPeration siehe TI-Assemblerhandbuch.

B - BRANCH

Syntax:

<Label> b B b gas b <Kommentar>

Beispiel:

```
LOS1    B     §ORT
```

verzweigt zur Instruktion an Speicherstelle ORT.

Definition:

Aufruf eines Programmteils an anderer Stelle.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

gas => PC

BL - BRANCH AND LINK

Syntax:

<Label> b BL b gas b <Kommentar>

Beispiel:

```
RUN2  BL   §ORT
```

ruft das Unterprogramm ORT als Workspace Unterprogramm auf.

Definition:

Unterprogrammaufruf.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

PC => R11
gas => PC

BLWP - BRANCH AND LOAD WORKSPACE POINTER

Syntax:

<Label> b BLWP b gas b <Kommentar>

Beispiel:

```
RUN3   BLWP §ORT
```

verzweigt zur Adresse ORT+2. In ORT und ORT+1 wird die alte Programmstelle gespeichert, um einen Rücksprung zu ermöglichen.

Definition:

Verzweigt zum Unterprogramm und speichert in den ersten beiden Speicherstellen die alte Programmstelle.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

```
WP => R13
PC => R14
ST => R15
gas => WP
gas+2 => PC
```

JEQ - JUMP IF EQUAL

Syntax:

<Label> b JEQ b exp b <Kommentar>

Beispiel:

```
RUN4   JEQ  §POS
```

verzweigt zu Speicherstelle POS, wenn im Statusbyte (Statusregister) das EQ-Bit auf 1 gesetzt ist.

Definition:

Sprung, wenn EQ=1.

Status Bit Benutzung:

Test von EQ. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn EQ=1 dann disp=> PC
b) Wenn EQ=0 dann keine Veränderung.

JGT - JUMP IF GREATER THAN

Syntax:

<Label> b JGT b exp b <Kommentar>

Beispiel:

```
RUN5   JGT   DORT
```

verzweigt nach DORT, wenn das A> Bit gesetzt ist (A> gleich 1).

Definition:

Verzweigung wenn A> = 1.

Status Bit Benutzung:

Test von A>. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn A>=1 dann disp=> PC
b) Wenn A>=0 keine Veränderung

JHE - JUMP IF HIGH OR EQUAL

Syntax:

<Label> b JHE b exp b <Kommentar>

Beispiel:

```
RUN6   JHE  ENDE
```

verzweigt nach ENDE wenn entweder das EQ-Bit oder das L> Bit oder beide Bits gesetzt sind.

Definition:

Verzweigung, wenn (EQ = 1 or L> = 1).

Status Bit Benutzung:

Test von EQ und L>. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn EQ=1 or L>=1 dann disp=> PC
b) Wenn EQ=0 and L>=0 keine Veränderung

JH - JUMP IF LOGICAL HIGH

Syntax:

<Label> b JH b exp b <Kommentar>

Beispiel:

```
RUN7   JH   WEITER
```

verzweigt nach WEITER, wenn das L> Bit gesetzt und das EQ Bit nicht gesetzt ist.

Definition:

Verzweigung, wenn (L> = 1 and EQ = 0).

Status Bit Benutzung:

Test von L> und EQ. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn L>=1 and EQ=0 dann disp=> PC
b) Wenn L>=0 or EQ=1 keine Veränderung

JL - JUMP IF LOGICAL LOW

Syntax:

<Label> b JL b exp b <Kommentar>

Beispiel:

```
RUN8   JL   BACK1
```

verzweigt nach BACK1 wenn sowohl das L> Bit als auch das EQ Bit (auf 0) zurückgesetzt ist.

Definition:

Verzweigung, wenn (L> = 0 and EQ = 0).

Status Bit Benutzung:

Test von L> und EQ. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn L>=0 and EQ=0 dann disp=> PC
b) Wenn L>=1 or EQ=1 keine Veränderung

JLE - JUMP IF LOW OR EQUAL

Syntax:

<Label> b JLE b exp b <Kommentar>

Beispiel:

```
RUN9   JLE  DAHIN
```

verzweigt nach DAHIN wenn das L> Bit auf 0 zurückgesetzt oder das EQ Bit auf 1 gesetzt ist.

Definition:

Verzweigung, wenn (L> = 0 or EQ = 1).
JLE bedeutet nicht 'Jump if Less or Equal'!

Status Bit Benutzung:

Test von L> und EQ. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn L>=0 or EQ=1 dann disp=> PC
b) Wenn L>=1 and EQ=1 keine Veränderung

JLT - JUMP IF LESS THAN

Syntax:

<Label> b JLT b exp b <Kommentar>

Beispiel:

```
RUN10  JLT  AWAY
```

verzweigt nach AWAY wenn sowohl das A> Bit als auch das EQ Bit (auf 0) zurückgesetzt ist.

Definition:

Verzweigung, wenn (A> = 0 and EQ = 0).

Status Bit Benutzung:

Test von A> und EQ. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn A>=0 and EQ=0 dann disp=> PC
b) Wenn A>=1 or EQ=1 keine Veränderung

JMP - JUMP (UNCONDITIONAL)

Syntax:

<Label> b JMP b exp b <Kommentar>

Beispiel:

RUN11 JMP >5B08

verzweigt nach >5B08, wenn die Adresse nicht weiter als >100 bytes von der augenblicklichen PC-Adresse entfernt ist.

Definition:

Verzweigung, wenn exp < ABS(PC - >100)

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

a) Wenn exp<=ABS(PC->100) dann disp=> PC
b) Wenn exp>ABS(PC->100) keine Veränderung

1.10

JNE - JUMP IF NOT EQUAL

Syntax:

<Label> b JNE b exp b <Kommentar>

Beispiel:

```
INPUT  JNE  OVER
```

verzweigt nach OVER, wenn das EQ bit (auf 0) zurückgesetzt ist.

Definition:

Verzweigung wenn EQ=0.

Status Bit Benutzung:

Test von EQ. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn EQ=0 dann disp=> PC.
b) Wenn EQ=1 keine Veränderung.

JNO - JUMP IF NO OVERFLOW

Syntax:

<Label> b JNO b exp b <Kommentar>

Beispiel:

```
INPUT1 JNO  DAHIN
```

verzweigt nach DAHIN, wenn das OV Bit (auf 0) zurückgesetzt ist.

Definition:

Verzweigung, wenn OV=0.

Status Bit Benutzung:

Test von OV. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn OV=0 dann disp=> PC.
b) Wenn OV=1 keine Veränderung.

JOP - JUMP IF ODD PARITY

Syntax:

<Label> b JOP b exp b <Kommentar>

Beispiel:

HIER JOP DORT

verzweigt nach DORT wenn das OP Bit gesetzt ist.

Definition:

Verzweigung, wenn OP=1.

Status Bit Benutzung:

Test von OP. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn OP=1 dann disp=> PC.
b) Wenn OP=1 keine Veränderung.

JOC - JUMP ON CARRY

Syntax:

<Label> b JOC b exp b <Kommentar>

Beispiel:

TEXAS JOC BONN

verzweigt nach BONN, wenn das C Bit gesetzt ist.

Definition:

Verzweigung, wenn C=1.

Status Bit Benutzung:

Test von C. Keine Veränderung.

Ergebnis der Durchführung:

a) Wenn C=1 dann disp=> PC.
b) Wenn C=0 keine Veränderung.

