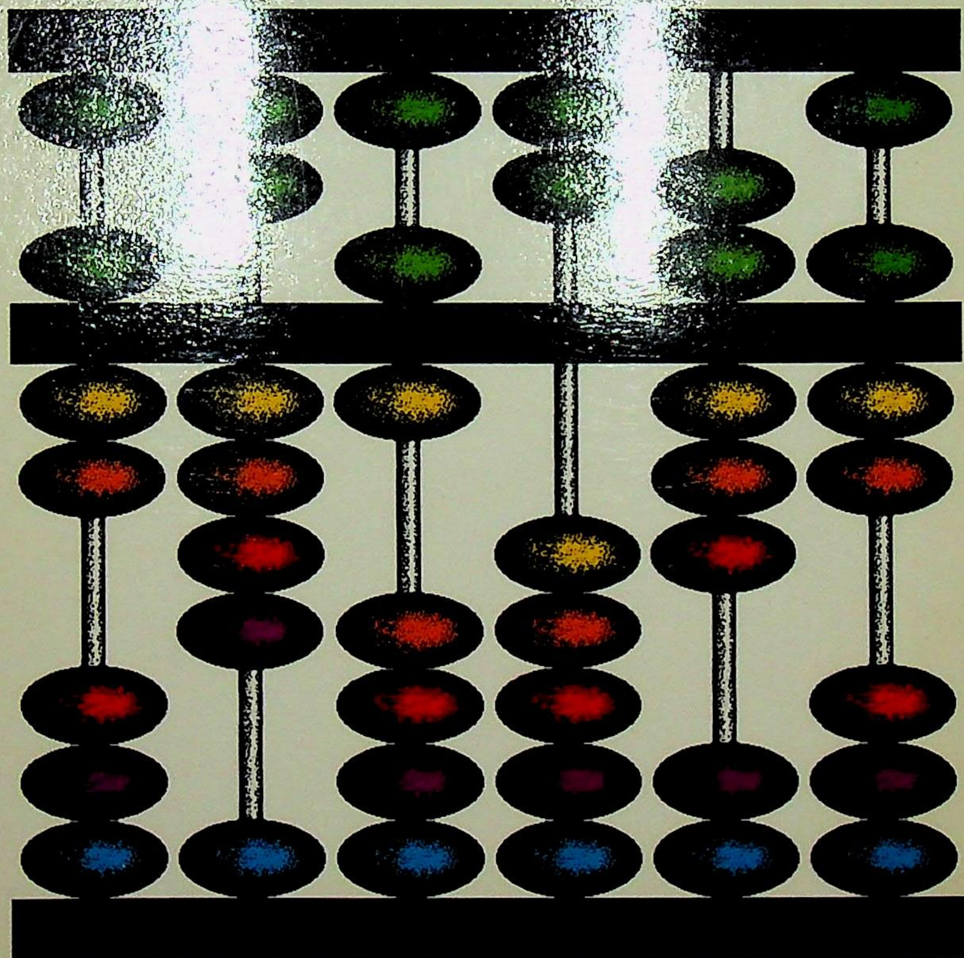
Visible Computer

von Charles Anderson



pandabooks

Visible Computer

Visible Computer

Charles Anderson

Pandabooks, Berlin

Titel der englischsprachigen Originalausgabe:
The Visible Computer: 6502
© Copyright 1982, 1984 by Software Masters

Deutsche Übersetzung: Botho Jung und Dietrich Wieczorek

Umschlagentwurf: Heike Freund

© Copyright 1985 Pandabooks GmbH

ISBN 3-89058-019-X

Printed in West Germany

Inhalt

Einleitung 7

Kapitel 1: Was ist Maschinensprache? 11

Kapitel 2: Alternative Zahlensysteme 15

Kapitel 3: Hardware 29

Kapitel 4: Starten des Programms 39

Kapitel 5: Das Arbeiten mit dem Speicher 47

Kapitel 6: Erste Programme 51

Kapitel 7: Das Prozessor-Status-Reister 61

Kapitel 8: Verzweigungen: Entscheidungen treffen 67

Kapitel 9: Adressierungsarten 71

Kapitel 10: Unterprogramme 75

Kapitel 11: Zwei nützliche Befehle: ADC/SBC 81

Kapitel 12: Weitere nützliche Befehle 89

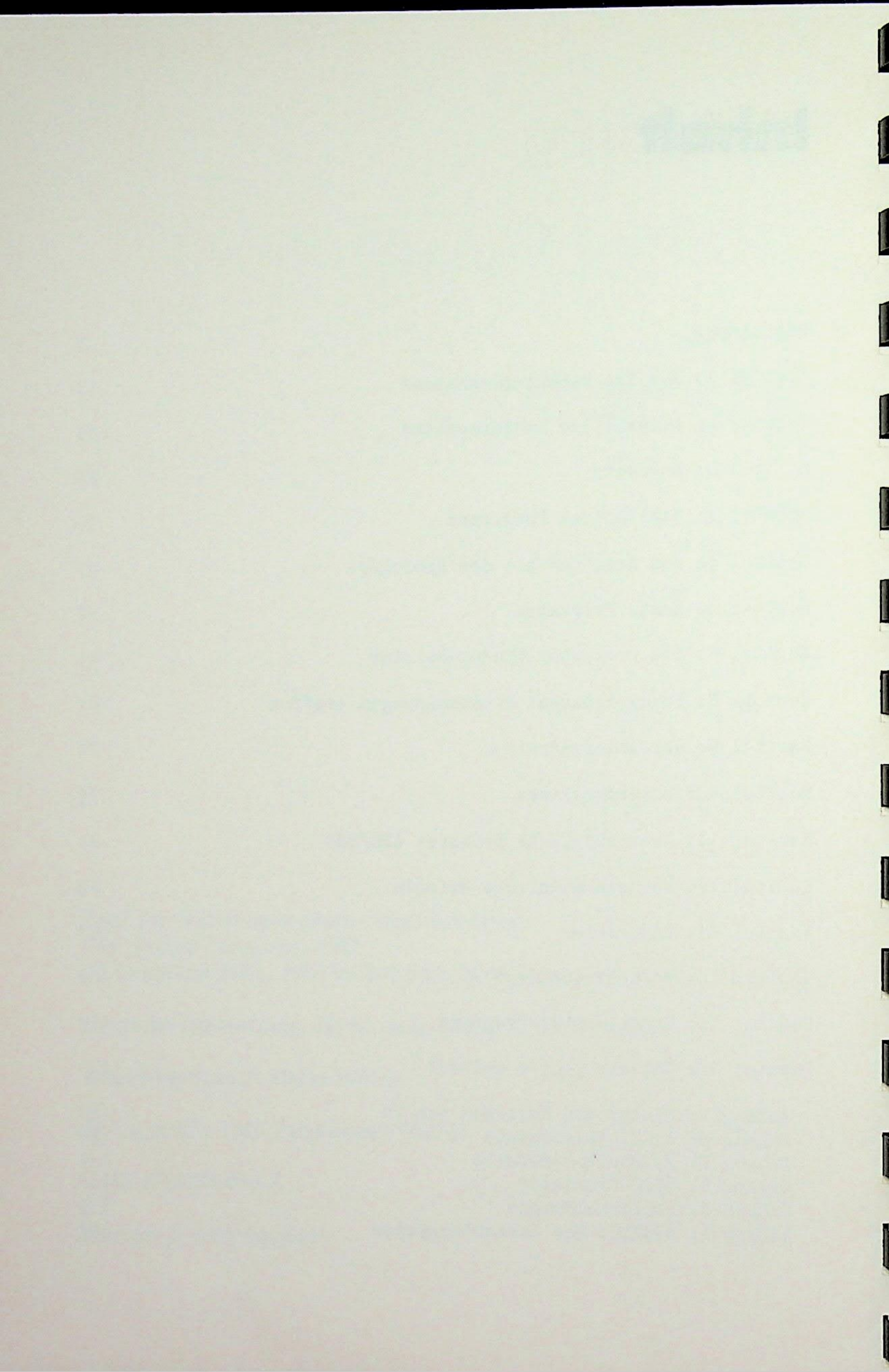
Kapitel 13: Indexieren 93

Kapitel 14: Verschiedenes 103

Kapitel 15: Unser erstes Programm 111

Kapitel 16: Und wie geht's weiter? 121

Anhang A: Hinter den Kulissen von VC 127
Anhang B: ASCII-Zeichensatz 129
Anhang C: VC Monitor-Befehle 131
Anhang D: 6502-Simulator 137
Anhang E: Fehlermeldungen 139
Anhang F: ASSYST: Das Assemblersystem 141



Einleitung

Das Maschinensprache-Lehrsystem VISIBLE COMPUTER besteht aus einem Buch und einer Diskette, die ein 6502-Simulatorprogramm enthält. Beides zusammen ermöglicht es Ihnen, die Programmierung in Maschinensprache auf einem APPLE II systematisch zu erlernen. Das Buch ist eine Einleitung in 6502-Maschinensprache und die eng damit verbundenen binären und hexadezimalen Zahlensysteme, während das Programm einen stark verlangsamten 6502-Mikroprozessor simuliert und damit eine eingehende Untersuchung seiner Arbeitsweise ermöglicht.

Voraussetzungen

Für die Benutzung dieses Buches wird davon ausgegangen, daß Sie mit der Programmiersprache BASIC vertraut sind. Programmierung ist Programmierung und je mehr Erfahrung Sie mit einer der gebräuchlichen Sprachen haben, desto besser. Darüber hinaus wird jedoch keinerlei Erfahrung mit Maschinensprache erwartet. Das vorliegende Buch enthält einleitende Kapitel über verschiedene Zahlensysteme und die Grundlagen der Hardware.

Erforderliche Hardware

Für das Simulatorprogramm brauchen Sie einen APPLE //e-, //c- oder II Plus-Computer (ältere APPLE II-Standardmodelle erfordern eine 16K-RAM- oder Language-Karte) und ein Diskettenlaufwerk, ein Drucker ist optional.

Ziel des Systems

Die meisten Bücher, die in die 6502-Maschinensprache einführen, sind, offenbar im Streben nach Vollständigkeit, mit endlosen Kapiteln über Gleitpunktarithmetik und Steuerprogrammen für hypothetische Typenraddrucker befrachtet. Dabei kommt die Darstellung der zugrunde liegenden Konzepte oft zu kurz. Bei VISIBLE COMPUTER wurde darauf Wert gelegt, Ihnen über die Anfangsschwierigkeiten der Programmierung in Maschinensprache hinwegzuhelfen, und nicht darauf, einen Algorithmus zu präsentieren, der ein Aufzugssystem steuern kann.

Wenn Sie sich die Kenntnisse dieses Buches angeeignet haben, werden Sie zwar noch nicht bei der Firma MICROSOFT COBOL Compiler schreiben können, Sie werden jedoch in der Lage sein, das Gelernte auf Ihrem Gebiet anzuwenden und weiterzuentwickeln, seien dies nun Bildschirmspiele, Schachprogramme oder neue, wundervolle Betriebssysteme. Und, wer weiß, vielleicht klappt es dann eines Tages doch noch bei MICROSOFT.

Über dieses Buch

In den Kapiteln 1, 2 und 3 werden eine Einführung in das binäre und hexadezimale Zahlensystem sowie das prinzipielle Blockdiagramm eines Computers präsentiert. Die ersten drei Kapitel können daher von denen übergangen werden, die bereits elf Diskussionen über Hex und Binär durchgestanden haben (das sind diejenigen, die schreien, wenn sie ein weiteres Blockdiagramm eines Computers sehen).

Das VC-Programm (VISIBLE COMPUTER) wird zwar erst in Kapitel 4 benutzt, Sie sollten jedoch bereits jetzt dort nachschlagen, um zu prüfen, ob Ihre Diskette korrekt läuft.

Den Kern des Kurses bilden die Kapitel 6 bis 14, in denen Sie immer schwieriger werdende 6502-Maschinenprogramme durcharbeiten werden, die auf der VC-Diskette enthalten sind. Am Ende von Kapitel 14 werden Sie nahezu jeden 6502-Befehl gelesen und demonstriert bekommen haben. Sie haben sich damit den ehrenvollen Titel eines VC-Meisters erworben.

Kapitel 15 zeigt Ihnen, wie Sie, ausgehend von einer Idee, über Design- und Codierphase, lauffähige Programme erstellen können. Außerdem wird die Rolle des Assemblers diskutiert.

Kapitel 16 enthält eine Liste gebräuchlicher Assembler, Tips zur Fehlersuche (Debugging) und Techniken, um BASIC- und Maschinenprogramme miteinander zu verbinden.

Bemerkungen zur zweiten (englischen) Ausgabe

Die vorliegende zweite Ausgabe von VISIBLE COMPUTER wurde gegenüber der ersten Ausgabe an folgenden Stellen verbessert:

Bessere Unterstützung des //e. Die erste Version von VISIBLE COMPUTER erschien im Dezember 1982, etwa einen Monat vor der Einführung des APPLE //e. Die vorliegende Version behandelt die geringen Unterschiede zwischen dem II Plus und dem //e nebeneinander. Trotzdem kann das Programm seinen ursprünglichen Zielrechner, den II Plus, auch weiterhin nicht verleugnen (nur Verwendung von Großbuchstaben und Benutzung der Links-Pfeil- anstelle der Delete-Taste).

Als diese Ausgabe in Druck ging, hielt der //c gerade seinen Einzug in die gute Stube. VC: 6502 läuft genauso gut auf einem //c, und Benutzer dieser Maschine werden 99,9% von dem, was sie in diesem Kursus lernen, auch auf ihrem Computer anwenden können.