RTWP - RETURN WITH WORKSPACE POINTER

Syntax:

<Label> b RTWP b <Kommentar>

Beispiel:

```
ENDE   RTWP
```

kehrt von einem Unterprogramm zurück, welches mit BLWP aufgerufen wurde.

Definition:

Rückkehr zum Hauptprogramm nach BLWP.

Status Bit Benutzung:

Das Statusregister wird mit den Werten aus Register R15 gefüllt. Dieses Register speichert bei BLWP-Aufruf das Status-Register für den Rücksprung.

Ergebnis der Durchführung:

R13 => WP
R14 => PC
R15 => ST

5.3

3. Vergleichs-Instruktionen

Übersicht

C	Compare Words	Vergleiche Worte
CB	Compare Bytes	Vergleiche Bytes
CI	Compare Immediate	Unmittelbarer Vergleich
COC	Compare Ones Corresponding	Vergleiche Übereinstimmung einsen
CZC	Compare Zeros Corresponding	Vergleiche Übereinstimmung Nullen

C - COMPARE WORDS

Syntax:

<Label> b C b gas,gad b <Kommentar>

Beispiel:

HAGERA C R1,R2

vergleicht den Inhalt von Register 1 mit dem Inhalt von Register 2, ohne dabei eines dieser Register zu verändern.

Definition:

Vergleich von Worten und Änderung des Statusregisters.
Die C-Instruktion vergleicht die Operanden als Vorzeichen-Zahl, als 2er-Komplement und als Integer. Das Ergebnis hängt von den Operanden ab. Beispiele finden Sie im TI-Assemblerhandbuch, Seite 140.

Status Bit Benutzung:

L>, A> und EQ.

Ergebnis der Durchführung:

a) Wenn logisch gas>gad dann L> = 1.
b) Wenn arithmet. gas>gad dann A> = 1.
c) Wenn gas=gad dann EQ = 1.
d) Andernfalls wird (auf 0) zurückgesetzt.

CB - COMPARE BYTES

Syntax:

<Label> b CB b gas,gad b <Kommentar>

Beispiel:

```
SUCH   CB   R1,R5
```

vergleicht das linke Byte von R1 mit dem linken Byte von R5.

Definition:

Vergleich von Bytes ohne Änderung einer Speicherstelle. Das Ergebnis dteht im Statusregister (Vgl. 'C').

Status Bit Benutzung:

L>, A>, EQ und OP.

Ergebnis der Durchführung:

a) Wenn logisch gas>gad dann L> = 1.
b) Wenn arithmet. gas>gad dann A> = 1.
c) Wenn gas=gad dann EQ = 1.
d) Andernfalls wird zurückgesetzt.

OP wird gesetzt, wenn die Binären Werte der Operanden eine ungerade Anzahl an übereinstimmnden Einsen haben. Beispiele finden Sie im Assembler-Handbuch, Seite 142.

CI - COMPARE IMMEDIATE

Syntax:

<Label> b CI b wa,iop b <Kommentar>

Beispiel:

```
BILD   CI   R7,>C
```

vergleicht den Inhalt von R7 mit dem Wert >C.

Definition:

Vergleich eines Registerinhaltes mit einem bestimmten Wert. Speicherstellen werden nicht verändert. Das Ergebnis steht im Status-Register.

Status Bit Benutzung:

L>, A> und EQ.

Ergebnis der Durchführung:

a) Wenn logisch wa>iop dann L> = 1.
b) Wenn arithmet. wa>iop dann A> = 1.
c) Wenn wa=iop dann EQ=1
d) Andernfalls wird zurückgesetzt.

COC - COMPARE ONES CORRESPONDING

Syntax:

<Label> b COC b gas,wad b <Kommentar>

Beispiel:

```
NULL   COC  §LOOP,R5
```

vergleicht die Übereinstimmung der gesetzten Einsen des Inhaltes von R5 mit den gesetzten Einsen in LOOP. Die Adressen werden dabei nicht geändert.

Definition:

Vergleich, ob die gesetzten Einsen in gas mit denen in wad übereinstimmen.

Status Bit Benutzung:

EQ.

Ergebnis der Durchführung:

a) Wenn BIT(wad)<=BIT(gas) dann EQ=1
b) Wenn BIT(wad)>BIT(gas) dann EQ=0

CZC - COMPARE ZEROS CORRESPONDING

Syntax:

<Label> b CZC b gas,wad b <Kommentar>

Beispiel:

```
TBIT   CZC  §ALFA,R6
```

vergleicht die Übereinstimmung der gesetzten Nullen des Inhaltes von R5 mit den gesetzten Nullen in ALFa. Die Adressen werden dabei nicht geändert.

Definition:

Vergleich, ob die gesetzten Nullen in gas mit denen in wad übereinstimmen.

Status Bit Benutzung:

EQ.

Ergebnis der Durchführung:

a) Wenn BIT(wad)>=BIT(gas) dann EQ=1
b) Wenn BIT(wad)<BIT(gas) dann EQ=0

5.4

4. Kontroll- und CRU-INSTRUKTIONEN

Übersicht

LDCR	Load CRU	Lade Kommunikations-Register Einheit (CRU)
SBO	Set CRU Bit to 1	Setze CRU-Bit auf Eins
SBZ	Set CRU Bit to 0	Setze CRU-Bit auf Null
STCR	Store CRU	Speichere CRU
TB	Test (CRU-)Bit	Teste (CRU-)Bit

Die Mnemonics CKOF, CKON, IDLE, RSET und LREX werden in diesem Einleitungskurs nicht behandelt. Sie finden eine Erläuterung im TI-Assemblerhandbuch.

LDCR - LOAD CRU

Syntax:

<Label> b LDCR b gas,cnt b <Kommentar>

Beispiel:

```
SCHIEB LDCR §ORT1,9
```

überträgt 9 Bits von Adresse ORT1 in die Communication Register Unit. Der Übertrag beginnt mit dem niedrigwertigsten Bit.

Definition:

Übertrag von Bits von einer Adresse in die CRU. Dabei werden die niedrigwertigen Bits zuerst übertragen. Wenn cnt=0, werden 16 Bits übertragen. Mehr über CRU erfahren Sie im Kapitel _____ (Register).

Status Bit benutzung:

L>, A>, EQ und OP.

Ergebnis der Durchführung:

BIT(gas) => CRU.

SBO - SET CRU BIT TO ONE

Syntax:

<Label> b SBO b disp b <Kommentar>

Beispiel:

```
SCRUB  SBO  5
```

setzt CRU-Bit 5 auf 1.

Definition:

Setzen von CRU-Bits auf 1. Die CRU-Bits werden Relativ zur CRU-Basis-Adresse angesprochen, welche sich in Register 12, Bits 3 bis 14, befindet. Mehr dazu im Kapitel _____ (Register).

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

1 => BIT(CRU).

SBZ - SET CRU BIT TO ZERO

Syntax:

<Label> b SBZ b disp b <Kommentar>

Beispiel:

```
ZCRUB  SBZ  8
```

setzt CRU-Bit 8, entsprechend der Erläuterung zu SBO, zurück (auf 0).

Definition:

Zurücksetzen von CRU-Bits. Siehe SBO.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

0 => BIT(CRU).