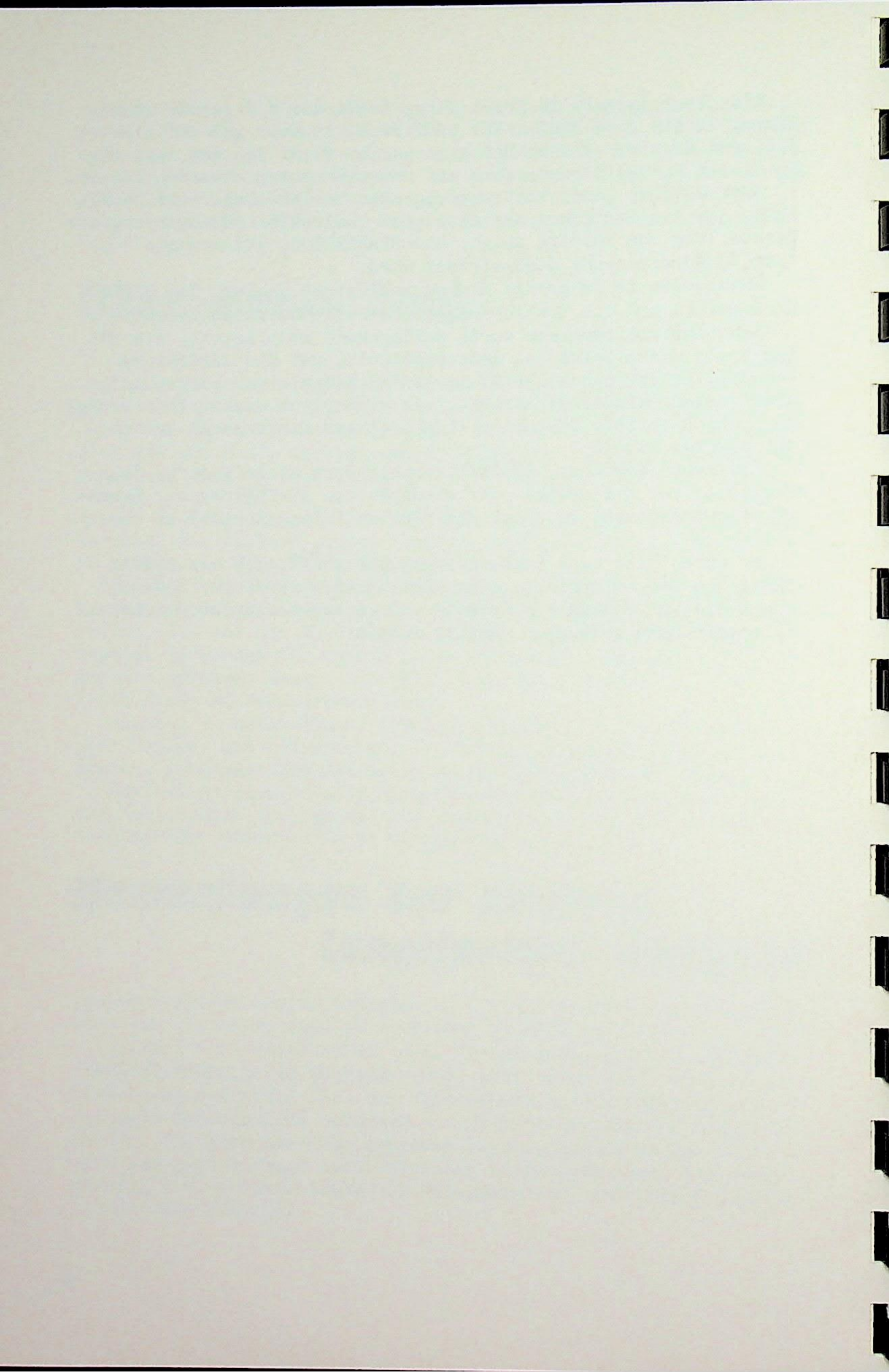
Zwei weitere Demonstrationsprogramme wurden eingefügt: COUNT-PROG, das die Benutzung der absoluten indirekten Adressierung zur Darstellung von Feldern zeigt, und BTABLEPROG, mit dem die (IND,X)-Adressierung demonstriert wird.

Änderungen im Programm: Ein geringfügiger Fehler, den UPPER CASE-Befehl und die Druckerausgabe betreffend, wurde beseitigt.

Das Simulatorprogramm wurde dahingehend modifiziert, ein BRK immer als einen Befehl zu interpretieren, der die Ausführung beendet. Früher durchlief VC im MASTER-Modus einen vollständig simulierten Hardware-Interrupt, was sicherlich einmal interessant zu beobachten ist, BRK als Hilfsmittel zur Fehlersuche jedoch unbrauchbar macht.

Ein neues Kommando, RESTART, erleichtert es Anfängern, den Prozessor bei der ersten oder wiederholten Ausführung der Demonstrationsprogramme in einen definierten Anfangszustand zu versetzen.

Es wurde eine neue Fehlermeldung BAD FNAME, die dem SYNTAX ERROR von DOS entspricht, eingefügt. Früher wurde der Benutzer durch die Verwendung von Kommata und gewisser anderer Zeichen in Filenamen dazu gezwungen, neu zu booten.



Kapitel 1
Was ist Maschinensprache?

Und wenn sie so schwierig ist, warum verwendet man sie? Dies ist eine berechtigte Frage, und wenn Sie sie sich noch nicht gestellt haben, so sollten Sie das jetzt tun. Bevor wir uns mit dem "Wie" der Maschinensprache befassen, wollen wir zunächst das "Warum" besprechen.

Obwohl jedermann von BASIC spricht, ist Maschinensprache das Idiom, das dem Siliziumherzen eines APPLE II am besten gerecht wird. In späteren Kapiteln dieses Buches werden wir eine mehr formale Definition entwickeln, im Moment reicht es festzustellen, daß Maschinensprache die Grundsprache eines jeden APPLE ist. Tatsächlich vergeht im Leben einer eingeschalteten 6502-Maschine kein Moment, in dem nicht Maschinensprache ausgeführt wird. Sprachen wie BASIC und PASCAL sind nichts als listige Kniffe, den armen Menschen vor dem unangenehmen binären Verhalten eines 6502-Prozessors zu bewahren.

Um dem weitverbreiteten Gerücht zu begegnen, daß Programmierung in Maschinensprache schwieriger ist als in BASIC, wollen wir uns die beiden folgenden Befehlsfolgen ansehen, die den Bau eines Holzzaunes in einem Hinterhof beschreiben:

BASIC

Baue mit 2m mal 15cm großen Holzlatten und Pfosten in 2,7m Abstand einen Zaun, der den Hof umschließt.

Maschinensprache

Fahre zur Holzhandlung. Kaufe 722 2m lange Holzlatten. Lade sie in den Wagen. Fahre nach hause. Entlade den Wagen. Beginne an der Nordostecke des Hofes. Grabe ein 1m tiefes Loch. Nimm einen Pfosten vom Haufen. Stecke den Pfosten in das Loch. Zementiere den Pfosten ein. Gehe 2,7m nach Westen. Wenn du noch nicht an der Ecke bist, grabe ein 1m tiefes Loch. Nimm einen Pfosten vom Haufen. Stecke den Pfosten in das Loch...

Ist die zweite Befehlsfolge schwieriger als die erste? Eigentlich doch nicht. Sie ist detaillierter, und man braucht länger, um sie niederzuschreiben, aber die einzelnen Aufgaben, die im zweiten Abschnitt beschrieben sind, sind die Einfachheit selbst: "Nimm einen Pfosten vom Haufen", "stecke den Pfosten in das Loch". Genau so ist es mit der Maschinensprache: Man arbeitet mit einer begrenzten Anzahl von etwa 50 einfachen Befehlen, kombiniert sie geschickt und erhält ein komplexes Ergebnis. Programmierer, die Maschinensprache verwenden, müssen kleinere Schritte machen, um an ihr Ziel zu gelangen. Das bedeutet, es dauert länger, ein Programm zu schreiben, länger, es zu entwerfen, zu codieren und die Fehler zu suchen. Über den Daumen gepeilt dauert es etwa zehnmal länger als in BASIC. Darüberhinaus kann nahezu alles, was in Maschinensprache programmierbar ist, auch in BASIC programmiert werden.