STCR - STORE CRU

Syntax:

<Label> b STCR b gas,cnt b <Kommentar>

Beispiel:

```
OVER   STCR §ORT,>A
```

überträgt 10 Bits aus der CRU nach ORT.

Definition:

Übertragen von Bits aus der CRU zu einer bestimmten Adresse. Die Beschreibungen zu LDCR (Load CRU) gelten entsprechend.

Status Bit Benutzung:

L>, A>, EQ und OP.

Ergebnis der Durchführung:

BIT(CRU) => gas.

TB - TEST BIT

Syntax:

<Label> b TB b disp b <Kommentar>

Beispiel:

```
PROBE  TB   8
```

setzt das EQ-Statusbit gleich zum getesteten CRU-Bit.

Definition:

überprüfen eines CRU-Bit-Wertes.

Status Bit Benutzung:

EQ.

Ergebnis der Durchführung:

BIT(CRU) => EQ

5.5

5. Lade- und Verschiebe-Instruktionen

Übersicht

LI	Load Immediate	Unmittelbar laden
LIMI	Load Interrupt Mask Immediate	Interrupt-Maske unmittelbar laden
LWPI	Load Workspace Pointer Immediate	Workspace-Zeiger unmittelbar laden
MOV	Move Words	Worte verschieben
MOVB	Move Bytes	Bytes verschieben
STST	Store Status	Statusbyte sichern
STWP	Store Workspace Pointer	Workspace-Zeiger sichern
SWPB	Swap Bytes	Bytes vertauschen

LI - LOAD IMMEDIATE

Syntax:

<Label> b LI b wa,iop b <Kommentar>

Beispiel:

```
OVER   LI   R6,>4
```

lädt Register R6 mit dem Wert >4.

Definition:

Direktes laden von Daten in ein Register.

Status Bit Benutzung:

Der zu ladende Wert wird mit >0 verglichen.
L>, A> und EQ werden benutzt.

Ergebnis der Durchführung:

iop => wa.

LIMI - LOAD INTERRUPT MASK IMMEDIATE

Syntax:

<Label> b LIMI b iop b <Kommentar>

Beispiel:

INTER LIMI 2

erlaubt Interrupts von Level 0,1 und 2.

Definition:

Interrupt-Behandlung. Beachten Sie die Hinweis im Assembler-Handbuch.

Status Bit Benutzung:

Interrupt Mask.

Ergebnis der Durchführung:

Die letzten vier Bits von iop werden ins Status Register, Bits 12-15, übertragen.

iop => int. Mask.

LWPI - LOAD WORKSPACE POINTER IMMEDIATE

Syntax:

<Label> b LWPI b iop b <Kommentar>

Beispiel:

```
WPNEU  LWPI >034C
```

setzt WPNEU gleich zu >034C.

Definition:

Änderung des Inhaltes des Workspace-Pointer.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

iop => WP.

MOV = MOVE WORD

Syntax:

<Label> b MOV b gas,gad b <Kommentar>

Beispiel:

```
SCHIEB MOV  §WERT,R3
```

Definition:

Kopieren eines Wertes von gas nach gad. Die Verschiebung kann erfolgen von Speicherstelle nach Speicherstelle, von Speicherstelle in ein Register, von Register zu Register und von einem Register zu einer Speicherstelle.

Status Bit Benutzung:

Der neue Wert in gad wird mit >0 verglichen. L>, A> und EQ werden benutzt.

Ergebnis der Durchführung:

gas => gad

MOVB - MOVE BYTE

Syntax:

<Label> b MOVB b gas,gad b <Kommentar>

Beispiel:

```
SCHIEB MOVB R5,@>324B
```

kopiert die vier ersten Bit (1.Byte) von Register R5 nach Adresse >324B.

Definition:

Kopieren eines Byte von gas nach gad. Die Art und Weise der möglichen Benutzung ist dieselbe wie bei MOV (Move Words).

Status Bit Benutzung:

L>, A>, EQ und OP.

Ergebnis der Durchführung:

gas => gad

STST - STORE STATUS

Syntax:

<Label> b STST b wa b <Kommentar>

Beispiel:

SAVEST STST 3

kopiert den Inhalt des Statusregisters in Workspace-Register 3.

Definition:

Übertrag des Statusregisters in ein Workspace Register.

Status Bit Benutzung:

Keine Veränderung.

Ergebnis der Durchführung:

ST => wa

STWP - STORE WORKSPACE POINTER

Syntax:

<Label> b STWP b wa b <Kommentar>

Beispiel:

SAVEWP STWP 8

kopiert den Workspace Pointer Inhalt in Workspace Register 8.

Definition:

Kopieren des WP in ein Register.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

WP => wa

SWPB - SWAP BYTES

Syntax:

<Label> b SWPB b gas b <Kommentar>

Beispiel:

```
TAUSCH SWPB 5
```

vertauscht das erste Byte mit dem zweiten Byte in Register 5. Wenn der Inhalt von R5 >A325 ist, so wird er durch SWPB in >25A3 getauscht.

Definition:

Vertauschen von Bytes in einem Wort zur Durchführung von Byte-Operationen.

Status Bit benutzung:

Keine.

Ergebnis der Durchführung:

LEFT(gas) <=> RIGHT(gas)

5.6

6. Logische Instruktionen

Übersicht

ANDI	And Immediate	Direktes Und
ORI	Or Immediate	Direktes Oder
XOR	Exclusive Or	Ausschließliches od
INV	Invert	Invertieren
CLR	Clear	Löschen (=>0)
SETO	Set to one	Gleich zu 1 setzen
SOC	Set ones Corresponding	übereinstimmende Einsen setzen
SOCB	Set ones Corresponding; Byte	übereinstimmende Einsen setzen; Byte
SZC	Set Zeros Corresponding	übereinstimmende Nullen setzen
SZCB	Set Zeros Corresponding; Byte	übereinstimmende Nullen setzen; Byte

ANDI - AND IMMEDIATE

Syntax:

<Label> b ANDI b wa,iop b <Kommentar>

Beispiel:

```
UND    ANDI 5,>A472
```

Vorausgesetzt, der Inhalt von R5 ist >CB0F, dann vergleicht die Instruktion ANDI beide Werte Bit für Bit mit der AND-Operation und setzt R5 gleich zum Ergebnis, hier >8002.

```
Binär:     1100101100001111 (>CB0F in R5)
AND        1010010001110010 (>A472)
Ergebnis:  1000000000000010 (Ergebnis)
```

Definition:

AND-Operation auf Binärbasis.

Status Bit Benutzung:

L>, A> und EQ.

Ergebnis der Durchführung:

wa AND gas => wa

ORI - OR IMMEDIATE

Syntax:

<Label> b ORI b wa,iop b <Kommentar>

Beispiel:

```
ODER   ORI  5,>A472
```

Vorausgesetzt, der Inhalt von R5 ist >CB0F, dann vergleicht die Instruktion ORI beide Werte Bit für Bit mit der OR-Operation und setzt R5 gleich zum Ergebnis, hier >EF7F.

```
Binär:     1100101100001111 (>CB0F in R5)
OR         1010010001110010 (>A472)
Ergebnis:  1110111101111111 (Ergebnis)
```

Definition:

OR-Operation auf Binärbasis.

Status Bit Benutzung:

L>, A> und EQ.

Ergebnis der Durchführung:

wa OR gas => wa

XOR - EXCLUSIVE OR

Syntax:

<Label> b XOR b gas,wad b <Kommentar>

Beispiel:

```
ODER   ORI  §WORT,6
```

Vorausgesetzt, der Inhalt von WORT ist >CBOF und der Inhalt von R6 ist >A472, dann vergleicht die Instruktion XOR beide Werte Bit für Bit mit der XOR-Operation und setzt R5 gleich zum Ergebnis, hier >6F7D.

```
Binär:     1100101100001111 (>CBOF in R5)
OR         1010010001110010 (>A472)
Ergebnis:  0110111101111101 (Ergebnis)
```

Definition:

XOR-Operation auf Binärbasis.

Status Bit Benutzung:

L>, A> und EQ.

Ergebnis der Durchführung:

gas XOR wad => wad

INV - INVERT

Syntax:

<Label> b INV b gas,wad b <Kommentar>

Beispiel:

```
NICHT  INV  §WORT
```

Vorausgesetzt, der Inhalt von WORT ist >CB0F dann bildet die Instruktion INV Bit für Bit das NOT-Resultat und setzt R5 gleich zum Ergebnis, hier >34F0.

```
Binär:     1100101100001111 (>CB0F in R5)
NOT (INV)  0011010011110000 (>34F0)
Ergebnis:  0011010011110000 (Ergebnis)
```

Definition:

NOT-Operation auf Binärbasis.

Status Bit Benutzung:

L>, A> und EQ.

Ergebnis der Durchführung:

NOT(gas) => gas

CLR - CLEAR

Syntax:

<Label> b CLR b gas b <Kommentar>

Beispiel:

```
LOESCH CLR  R0
```

setzt R0 gleich zu >0000.

Definition:

16 Binäre Nullen werden in den Operanden geschoben.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

>0000 => gas

SETO - SET TO ONE

Syntax:

<Label> b SETO b gas b <Kommentar>

Beispiel:

SETZEN SETO §ORT

setzt ORT gleich zu >FFFF.

Definition:

16 Binäre Einsen werden in den Operanden geschoben.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

>FFFF => gas

CLR - CLEAR

Syntax:

<Label> b CLR b gas b <Kommentar>

Beispiel:

```
LOESCH CLR  R0
```

setzt R0 gleich zu >0000.

Definition:

16 Binäre Nullen werden in den Operanden geschoben.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

>0000 => gas

SETO - SET TO ONE

Syntax:

<Label> b SETO b gas b <Kommentar>

Beispiel:

```
SETZEN SETO §ORT
```

setzt ORT gleich zu >FFFF.

Definition:

16 Binäre Einsen werden in den Operanden geschoben.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

>FFFF => gas

SOC - SET ONES CORRESPONDING

Syntax:

<Label> b SOC b gas,gad b <Kommentar>

Beispiel:

OPODER SOC R4,§ORT

bildet die OR-Operation zwischen R4 und ORT.

Definition:

OR auf BIT-Basis zwischen zwei Operanden. Das Ergebnis steht in gad.

Status Bit Benutzung:

Das Ergebnis wird mit >0 verglichen. L>, A> und EQ werden benutzt.

Ergebnis der Durchführung:

gas OR gad => gad

SOCB - SET ONES CORRESPONDING; BYTE

Syntax:

<Label> b SOCB b gas,gad b <Kommentar>

Beispiel:

```
OPODER SOCB R4,§ORT
```

bildet die OR-Operation zwischen den ersten Byte von R4 und dem ersten Byte von ORT. Wenn in R4 >CBOF steht und in ORT >A472 so ist das Ergebnis >EF72, denn die jeweils letzten Bytes bleiben unberücksichtigt.

Definition:

OR auf BIT-Basis zwischen zwei Operanden, wobei jeweils nur das führende Byte beachtet wird. Das Ergebnis steht in dem führenden Byte von gad.

Status Bit Benutzung:

Das Ergebnis wird mit >0 verglichen. L>, A>, EQ und OP werden benutzt.

Ergebnis der Durchführung:

MSB(gas) OR MSB(gad) => msb(gad)

SZC - SET ZEROS CORRESPONDING

Syntax:

<Label> b SZC b gas,gad b <Kommentar>

Beispiel:

OPODER SZC R4,@ORT

vergleicht R4 und ORT auf Binärbasis mit der NOR-Operation, wobei übreinstimmende Einsen als Ergebnis BIN(0) ergeben.

```
Binär:      1100101100001111 (>CB0F in R4)
NOR         1010010001110010 (>A472 in ORT
Ergebnis:   0010010001110000 (>2470 in ORT
```

Definition:

NOR auf BIT-Basis zwischen zwei Operanden, hier durch Vergleich der Nullen. Das Ergebnis steht in gad.

Status Bit Benutzung:

Das Ergebnis wird mit >0 verglichen. L>, A> und EQ werden benutzt.

Ergebnis der Durchführung:

gas NOR gad => gad

SZCB - SET ZEROS CORRESPONDING; BYTE

Syntax:

<Label> b SZCB b gas,gad b <Kommentar>

Beispiel:

OPODER SZCB R4,@ORT

vergleicht R4 und ORT auf Binärbasis mit der NOR-Operation, wobei übereinstimmende Einsen als Ergebnis BIN(0) ergeben. Berücksichtigt werden nur die führenden Bytes der Operanden.

```
Binär:      1100101100001111 (>CB0F in R4)
NOR         1010010001110010 (>A472 in ORT)
Ergebnis:   0010010001110010 (>2472 in ORT)
```

Definition:

NOR auf BIT-Basis zwischen zwei Operanden, hier durch Vergleich der Nullen. Das Ergebnis steht in gad. Berücksichtigt werden jeweils nur die führenden Bytes.

Status Bit Benutzung:

Das Ergebnis wird mit >0 verglichen. L>, A>, EQ und OP werden benutzt.

Ergebnis der Durchführung:

MSB(gas) NOR MSB(gad) => gad

5.7

7. Workspace-Register Schiebeinstruktionen

Übersicht

SRA	Shift Right Arithmetic	Arithmetische Rechtsverschiebung
SRL	Shift Right Logical	Logische Rechtsverschiebung
SLA	Shift Left Arithmetic	Arithmetische Linksverschiebung
SRC	Shift Right Circular	Rechts-Rotation

SRA - SHIFT RIGHT ARITHMETIC

Syntax:

<Label> b SRA b wa,scnt b <Kommentar>

SHIFT SRA 5,4

verschiebt den Inhalt von R5 um vier Stellen nach rechts und füllt die freigewordenen Bitstellen mit dem Vorzeichenbit auf.

Definition:

Rechtsverschiebung von Bits in einem Register, wobei freiwerdende Stellen durch das Vorzeichenbit ergänzt werden.
Anmerkung: Wird die Anzahl der Verschiebungen mit 0 spezifiziert, so entspricht die wirkliche Anzahl dem Wert, der in den niedrigwertigsten Bits von R0 steht.

Status Bit Benutzung:

L>, A>, EQ und C.