Warum also legen sich die Leute selber aufs Kreuz und lernen es mühsam, in Maschinensprache zu programmieren? Es gibt zwei wichtige Gründe: 1. Wegen der höheren Geschwindigkeit. 2. Wegen noch höherer Geschwindigkeit. Maschinenprogramme laufen etwa 10 bis 100 mal schneller ab als entsprechende Programme, die in BASIC geschrieben sind. (Puristen und andere Geizhälse werden etwas gegen diese Aussage einzuwenden haben, und tatsächlich muß dazu gesagt werden, daß es kein BASIC gäbe, auch nicht als Alternative, wenn nicht jemand ein Maschinenprogramm namens APPLESOFT geschrieben hätte.) Ist nun Geschwindigkeit so wichtig? Es kommt darauf an...

In einem Rechnungsprogramm, bei dem der Computer die meiste Zeit auf einen Tastendruck des Bedieners wartet oder darauf, daß der Drucker fertig wird oder das Diskettenlaufwerk die benötigten Daten gelesen hat, ist Geschwindigkeit nicht so wichtig. Man spricht dann von "druckerbegrenzt" (printer bound) oder "diskettenbegrenzt" (floppy bound). Ein Programm, das druckerbegrenzt ist, kann nur unter Verwendung eines schnelleren Druckers beschleunigt werden. Würde man ein Rechnungsprogramm in Maschinensprache schreiben, erhielte man ein Programm, das mit sehr hoher Geschwindigkeit auf Eingaben des Bedieners wartet und in der Entwicklung zehnmal so viel kostet wie ein mit akzeptabler Geschwindigkeit arbeitendes BASIC-Programm; auf diese Idee wird man also gar nicht erst kommen.

Bei anderen Problemen ist Geschwindigkeit jedoch erwünscht, manchmal sogar kritisch. Die meisten Bildschirmspiele würden in BASIC geschrieben nicht funktionieren, denn das Bewegen von Figuren auf dem Bildschirm erfordert die sorgfältig koordinierte Verschiebung von Tausenden von Zahlen und stellt somit außerordentlich hohe Anforderungen an ein Computersystem. Mit BASIC-Geschwindigkeit wären Spiele wie PAC MAN, mit sich schneckenhaft bewegenden Geistern, die Minuten dazu brauchen, von einer Seite des Irrgartens zur anderen zu gelangen, mustergültige Geduldsübungen. Bei Spielprogrammen, besonders bei den Bildschirmspielen, brauchen wir also die Geschwindigkeit der Maschinensprache.

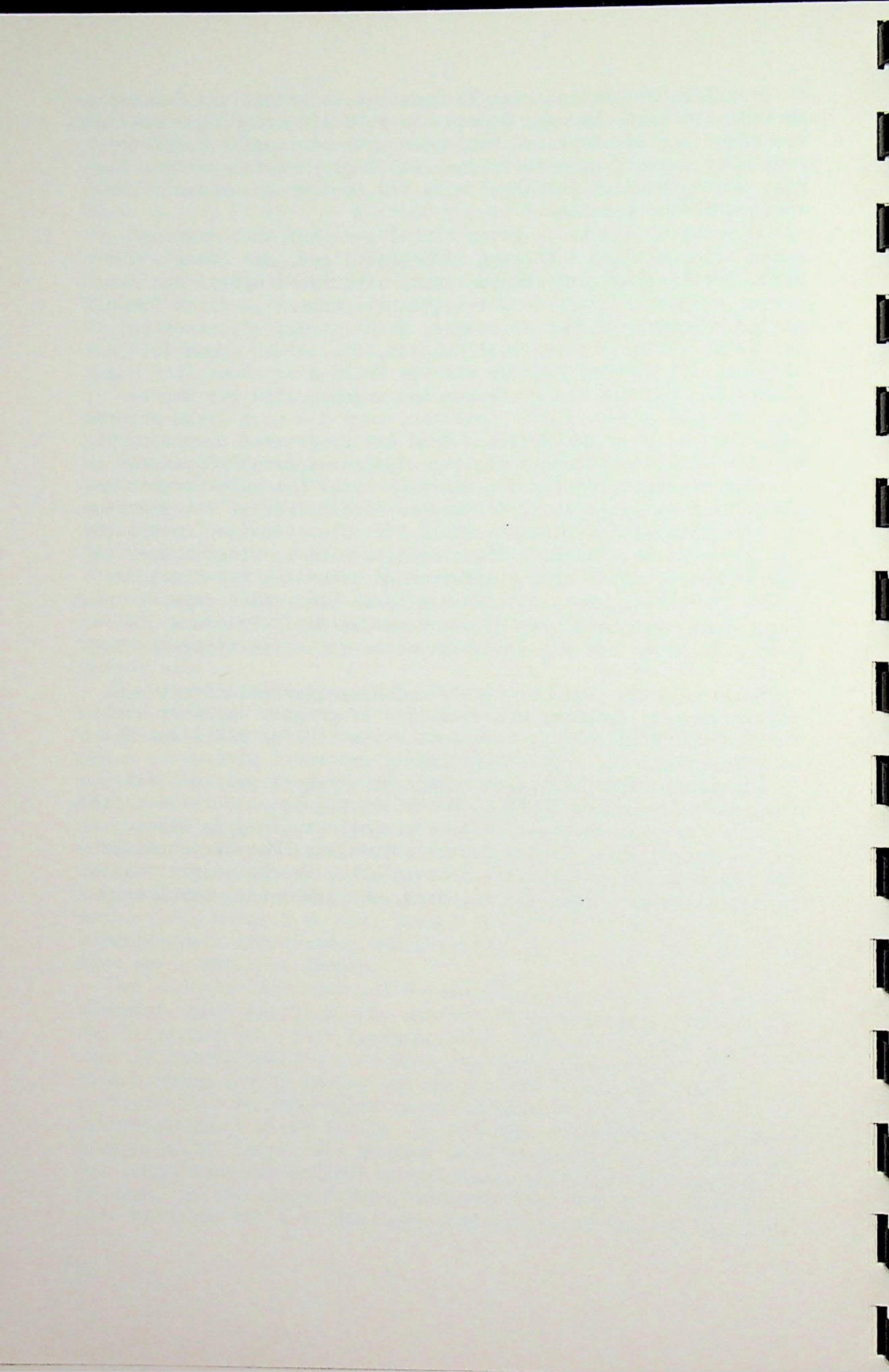
In vielen Fällen ist eine Kombination von BASIC und Maschinensprache die beste Lösung. Nehmen wir z.B. die Rechnungserstellung von eben: Der Hauptteil des Programms kann in langsam ausführbarem, aber schnell zu erstellendem BASIC geschrieben werden. Einige zeitaufwendige Aufgaben, z.B. das Sortieren, werden in Maschinensprache erledigt.