Ergebnis der Durchführung:

Das Beispiel von oben vorausgesetzt, ergibt die Binärverschiebung folgendes, wenn in Register 5 der Wert >A2B1 steht:

```
1010001010110001  (Inhalt von R5)
1101000101011000  (nach 1. Verschiebung)
1110100010101100  (nach 2. Verschiebung)
1111010001010110  (nach 3. verschiebung)
.1111010001010l1  (Inhalt von R5, neu)
```

SRL - SHIFT RIGHT LOGICAL

Syntax:

<Label> b SRL b wa,scnt b <Kommentar>

Beispiel:

```
SHIFT  SRL  3,4
```

verschiebt den Inhalt von R3 um 4 stellen nach rechts, wobei die freigewordenen Stellen durch Nullen ersetzt werden.

Definition:

Rechtsverschiebung von Bits in einem Register wobei freiwerdende Stellen durch Nullen ersetzt werden. Für scnt=0 gelten die gleichen Regeln wie für SRA.

Status Bit Benutzung:

L>, A>, EQ und C.

Ergebnis der Durchführung:

Das Beispiel von oben vorausgesetzt, ergibt die Binärverschiebung folgendes, wenn in Register 3 der Wert >A2B1 steht:

```
1010001010110001   (Inhalt von R3)
0101000101011000   (nach 1. Verschiebung)
0010100010101100   (nach 2. Verschiebung)
0001010001010110   (nach 3. verschiebung)
0000101000101011   (Inhalt von R3, neu)
```

SLA - SHIFT LEFT ARITHMETIC

Syntax:

<Label> b SLA b wa,scnt

Beispiel:

```
DOPP   SLA 6,4
```

verschiebt den Inhalt von R6 um 4 Binärstellen nach links. Freiwerdende Stellen werden durch Binärnullen ersetzt.

Definition:

Linksverschiebung von Binärstellen, wobei freiwerdende durch Nullen ersetzt werden. Wenn sich das Vorzeichenbit während des Schiebevorgangs ändert, wird das OV-Statusbit gesetzt. Das C-Statusbit beinhaltet jeweils den Wert, der aus dem Register hinausgeschoben wird.

Status Bit Benutzung:

L>, A>, EQ, C und OV.

Ergebnis der Durchführung:

Das Beispiel von oben vorausgesetzt, ergibt die Binärverschiebung folgendes, wenn in Register 6 der Wert >A2B1 steht:

```
1010001010110001  (Inhalt von R3)
0100010101100010  (nach 1. Verschiebung)
1000101011000100  (nach 2. Verschiebung)
0001010110001000  (nach 3. verschiebung)
0010101100010000  (Inhalt von R3, neu)
```

OV und C werden gesetzt.

SRC - SHIFT RIGHT CIRCULAR

Syntax:

<Label> b SRC b wa,scnt b <Kommentar>

Beispiel:

```
ROTATE SRC  7,4
```

verschiebt den Inhalt von R7 um 4 Binärstellen nach rechts. Freiwerdende Stellen werden von den Bits ersetzt, die rechts aus dem Register hinausgeschoben werden.

Definition:

Verschiebung von Bits nach dem Rotationsprinzip. Für scnt=0 gelten die bekannten Regeln.

Status Bit Benutzung:

L>, A>, EQ und C.

Ergebnis der Durchführung:

Das Beispiel von oben vorausgesetzt, ergibt die Binärverschiebung folgendes, wenn in Register 7 der Wert >A2B1 steht:

```
1010001010110001  (Inhalt von R7)
1101000101011000  (nach 1. Verschiebung)
0110100010101100  (nach 2. Verschiebung)
0011010001010110  (nach 3. verschiebung)
0001101000101011  (Inhalt von R7, neu)
```

5.8

8. Pseudo-Instruktionen

Übersicht

NOP	No Operation	Keine Operation
RT	Return	Rückkehr

NOP - NO OPERATION

Syntax:

<Label> b NOP b <Kommentar>

Beispiel:

```
STOP   NOP
```

führt an dieser Stelle keine Operation aus. Die Wirkung ist dieselbe, als wenn man schreibt:

```
STOP   JMP  $+2
```

Definition:

Keine Operation.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

PC+2 => PC

RT = RETURN

Syntax:

<Label> b RT b <Kommentar>

Beispiel:

```
BACK   RT
```

bringt die Programmkontrolle aus einem Unterprogramm zum rufenden Programm zurück, wenn das Unterprogramm mit BL aufgerufen wurde. RT hat in diesem Zusammenhang dieselbe Bedeutung wie:

```
BACK   B
```

Definition:

Rückkehr zum Hauptprogramm von einem mit BL aufgerufenen Unterprogramm.

Status Bit Benutzung:

Keine.

Ergebnis der Durchführung:

wa => PC

6

VI. Besonderheiten des TI-Assemblers

Neben den Mnemonics (Programmbefehlen) verfügt der TI-Assembler über eine Reihe von weiteren Befehlen, welche die Programmierarbeit zum einen Erleichtern, aber auch für größte Programmeffizienz sorgen. Es handelt sich dabei zum einen um Direktiven, welche dem Prozessor des TI sagen können, wo er etwas ganz bestimmtes speichern soll. Nur die wesentlichen, die für die Einführung in die Assemblersprache notwendig sind, sollen in diesem Kurs besprochen werden. Alle anderen können ausführlich im TI-Assemblerhandbuch nachgelesen werden.

Weiterhin gibt es Utilities, die von Ihnen genutzt werden können. Meißtens handelt es sich dabei um Speicherstellen mit besonderen Aufgaben. Eine davon ist zum Beispiel das GPL-Status-Byte, welches Sie im Kapitel Tastaturabfrage kennengelernt haben. Außerdem gibt es eine Reihe von vordefinierten Symbolen, wie zum Beispiel GPLWS und SCAN, die viel Kopfarbeit ersparen, weil man sich Zutrittadressen nicht zu merken braucht.

Zuletzt verfügt der TI-Assembler über eine Reihe von eingebauten Unterprogrammen; die wichtigsten davon sind wohl VMBW, VSBW, VMBR und VSBR für die Bildschirmbehandlung, die unentbehrlich für die meißten Arbeiten am Computer sind.

6.1

1. Direktiven

Übersicht

AORG	Absolute Origin	Niedrigster Speicher
BSS	Block start with Symbol	Blockbeginn mit Symbol
EVEN	Word Boundary	Start an geradzahliger Adresse
EQU	Define Assembly Time constant	Assemblerzeitkonstante definieren
BYTE	Initialize Byte	Bytekonstante initial.
DATA	Initialize Word	Wortkonstante initial.
TEXT	Initialize Text	Textkonstante initial.
DEF	Ext. Definition	Externe Definition
REF	Ext. Reference	Externer Bezug
END	Program End	Programmende

Die Direktiven RORG, DORG, BES, PSEG, PEND, CEND, DSEG, DEND, LIST, PAGE, TITL, IDT, COPY, LOAD, SREF und DXOP werden ausführlich im TI-Assemblerhandbuch beschrieben.

AORG - ABSOLUTE ORIGIN

Syntax:

<Label> b AORG b wd-exp b <Kommentar>

Beispiel:

```
START  AORG >A000
```

veranlaßt den Assembler, Programme und Daten beginnend bei >A000 zu speichern.

Definition:

Placierung von Programmen und Daten ab einer spezifizierten Speicherstelle. Es wird so ermöglicht, einen Bestimmten Befehl oder bestimmte Daten an exakt feststehenden, also absoluten Adressen, zu speichern. Wenn AORG nicht gegeben ist, ist die einzige Möglichkeit, bestimmte Programmstellen anzuspringen, durch die Eingangsstellen (Label).