Denjenigen, die es in ihrer bisherigen Laufbahn vermieden haben, sich mit dem Sortieren zu beschäftigen, sei gesagt, daß BASIC-Sortierprogramme langsam sind, wirklich langsam! Das Sortieren einer Liste von 3000 Arbeitnehmernummern in einen Stapel mit der höchsten Nummer am unteren Ende und der kleinsten zuoberst dauert mindestens zwei Minuten, vielleicht sogar 10, je nachdem, mit welcher Methode wir das Problem angehen. (Die Möglichkeiten reichen von einfachen und ungehobelten bis hin zu komplexen Methoden. Viele Studenten, auch die noch nicht geborenen, werden ihren akademischen Grad mit Programmen erwerben, die 0,01% schneller sortieren als irgendwelche andere Programme.)

Zwei Minuten sind für den Bediener eines Geschäftsprogrammes eine verdammt lange Zeit, 10 Minuten eine Ewigkeit. Ein geschickter Programmierer wird daher BASIC für alles benutzen, nur nicht für das Sortieren selbst. Diese Aufgabe wird er einem schwer zu schreibenden, dafür aber atemberaubend schnellen Maschinenprogramm übertragen. Nach 10 Sekunden (oder einer oder zwei, je nach verwendeter Methode) präsentiert dann das BASIC-Programm die sortierte Liste der Angestelltennummern auf einem silbernen Tablett.

Arbeitsteilung zwischen BASIC und Maschinensprache ist eine vielverwendete Technik, wie unzählige Programme, darunter auch VISIBLE COMPUTER selbst, beweisen. Hauptsächlich BASIC und Maschinensprache da, wo Geschwindigkeit gebraucht wird.

Zusammenfassend läßt sich sagen: Der beste Grund, bei der Programmierung eines APPLE II Maschinensprache zu verwenden, ist, Prozesse zu beschleunigen, die andernfalls zu langsam wären. Anders ausgedrückt, es ist Zeitverschwendung, Maschinensprache für Dinge zu benutzen, die mit akzeptabler Geschwindigkeit auch in BASIC laufen würden (es sei denn, man macht dies zur Übung).



Kapitel 2
Alternative Zahlensysteme

Wenn Sie VISIBLE COMPUTER in der Hoffnung erworben haben, er würde Ihnen die Mühe abnehmen, den Hexadezimalberg zu ersteigen, er würde Sie stattdessen auf magische Weise hochheben und sicher im Tal der Maschinensprache auf der anderen Seite absetzen, so muß ich Sie nun enttäuschen.

Man benutzt binäre und hexadezimale Zahlen nicht, um die Programmierung in Maschinensprache leichter zu machen; man benutzt diese seltsamen Zahlensysteme, um Maschinensprache überhaupt erst zu ermöglichen. Obwohl nicht bestritten werden kann, daß es möglich ist, eine Maschinensprache zu erlernen, auch ohne sich mit Hexzahlen zu beschäftigen, wird jemand, der diesen Weg geht, feststellen, daß er es doppelt so schwer hat wie jemand, der zuerst die Werkzeuge des Gewerbes lernt und dann die Programmierung.

Eine Zwölf ist eine 12 ist eine 1100

Die meisten Leute sind sich darüber einig, daß die Symbole "1" und "2" so nebeneinandergeschrieben:

12

eine spezielle numerische Bedeutung haben, nämlich daß "12" die folgende Anzahl von Punkten repräsentiert:

....

Oder so viele Kommata:

,,,,,,,,,,,,

Es gibt jedoch nichts typisch "12-haftes" an diesen beiden nebeneinandergeschriebenen Symbolen. Wenn wir wollten, könnten wir einen Club günden, in dem "*" für 12 und "#" für 17 steht. Wir wollen dies einmal versuchen. Sie und ich sind priviligierte Mitglieder des "* = 12 und # = 17"-Clubs. Bis auf weiteres stellt "*" so viele Dinge dar:

,,,,,,,,,,,,

und "#" so viele:

,,,,,,,,,,,,,,,,,

Wie viele Eier sind ein Dutzend? Sehr gut, * ist richtig. Welche berühmte Gruppe hatte 1964 einen Hit mit dem Titel "She was Just #?" Wieder richtig, die BEATLES. (Ich habe Sie reingelegt - das Lied hieß "I Saw Her Standing There") Obwohl wir uns in den ersten Monaten sicher damit schwer tun würden, könnte man sich diese Darstellung sicher genauso angewöhnen wie die alte.

Anders sieht es aus, wenn wir uns mit Mathematik beschäftigen. Wieviel ist * mal #? Sogar für eingefleischte Clubmitglieder dürfte die Beantwortung dieser Frage nicht ganz leicht sein. Während die gewohnte Schreibweise "12 mal 17" wie von selbst zu Rechenregeln wie Übertrag und ziffernweiser Produktbildung führt, gibt uns unsere neue Schreibweise keinerlei Anhaltspunkte. Es bleibt uns daher nichts weiter übrig, als entweder alle möglichen Kombinationen bei der Multiplikation und Division mit * und # auswendig zu lernen oder die vergleichende Betrachtung für immer aufzugeben.

Diese Situation ist gar nicht so weit hergeholt, wie Sie vielleicht meinen. Denken Sie nur an das römische Imperium. Trotz aller Bildung war Roms Methode, Quantitäten darzustellen, das, was wir heute das römische Zahlensystem nennen. (Die Römer werden sie einfach Zahlen genannt haben.) Genau wie das Zahlensystem in unserem Club sind die römischen Zahlen gut für Dinge wie Papstnamen oder Bücherlisten, dagegen unbrauchbar, sobald es um Berechnungen geht.

Es ist direkt ein Wunder, was für Bauwerke sie errichtet haben, wenn man daran denkt, wie schwer sich ihre Ingenieure getan haben müssen, allein diese einfache Division auszuführen:

LXXIX / XIV

Denken Sie einen Moment darüber nach! Wenn Sie dieses Problem zu lösen hätten, würden Sie vermutlich folgendermaßen vorgehen: Übersetzung in "normale" Schreibweise, Ausführung der Division wie üblich, schließlich Rückübersetzung ins römische Zahlensystem. Unglücklicherweise gab es damals noch keine "normale" Schreibweise, denn erst 500 Jahre nach Christus ersann ein arabischer Astronom ein besseres System. Diese neue arabische Schreib-

weise war nicht nur besser geeignet zur Darstellung langer Zahlen, sie erleichterte auch arithmetische Berechnungen beträchtlich. Lassen Sie uns kurz untersuchen, warum die arabische Methode so überlegen ist: Zahlen, die in diesem System geschrieben sind, können systematisch in ihre Komponenten zerlegt werden.