Ergebnis der Ausführung:

wd-exp => LC.

BSS - BLOCK STARTING WITH SYMBOL

Syntax:

<Label> b BSS b wd-exp b <Kommentar>

Beispiel:

```
BUFFER BSS  40
```

reserviert einen 40-Byte-Raum an Adresse BUFFER.

Definition:

Reservierung von Speicherstellen an bestimmten Stellen. BSS wird benötigt, wenn Datenmengen placiert gespeichert werden soll und ein direkter Zugriff auf einzelne Daten möglich sein soll.

Ergebnis der Ausführung:

LC => BUFFER
LC+wd-exp => LC

EVEN - WORD BOUNDARY

Syntax:

<Label> b EVEN b <Kommentar>

Beispiel:

```
NEXTD  EVEN
```

veranlaßt die Speicherung weiterer Daten und Programme ab der nächsten geradzahligen Adresse.

Definition:

Wie im Beispiel beschrieben. Sollte die augenblickliche Speicherstelle geradzahlig sein, so hat EVEN keine Bedeutung.

Ergebnis der Ausführung:

a) Wenn LC/2=INT(LC/2) dann LC => LC.
b) Wenn LC/2<>INT(LC/2) dann LC+1 => LC.

EQU - DEFINE ASSEMBLY-TIME CONSTANT

Syntax:

Label b EQU b exp b <Kommentar>

Beispiel:

```
STAT   EQU  >8375
```

ordnet den Wert >8375 dem Symbol STAT zu.

Definition:

Zuordnung eines Wertes zu einem Symbol. Nach Durchführung von EQU sind der Wert und das Symbol untereinander austauschbar, so daß für den Anwender das Merken bestimmter Speicherstellen entfällt. Man braucht lediglich ein Symbol, welches den Sinn des Wertes erkennen lässt. Fortan kann anstelle von >8375 auch STAT geschrieben werden.

Ergebnis der Durchführung:

symb = exp.

BYTE - INITIALIZE BYTE

Syntax:

<Label> b BYTE b exp,<exp,...> b <Kommentar>

Beispiel:

```
BKONST BYTE >00,>C2,>A-2,>F
```

initialisiert 4 Bytes, beginnend mit dem Byte an Speicherstelle BKONST.

Definition:

Bytes Initialisieren.

Ergebnis der Durchführung:

Siehe Beispiel.

DATA - INITIALIZE WORD

Syntax:

<Label> b DATA b exp<,exp,...> b <Kommentar>

Beispiel:

```
DKONST DATA >C0F0,>D040,'C'+2
```

initialisiert drei Wort-Ausdrücke, beginnend mit den Bytes an Speicherstelle DKONST.

Definition:

Initalisieren von Wort-Ausdrücken.

Ergebnis der Durchführung:

Siehe Beispiel.

TEXT - INITIALIZE TEXT

Syntax:

<Label> b TEXT b <->'string' b <Kommentar>

Beispiel:

```
WORT   TEXT 'Hagera'
```

initialisiert die ASCII-Werte von Hagera in den Speicherstellen beginnend bei WORT. Es ergibt sich die Reihenfolge >4861, >6765, >7261 Aus diesem Beispiel.

Definition:

Initialisierung von Text-Strings. Wenn das Minuszeichen gesetzt wird, negiert der Assembler das letzte Zeichen des Strings.

Ergebnis der Ausführung:

Siehe Beispiel.

DEF - EXTERNAL DEFINITION

Syntax:

<Label> b DEF b sym<,sym,...> b <Kommentar>

Beispiel:

```
DEFINE DEF  START,CLOAD,REMEM
```

veranlasst den Assembler, die Symbole START, CLOAD und REMEM in den Objektcode einzuschließen, damit auf sie von anderen Programmen, so auch mit CALL LINK von Extended Basic oder LOAD AND RUN PROGRAMNAME vom Editor Assembler, zugegriffen werden kann.

Definition:

Einschließen von Symbolen in den Objektcode. Mindestens die Eingangsstelle muß mit DEF zugänglich gemacht werden, wenn das Programm nicht über eine automatische Startvorrichtung verfügt, da es sonst nicht gestartet werden kann.

Ergebnis der Durchführung:

Siehe Beispiel.

REF - EXTERNAL REFERENCE

Syntax:

<Label> b REF b sym<,sym,...> b <Kommentar>

Beispiel:

```
BEZUG  REF  PILOT,OUTPR
```

veranlasst den Assembler, die Symbole PILOT und OUTPR in den Objektcode zu übernehmen, so daß die zu PILOT und OUTPR gehörenden Programme (Adressen) vom laufenden Programm angesprochen werden können.

Definition:

Einschluß fremder Programme in das laufende. Alle eingebauten Unterprogramme wie VWTR und VMBW können mit REF zugänglich gemacht werden (im Editor/Assembler).

Ergebnis der Durchführung:

Siehe Beispiel.

END - PROGRAM END

Syntax:

<Label> b END b <sym> b <Kommentar>

Beispiel:

```
PRENDE END  BEGINN
```

markiert das Programmende bei PRENDE. Alle nachfolgenden Anweisungen werden ignoriert. Wenn (wie im vorliegenden Beispiel) sym benutzt wird, startet das Programm nach dem Laden automatisch an der angegebenen Stelle, hier BEGINN. Der Eingangspunkt muß mit DEF definiert sein. Das Kommentarfeld kann nur benutzt werden, wenn auch das sym-Feld benutzt wird.

Definition:

Markieren des Programmendes (und Autostart).

Ergebnis der Ausführung:

Siehe Beispiel.

6.2

2. Vordefinierte Symbole und Utilities

Übersicht

UTLTAB	>2022	Startadresse Utilityvariable
PAD	>8300	Start 'CPU SCRATCH PAD RAM'
GPLWS	>83E0	GPL Workspace-Anzeiger
SOUND	>8400	Tongenerator-Register
VDPRD	>8800	VDP RAM Datenlese-Register
VDPSTA	>8802	VDP RAM Status Register
VDPWD	>8C00	VDP RAM Datenschreibe-Register
SCAN	>000E	Tastaturabfrage-Verzweigadr.

Diese und weitere wichtige Eingangsadressen werden im TI-Assemblerhandbuch beschrieben. Sie sind nur der Vollständigkeit halber hier aufgeführt.

Es gibt weitere vordefinierte Symbole für den Gebrauch von Sprache (Sprachsynthesizer).

Die hier genannten sind die wichtigsten.

6.3

3. Unterprogramm-Utilities

Übersicht:

VSBW	Einzelnes Byte ins VDP schreiben
VMBW	Mehrere Bytes ins VDP schreiben
VSBR	Einzelnes Byte vom VDP lesen
VMBR	Mehrere Bytes vom VDP lesen
VWTR	Einzelnes Byte in VDP-Register schreiben
KSCAN	Tastaturabfrage
GPLLNK	Grafikprogrammiersprachen-Verkettung
XMLLNK	ROM/RAM-Unterprogrammverkettung (mit der Konsole)
DSRLNK	Device Service Routine Verkettung
LOADER	Verkettung mit TMS9900-markierten Objekt Codes

DSRLNK ist nicht ins Extended Basic Modul eingebaut, während das GPLLNK so grundverschieden zum Editor/Assembler arbeitet, daß an dieser Stelle nicht weiter darauf eingegangen werden soll, weil eine umfassende Erläuterung die Aufgaben dieses Einführungskurses sprengen würde.