Der Wert einer Ziffer wird von ihrer Position in der Zahl bestimmt; er ist immer zehn mal dem Wert derselben Ziffer eine Position weiter rechts und ein Zehntel des Wertes derselben Ziffer eine Position weiter links. Das folgende Diagramm zeigt, wie Zahlen in Ihre Bestandteile zerlegt werden.

vierte Ziffer	dritte Ziffer	zweite Ziffer	erste Ziffer
10^3	10^2	10^1	10^0
1000	100	10	1

3479 zerfällt in:

3 * 1000		4 * 100		7 * 10		9 * 1
3000	+	400	+	70	+	9

209 ergibt:

		2 * 100		0 * 10		9 * 1
		200	+	0	+	9

Wenn in früheren Zeiten Zahlensysteme erstellt wurden, so ergab sich deren Hauptschwäche jeweils daraus, daß nie die Notwendigkeit eingesehen wurde, ein Symbol für die Zahl Null einzuführen, für das Nichts. Ohne die Null als Platzhalter läßt sich aber keine positionsgebundene Darstellung aufbauen.

Die Nützlichkeit des positionsgebundenen arabischen Zahlensystems hat nichts zu tun mit dem Wert oder Unwert der Symbole, die das Zahlenalphabet aufbauen. 6 und 7 sind genauso gut oder schlecht wie V und X. Erst das Konzept der Positionierung macht das arabische System dem römischen überlegen. Im folgenden werden wir dieses geniale, uns vertraute Zahlensystem nicht das "arabische positionsgebundene" sondern das dezimale nennen (nach dem lateinischen Wort decem, zehn), da zehn der Wert ist, auf den jede Position fußt.

Andererseits ist die Anzahl

* * * * * * * * * *

im Universum in keiner Weise hervorgehoben, jedenfalls ist sie ebensowenig "rund" oder "gerade" wie diese Anzahl:

* * * * * * * * * * * *

Warum also benutzen wir 10 als den Basiswert in unserer positionsgebundenen Schreibweise? Hat jemand eine Idee? Richtig. Aller Wahrscheinlichkeit nach, weil Menschen 10 Finger haben, die für sie seit Millionen von Jahren das einzige Hilfsmittel waren, mit dem sich Zahlen darstellen ließen. Intelligente Lebewesen auf einem anderen Planeten mit 2 Händen und 4 Fingern an jeder Hand würden ihr positionsgebundenes Zahlensystem sicherlich auf der Zahl

* * * * * * * *

aufgebaut haben.

Das dezimale Zahlensystem blieb wegen seiner extrem nützlichen Art, Zahlen darzustellen, 1500 Jahre lang unverändert. Es wurden in dieser Zeit Rechenmethoden entwickelt, die es Zwölfjährigen gestatten, 5-stellige Quadratwurzeln nur mit Bleistift und Papier zu berechnen. Aller Voraussicht nach wird es auch noch weitere 1500 Jahre lang beliebt bleiben, obwohl seit dem Erscheinen von Taschenrechnern, die für wenige Mark erhältlich sind, einige seiner besten Anwendungsmöglichkeiten (die Einfachheit manueller Berechnungen) nicht mehr so wichtig sind. Hätten sich die römischen Zahlen in das Zeitalter der billigen Rechner hinübergerettet, so wären sie auch heute noch verbreitet.

Unglücklicherweise fällt das dezimale Zahlensystem bei der Programmierung von Computern, speziell in Maschinensprache, gründlich auf die berühmte Nase. Aufgrund ihrer Arbeitsweise haben Computer nur ein Vokabular von zwei Ziffern zur Verfügung. Es ist einfach, ein elektronisches Gerät herzustellen, das 1 und 0 speichern kann, dagegen aber schwierig, eines zu bauen, das 0 bis 9 speichert. Einen Sensor zu bauen, der feststellen kann, ob eine Glühlampe ein- oder ausgeschaltet ist, ist trivial. Der Bau eines Gerätes, das richtig zwischen 10 diskreten Helligkeiten unterscheiden kann, ist dagegen weit schwieriger.

Computer brauchen daher ein zweiziffriges oder binäres positionsgebundenes Zahlensystem. Jede vorstellbare Aufgabe, die von Computern gelöst wird, läßt sich auf die Manipulation von Einsen und Nullen zurückführen.

Genauso wenig wie wir im dezimalen Zahlensystem mehr als 10 Symbole zur Darstellung von Werten größer als 9 benötigen, brauchen wir bei binären Zahlen neue Symbole um die Werte 2 bis 9 darzustellen, da diese aus Kombinationen von Einsen und Nullen gebildet werden können.

Im folgenden Diagramm sind die ersten 4 Stellen des binären Zahlensystems dargestellt:

vierte Ziffer	dritte Ziffer	zweite Ziffer	erste Ziffer
2^3	2^2	2^1	2^0
8	4	2	1

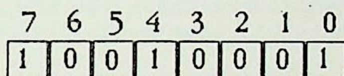
1010 zerfällt in:

1 * 8 + 0 * 4 + 1 * 2 + 0 * 1 = 10 dez.

1110 ergibt:

1 * 8 + 1 * 4 + 1 * 2 + 0 * 1 = 14 dez.

Bei der Programmierung von Kleincomputern enthalten die verwendeten binären Zahlen meistens 8 oder 16 Ziffern. Die einzelnen Ziffern oder Bits (ein Kunstwort aus binary digits = binäre Ziffern) werden, wie im Diagramm gezeigt, von rechts nach links durchnummeriert, wobei wir, wie so oft bei der Verwendung von Maschinensprache, nicht bei eins, sondern bei null zu zählen beginnen. Bit 0 wird oft das niedrigst wertige Bit (LSB = least significant bit) genannt; entsprechend heißt Bit 7 das höchstwertige Bit (MSB = most significant bit).



Es ist weiterhin üblich, führende Nullen bei der Arbeit mit binären Zahlen mitzuschreiben (z.B. wird 101 oft als 00000101 geschrieben).

Binäre Zahlen können mit derselben Methode addiert und subtrahiert werden, die wir vom Dezimalsystem kennen:

```
                1          11     <---  Überträge (Carrys)

  1010        1010        1001
+ 0100      + 0010      + 0011
  ----        ----        ----
  1110        1100        1100
```

Abgesehen davon, daß es in binären Additionen viele Überträge und in Subtraktionen viele Unterträge (borrows) gibt, ist an binärer Arithmetik nichts besonders aufregendes Neues.

"Runde Zahlen" im binären Zahlensystem sind die Zweierpotenzen: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096... Diesen Werten werden Sie in ihrer Laufbahn als Maschinensprachprogrammierer immer wieder begegnen; Sie sollten sich mit ihnen vertraut machen.

Es folgt nun eine Formel (die einzige in diesem Buch), mit der man den größten Wert X berechnen kann, den man mit n Symbolen in einem Zahlensystem zur Basis B erhalten kann:

$$X = B^{n-1}$$

Der größte Wert, den man mit drei dezimalen Ziffern darstellen kann ist 999 ($10^3 - 1$). Mit der 7 als Basis läßt sich durch drei Ziffern maximal der Wert 342 ($7^3 - 1$) darstellen. Bei den binären Zahlen schließlich wird schon die Darstellung bescheidener Werte zur Papierverschwendung; z.B. das Zählen bis 10:

```
   0
   1
  10
  11
 100
 101
 110
 111
1000
1001
1010
```

Zur Handhabung des Zahlenbereiches, mit dem wir im täglichen Leben umgehen, brauchen wir eine Menge binäre Ziffern:

365 = 101101101

$$1*2^8+0*2^7+1*2^6+1*2^5+0*2^4+1*2^3+1*2^2+0*2^1+1*2^0$$

531 = 1000010011

$$1*2^9+0*2^8+0*2^7+0*2^6+0*2^5+1*2^4+0*2^3+0*2^2+1*2^1+1*2^0$$

Zahlen wie 1000010011 und 101101101 tendieren dazu, die Menschen zu verwirren. Es hilft ein wenig, wenn wir die binären Ziffern, die Bits, in gleich große Gruppen zusammenfassen. Üblich sind Gruppen von 4 Bits, Nibbles genannt, und Gruppen von 8 Bits, die man Bytes nennt. Schreibt man 531 in Nibbleform, ergibt sich 0010 0001 0011. Das ist zwar etwas besser, aber nicht viel.

Natürlich bedeutet die Tatsache, daß Computer intern nur binäre Zahlen verarbeiten, nicht, daß die Menschen, die Computer benutzen, auch nur mit Nullen und Einsen umgehen müssen. BASIC-Programmierer z.B. sind in der glücklichen Lage, in einer Welt arbeiten zu können, die von dezimalen Zahlen beherrscht wird. Demgegenüber muß man bei Verwendung von Maschinensprache mit dem Rechner häufig auf binärer Ebene arbeiten.

Ein Weg, das Problem "Computer lieben binäre Zahlen, aber die Menschen lieben die dezimalen" zu lösen, ist, die Menschen wei-

terhin dezimal denken zu lassen und die Zahlen in das oder aus dem binären Zahlensystem zu konvertieren, wenn das zur Kommunikation mit der Maschine notwendig ist. Das hört sich wie eine gute Idee an, aber die Konvertierung zwischen binärem und dezimalem Zahlensystem ist eine, wie die Mathematiker sagen, nicht-triviale Aufgabe. Dabei ist der Übergang vom Binären ins Dezimale noch relativ einfach; wir werden daher damit anfangen.

Nimm eine binäre Zahl. Beginne mit der am weitesten links stehenden Ziffer (dem MSB) und summiere die Dezimaläquivalente einer jeden Position:

Position:	5	4	3	2	1	0	
Dezimaläquivalent:	32	16	8	4	2	1	
0000 1001 ergibt			8 +			1	= 9
0001 1010 ergibt		16 +	8 +		2		= 26
0010 0101 ergibt	32 +			4 +		1	= 37

Wenn Sie sich die Zweierpotenzen merken können und einfach zusammenzählen, so können Sie binäre Zahlen ins Dezimale konvertieren.

Dezimal nach binär

In dieser Richtung umzuwandeln ist schwieriger, es läuft auf eine vereinfachte Art der Division hinaus.

Umwandlung von 21 dezimal nach binär:

		7	6	5	4	3	2	1	0
Die größte Zweierpotenz, durch die man 21 teilen kann, ist 16.	21/16 = 1 Rest = 5	0	0	0	1				
Als nächstes versuche, den Divisionsrest durch die nächst kleinere Zweierpotenz zu teilen.	5/8 = 0 Rest 5				1	0			
... und so weiter bis der Divisionsrest Null ergibt.	5/4 = 1 Rest = 1				1	0	1		
	1/2 = 0 Rest = 1				1	0	1	0	
	1/1 = 1 Rest = 0				1	0	1	0	1

21 dezimal ergibt 10101 binär

Üben Sie diese Umwandlung nur so lange, bis Sie das Prinzip verstanden haben. Sie können Rechner kaufen, die auf Basisumwandlungen spezialisiert sind. Jedoch selbst mit solchen Rechnern ist der Abgrund zwischen den ins Binäre verliebten, papierverschwendenden Computern und den mit 10 Fingern versehenen, Bäume liebenden Menschen meilenweit. Zahlen, die im Binären sehr "rund" sind, wie 1000 0000 0000, erscheinen uns im Dezimalen eher ungerade (2048) und umgekehrt: 2000 dezimal ist 0111 1101 0000 binär. Glücklicherweise gibt es eine Brücke, hexadezimal genannt, über diesen dezimal/binären Abgrund.

Nehmen wir einmal an, Telefone hätten zwei Tasten

Nehmen wir einmal an, die Post würde beschließen, ein neues, verbessertes Telefon, "DigiPhon, das Telefon von Morgen", mit nur zwei Tasten, 1 und 0, einzuführen. Sie würde eine große Werbekampagne starten, um die Leute davon zu überzeugen, daß das DigiPhon schneller, moderner und überhaupt in jeder Beziehung besser sei als das alte, dezimale Telefon.

Jede Telefonnummer wird ins Binäre übersetzt: aus 844 71 71 wird 1000 0000 1110 0100 1100 0011. Vorwahlnummern bestehen nicht mehr aus maximal 5 Ziffern sondern aus maximal 14 Bits, genug, um alle möglichen Gebiete zu erreichen. Die Telefonbücher werden doppelt so dick, aber das ist kein Problem - man macht einfach die Schrift halb so groß.

Die Postkunden jedoch nehmen das neue System nicht an. Sie sagen es sei nahezu unmöglich eine Telefonnummer wie:

(10 1100 1001 1101) 0101 0000 1000 1001 0011 1000

korrekt zu wählen, geschweige denn im Kopf zu behalten. Ein Fehler, und man habe ein McDonald's Restaurant in München angerufen, anstatt seine Großmutter in Hamburg-Altona. Die Post jedoch

hat bereits 86 Millionen DigiPhons gebaut, die sie nicht verschrotten will. Sie bietet daher in Beiblättern, die allen Zeitungen im ganzen Land beigelegt werden, folgenden Kompromiß an:

"Verehrte Postkunden, wir werden folgendes tun. Wir kehren zurück zum gewohnten Telefonbuch und veröffentlichen jedermanns Telefonnummer in der gewohnten 10-Tastenform. Die Nummern werden sich leicht merken lassen, genauso wie früher. Wenn Sie jemanden anrufen wollen, wandeln Sie die Nummer in die 2-Tastenform um und wählen dann."

"Die Umwandlung der bekannten, dezimalen Telefonnummern in die moderne, digitale Form ist ganz einfach." Versuchen Sie zunächst, die Nummer durch 8388608 zu teilen. Wenn das geht, ist die erste Ziffer eine Eins, ansonsten eine Null. Anschließend teilen Sie den Divisionsrest durch 4194304. Wenn das geht, ist die zweite Ziffer eine Eins, ansonsten eine Null. Anschließend..."

Die Leute lassen die Post wissen, daß die Notwendigkeit, jedesmal 10 Minuten lang einen Taschenrechner bemühen zu müssen, wenn man telefonieren will, eine nicht gerade perfekte Lösung sei. Man steckt daraufhin eine Woche lang die Köpfe zusammen und gibt einen zweiten Kompromiß bekannt.

Die Lösung? Ein neues Telefonbuch mit Nummern in einem neuen, wirklich leicht zu behaltenden Format, das sich einfach, nahezu automatisch, ins Binäre übertragen läßt. Der große Durchbruch? Etwas, das sich hexadezimal nennt. Leichter zu behalten als binäre Zahlen; zwar nicht ganz so einfach wie die Dezimalzahlen, die wir schon in der ersten Klasse erlernt haben, aber viel leichter als binäre Zahlen. Vor allen Dingen aber beim Wählen einfach umzuwandeln.