Die wichtigsten Unterprogramme werden im ersten Kapitel des Kurses ausführlich behandelt.

Eine ausführliche Übersicht über die Benutzung der vordefinierten Symbole und Utilities erhalten Sie im Assembler-Handbuch.

7

VII ERLÄUTERUNGEN ZUM TEXT

Bit: Eine Speicherstelle des Computers. Sie kann den Wert 1 (Strom) oder 0 (kein Strom) annehmen. Das Rechnen mit Bits entspricht somit dem Binären Zahlensystem.

Byte: 8 Bit sind ein Byte. Ein Byte kann 256 verschiedene Zustände haben (jedes Bit in einem Byte kann ein oder ausgeschaltet sein). Die höherwertigen Bits stehen dabei links, die niedrigwertigen rechts.

Beispiel: 10001010 Binär = Wert 138.

```
          ---------- 1x2   =   2
        ------------ 1x8   =   8
      ---------------- 1x128 = 128
                     =         138
```

Wort: 2 Bytes ergeben einen Wort-Ausdruck. Ein Wort-Ausdruck kann 65536 verschiedene Zustände annehmen. Einen Rechner, der über die Möglichkeit verfügt, gleichzeitig 16 Bits (2 Bytes a 8 Bit) von einer Speicherstelle zu einer anderen zu verschieben, heißt 16-Bit-Rechner. Der TI-99/4a ist ein solcher.

Nybble: In einem Byte ergeben jeweils die

vier höherwertigen und die niedrigwertigen Bits ein Nybble.

Übersicht:

0 ODER 1 = Bit
4 Bits = 1 Nybble
2 Nybble = 1 Bytes
8 Bits = 1 Byte
2 Bytes = 1 Wort
16 Bits = 1 Wort

Sie müssen beim Programmieren in Assembler stets unterscheiden zwischen:

- Bit-Operationen wie SLA,COC,TB
- Byte-Operationen wie CB,MOVB,AB
- Wort-Operationen wie MOV,C,A.

Ein Status oder Flag kann bereits in einem einzigen Bit enthalten sein. Jedes Zeichen, das Sie auf den Bildschirm ausgeben können, ist in einem Byte gespeichert. Daher können die ASCII-Zahlen z.B. immer mit einem Byteausdruck angegeben werden. Die Bezeichnung einer Speicheradresse hingegen ist eine 2-Byte-Zahl (ein Wortausdruck).

Register: Ein Register besteht aus einem oder mehreren Bytes und erfüllt bestimmte Aufgaben. Der TI-99/4a besitzt verschiedenartige Register.

a) Workspace Register sind 2-Byte-Register. Mit ihnen können wir zum Beispiel Parameter an die VDP-Unterprogramme weitergeben, aber auch Rücksprungadressen in Unterprogrammen sichern. Die Register werden auch für unmittelbare Operationen benötigt (AI, CI, LI, etc.). Der TI-99/4a verfügt über 16 WS-Register mit den Bezeichnungen R0-R15. WS-Register können gelesen und verändert werden.

b) Das Status Register beinhaltet das Ergebnis der letzten Instruktion. Anhand dieses Registers können wir Verzweigungen, Vergleiche und Sprünge vorschreiben. Welche Befehle das Status Register wie beeinflussen, ist im Assembler-Hanbuch genau beschrieben. Das Status Register ist ein 16-Bit Register und kann nur gelesen, aber nicht verändert werden.

c) Das GPL-Status Register ist ein 8-Bit Register mit besonderen Aufgaben im Grafik-Bereich. Es erfüllt verschiedene Aufgaben.

d) Die VDP-Register sind spezielle Register für den VDP-Speicherbereich. VDP-Register sind 8-Bit-Register und können beschrieben, aber nicht gelesen werden. VDP-Register 7 wird im Kapitel 'Farben bringen Leben' erläutert.

Es ist möglich, eigene Register für bestimmte Aufgaben zu definieren. Diese Möglichkeit wird jedoch in diesem Einführungskurs nicht behandelt.

Binärcode: Das Rechnen mit 0 und 1. Jeder Computer arbeitet mit diesem System, da es stets genau 2 Zustände beschreiben kann - 0 und 1. Nachfolgend einige Rechnungen im Binärsystem.

Addition: 01 + 01 = 10.
          01 + 11 = 100.

Subtraktion: 10 - 01 = 01.
           : 11 - 10 = 01.

Multiplikation: 11 x 10 = 110
              : 11 x 11 = 1001

Division   : 10 / 10 = 01
              : 11 / 01 = 11

Rechnen wir eine Binärzahl in eine 'normale' Dezimalzahl um, so müssen wir die Stellen der Binärzahl mit einer 2er-Potenz multiplizieren.

Beispiel: 101 = 1x1 + 2x0 + 4x1 = 5.
          011 = 1x1 + 2x1 + 4x0 = 3.
          111 = 1x1 + 2x1 + 4x1 = 7.

Hexadezimalcode: Da das Rechnen mit Binärzahlen für den Mensch sehr unübersichtlich wird, wenn es um große Zahlen geht, rechnet man besser mit dem Hexadezimalsystem. Hierbei werden jeweils 4 Bits zu einer Zahl zwischen 0 und 15 zusammengezogen. Die Zahlen von 10 bis 15 werden dabei durch die Großbuchtaben A bis F dargestellt.

Tabelle:

Binär	Dezimal	Hexadezimal
0000	0	>0
0001	1	>1
0010	2	>2
0011	3	>3
0100	4	>4
0101	5	>5
0110	6	>6
0111	7	>7
1000	8	>8
1001	9	>9
1010	10	>A
1011	11	>B
1100	12	>C
1101	13	>D
1110	14	>E
1111	15	>F

Dieses System wird zum Beispiel im Basic bei der Sonderzeichendefinition von CALL CHAR verwendet.

Nehmen wir an, wir stoßen auf die Hexadezimalzahl >3C. Dies bedeutet im dezimalen Zahlensystem 60 und im binären Zahlensystem 00111100.

2er-Komplement: Da wir negative Zahlen nicht ohne weiteres darstellen können, muß eine Vereinbarung getroffen werden. Zahlen gelten arithmetisch als negativ (<0), wenn das erste Bit gesetzt ist. Daraus ergibt sich, daß von jeweils 16 Binärzahlen genau die Hälfte als Negativ zu gelten hat. Betrachten wir nochmals unsere Tabelle:

Binär	Dezimal	Hexadezimal
0 000	0	>0
0 001	1	>1
0 010	2	>2
0 011	3	>3
0 100	4	>4
0 101	5	>5
0 110	6	>6
0 111	7	>7
1 000	- 8	>8
1 001	- 7	>9
1 010	- 6	>A
1 011	- 5	>B
1 100	- 4	>C
1 101	- 3	>D
1 110	- 2	>E
1 111	- 1	>F

Wir berechnen eine negative Zahl durch Umkehrung der Bitstellen (Negierung) und addition um 1:

```
Beispiel :   0010          = 2
Inverse  :   1101
Addition :   0001
Ergebnis :   1110          =-2
```

Label: Eine Programmeintrittsstelle. Mit einem Label kennzeichnen wir eine bestimmte Adresse in der Programmausführung. Ein Beispiel:

```
0005 ADR    LI R1,>4
```

kennzeichnet die Stelle im Programm, an der sich der Befehl LI... befindet, als ADR. Wenn von anderer Stelle im Programm fortan nach ADR verzweigt wird, so wird alles hinter ADR ausgeführt.

Mnemonic: Ein Assembler-Befehl. Um in Maschinensprache zu programmieren, benutzen wir statt der binären oder hexadezimalen Werte, die der Computer versteht, Mnemonics, die für uns leichter verständlich sind und die wir später 'assemblieren', also in einen für den Computer verständlichen Befehl umwandeln. Ein Beispiel:

INCT wird übersetzt in 0000010111000000 oder >05C0.

Operand: Operanden sind alles, womit der Maschinenbefehl arbeitet. Lautet der Befehl zum Beispiel LI R1,>5C13 so sind R1 und >5C13 Operanden.

Accumulator: Das 'Herz' des Computers. Hier werden alle Daten verarbeitet und ausgewertet. Siehe hierzu auch --> Register.

Indirekte Adressierung: Mit Hilfe besonderer Kennzeichen ist es möglich, im TI eine Speicherstelle indirekt anzuspringen, indem man zum Beispiel ein Register bezeichnet, welches die Speicherstelle enthält, die das zu verarbeitende Datenwort enthält. Dies klingt kompliziert, ist es aber nicht. Die Möglichkeiten zur indirekten Adressierung werden auf Seite 106 erläutert.

Terme: Anstelle eines Wertes kann oft

auch ein Ausdruck stehen. Wenn Sie zum Beispiel eine Eingangsstelle DORT haben, aber nicht zu DORT, sondern 2 Bytes weiter verzweigen wollen, können Sie anstelle eine weitere Eingangsstelle zu definieren auch schreiben:

DORT+2.

Wenn Sie irgendetwas nicht verstanden haben, schreiben Sie uns. Wir sind bemüht, diesen Kurs ständig zu verbessern.

8

8.1

DISKETTENFILES

Die beiliegende Diskette enthält einige Programme zu Studienzwecken sowohl im Objekt- als auch im Quellencode. Die Maschinenprogramme sind in komprimierter Form gespeichert. Zum Einlesen des Source-Files von Diskette benutzen Sie die LOAD-Funktion von der Editor-Auswahlliste.
Object-Files beginnen mit einem O. Die Programmeintrittsstelle lautet START bei allen Maschinenprogrammen, mit Ausnahme des Spiels OVILLAGEX. Dort lautet sie BEGIN1.

VILLAGEX .....Spiel
OVILLAGEX

AUFGI6 .....Aufgabe I.6.
OAUFGI6

AUFGII2 .....Aufgabe II.2.
OAUFGII2

AUFGIII2 .....Aufgabe III.2.
OAUFGIII2

AUFGIV1 .....Aufgabe IV.1.
OAUFGIV1

AUFGIV2 .....Aufgabe IV.2.
OAUFGIV2

AUFGV4 .....Aufgabe V.4.
OAUFGV4

Urheberrechtlicher Hinweis:

NACHWORT

Dies ist nun bereits die 3. Fassung unseres Assembler-Kurses, erstmals erschienen als gebundenes Buch. Daß er jetzt 344 statt bisher 312 Seiten umfasst, ist drucktechnisch bedingt.

Wir hoffen, mit dieser Änderung allen Benutzern eine Freude zu machen. Die Loseblattsammlung wird ab sofort nicht mehr ausgeliefert.

Der ASSEMBLER-KURS II wird demnächst durch ASSEMBLER KURS III ergänzt, eine Information darüber finden Sie auf den folgenden Seiten.

Wir hoffen, daß Ihnen dieser Band gefallen und einen leichten Einstieg in die Maschinensprache des TI-99/4a ermöglicht hat. Wenn Sie irgendwelche Fragen, Verbesserungsvorschläge oder Wünsche haben, schreiben Sie uns.

Mit freundlichen Grüßen

Rausch & Haub Vertriebs-GdbR.
Hans-Georg Rausch
Der Autor

MODE_CONTROL_II

Kennen Sie schon die 4 Arbeitsmodi des TI-99/4a?

- Den Grafik-Modus bestimmt, denn er ist aus dem Basic heraus ansprechbar. Dort haben Sie 24x32 Zeichenpositionen zur Verfügung.

Aber es gibt auch

- den Text-Modus mit 24x40 Zeichen, ideal für Verwaltungsprogramme aller Art;

- den Multicolor-Modus mit 48x64 Zeichenpositionen, z.B. für Diagramme, Grundrißzeichnungen etc.;

- den Bitmap-Modus, in dem alle 192x256 Bildschirmpixel einzeln ansprechbar sind.

Dies alles ist natürlich nur in Assembler möglich. 18 neue Maschinenbefehle, voll aus dem Basic/Extended Basic ansprechbar. Wahlweise für Extended Basic oder Editor Assembler Modul lieferbar.

NUR DM 39.90

ASSEMBLER-KURS III

Nach dem großen Erfolg von Kurs II ist es jetzt endlich soweit...

ASSEMBLER-KURS III für TI-99/4a

erscheint am 10. Dezember bei

Rausch & Haub Vertriebs-GdbR!!!

über 300 Seiten mit weiterführenden Informationen zum TI-Assembler, Systemadressen, nützliche Routinen, Zugriff aus dem Basic auf Maschinenprogramme und vieles mehr...

Beachten Sie unsere aktuellen Anzeigen in COMPUTER KONTAKT!!!

Zum Kurs die Diskette, die aus Ihrem Basic ein Superbasic macht - TORPEDO BASIC! Diese Diskette erweitert EXTENDED BASIC um 24 sagenhafte Befehle. Jetzt schon exemplare sichern (nur Bestellen, Bezahlung bei Auslieferung per Nachnahme ab 10. Dezember!).

ASSEMBLER KURS III	Buch)	DM 79.90
TORPEDO BASIC Diskette	dazu	DM 99.00

PROBLEMLÖSUNGEN

Haben Sie Softwaresorgen???

Wir lösen Ihre Softwareprobleme

- zu günstigen Preisen
- auf TI-99/4a oder SCHNEIDER CPC464
- auf allen Gebieten!

Melden Sie sich jetzt! Fordern Sie unsere Gratisinfo bezüglich AUFTRAGSARBEITEN an. Wir lösen Ihre Softwareprobleme!

Und auch dann, wenn Sie kein Programm schreiben lassen wollen, sind wir in Sachen COMPUTER für Sie die richtigen Ansprechpartner!

Entscheiden Sie sich rasch!

INFORMATIONEN

über unser reichhaltiges Programm an

- SOFTWARE
- HARDWARE
- ZUBEHöR

für

- TI-99/4a
- SCHNEIDER CPC464/664

erhalten Sie Gratis bei Rausch & Haub!
Bitte unbedingt System angeben!

Wer noch mehr wissen will, erfährt dies im aktuellen Gesamtkatalog (3.50 UKB, wird bei einer Bestellung verrechnet). Heute noch anfordern!

*- Thanks to 99'er User: **Alfredo Cevolini**, for the Scan of this document.*

- Reworked by TI99 Italian User Club (info@ti99iuc.it) - 2014