Während im Dezimalen zehn Symbole das Zahlenalphabet bilden und im Binären zwei, gibt es im Hexadezimalen 16 Symbole. Dies scheint ein Problem zu sein, denn es liegen Symbole nicht einfach so herum, die die Werte von 10 bis 15 darstellen können. Obwohl es möglich gewesen wäre, völlig neue Symbole einzuführen, war es zweckmäßiger, etwas zu benutzen, von dem die Menschen oder die Schreibmaschinen bereits wissen, wie es geschrieben wird. Es wurde beschlossen, daß die ersten 6 Buchstaben des Alphabets die fehlenden Symbole sein sollten: A = 10, B = 11, C = 12, D = 13, E = 14 und F = 15. Hierbei war die Musik ein Vorbild, in der ja auch Buchstaben die Töne Do-Re-Mi-Fa-So-La-Ti darstellen.

Bewaffnet mit dem Wissen, daß Buchstaben manchmal auch Zahlen sein können, sehen Sie sich bitte nun das folgende Diagramm der Hexzahlen an.

Positionsgebundene Hexadezimalzahlen

vierte Ziffer	dritte Ziffer	zweite Ziffer	erste Ziffer
16^3	16^2	16^1	16^0
4096	256	16	1

A3F zerfällt in:

	10 * 256 +	3 * 16 +	15 * 1 = 2623

D006 ergibt:

13 * 4096 +	0 * 256 +	0 * 16 +	6 * 1 = 53254

Die Nützlichkeit von Hexadezimalzahlen liegt nicht darin, wie einfach sie sich in Dezimalzahlen umrechnen lassen, sondern darin, wie gut sie mit Binärzahlen zusammengehen. Hexadezimalzahlen vermitteln dem Menschen ein gutes, rundes Gefühl für Zahlen (zugegebenermaßen nicht so komfortabel wie die Dezimalzahlen), während sie gleichzeitig eine direkte Eins-zu-eins-Konvertierung ins Binärsystem ermöglichen. A03 mag Ihnen jetzt seltsam erscheinen, es läßt sich jedoch viel leichter damit umgehen als mit 1010 0000 0011.

Wie einfach ist es nun wirklich, Hex und Binär ineinander umzuwandeln? Studieren Sie die folgende Tabelle, in der in den drei Zahlensystemen gezählt wird:

Dezimal	Binär	Hex	Dezimal	Binär	Hex
0	0000 0000	00	16	0001 0000	10
1	0000 0001	01	17	0001 0001	11
2	0000 0010	02	18	0001 0010	12
3	0000 0011	03	19	0001 0011	13
4	0000 0100	04	20	0001 0100	14
5	0000 0101	05	21	0001 0101	15
6	0000 0110	06	22	0001 0110	16
7	0000 0111	07	23	0001 0111	17
8	0000 1000	08	24	0001 1000	18
9	0000 1001	09	25	0001 1001	19
10	0000 1010	0A	26	0001 1010	1A
11	0000 1011	0B	27	0001 1011	1B
12	0000 1100	0C	28	0001 1100	1C
13	0000 1101	0D	29	0001 1101	1D
14	0000 1110	0E	30	0001 1110	1E
15	0000 1111	0F	31	0001 1111	1F
			32	0010 0000	20

Bemerken Sie den Zusammenhang zwischen Hex und Binär? Jede Hex-Ziffer steht in direktem Zusammenhang mit einem binären Nibble.

Wenn Sie sich einmal die Bitmuster der 16 verschiedenen Nibbles und die zugehörigen Hex-Ziffern gemerkt haben, so ist die Umwandlung zwischen Hex und Binär ein Kinderspiel.

Eine hexadezimale Telefonnummer wie 45 6C A0 z.B. ergibt sofort:

4 = 0100
5 = 0101
6 = 0110
C = 1100
A = 1010
0 = 0000

Zusammen also die binäre Nummer:

0100 0101 0110 1100 1010 0000

Die Umwandlung vom Binären ins Hexadezimale ist genauso einfach. Ersetzen Sie nur jedes Nibble durch die äquivalente Hexziffer:

0011 0111 0001 1000 1100 0010

3 7 1 8 C 2 = 3718C2

Zwei Probleme bleiben noch übrig, bevor wir uns an dieses neue Zahlensystem gewöhnt haben: Erstens, woran sieht man einer Zahl wie 345 an, ob sie hexadezimal oder dezimal ist, und zweitens, wie um alles in der Welt spricht man eine Zahl wie F3C0 aus?

Zur Klärung des ersten Problems haben sich die 6502-Programmierer darauf geeinigt, hexadezimale Zahlen immer mit einem Dollar-Zeichen ("$") einzuleiten. Auch in diesem Buch werden wir dieser Konvention folgen. Bitte verwechseln Sie die Bedeutung des Dollar-Zeichens hier nicht mit derjenigen in BASIC, wo es zur Kennzeichnung eines Strings üblich ist. 345 ist also eine dezimale Zahl und $345 eine Hexadezimalzahl, die 837 dezimal entspricht.

Für die meisten von Ihnen wird es noch lange ein Problem bleiben, Hexzahlen auszusprechen, die Buchstaben enthalten. Wohl niemand hat diese Schwierigkeit ganz überwunden. Am besten ist es, bei Zahlen, die eine der neuen Ziffern enthalten, jede Ziffer einzeln auszusprechen. $C13 ergibt dann "Ce-eins-drei". Versuchen Sie es auch einmal mit "Ef-tausend" für $F000 und "Ce-hundert" für $C00.

Logische Operatoren

Binäre Zahlen haben einige Eigenschaften, die darüber hinausgehen, einfach nur dezimale Zahlen darzustellen und Papier zu füllen. Die Einfachheit der Zahlen 0 und 1 verleiht ihnen beson-

dere Möglichkeiten, unter anderem ermöglichen sie auch das, was man logische oder boole'sche Operatoren nennt (nach George Boole, einem englischen Mathematiker des 19. Jahrhunderts). Diese Operatoren sind UND ODER sowie EXKLUSIV ODER.

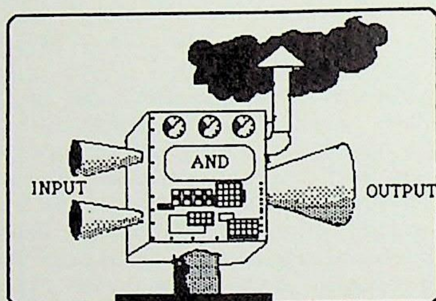
Die logischen Operatoren sind den 4 bekannten arithmetischen Operatoren plus, minus, mal und geteilt durch nicht unähnlich. Der größte Unterschied ist der, daß sie auf binäre Zahlen wirken, die nur ein Bit lang sind. Manchmal benutzt man auch die Ausdrücke "wahr" und "falsch" anstelle von 1 und 0. Beispiele logischer Operatoren sind:

1 UND 1

0 ODER 1

Wahr UND Wahr

Oft werden logische Operationen schematisch als "black box", mit 2 Eingängen, einem mysteriösen internen Prozeß und einem Ausgang, dargestellt.



Eine UND-Operation ergibt genau dann eins, wenn die beiden Operanden 1 sind.

Eine ODER-Operation ergibt eins, wenn einer oder beide Operanden eins sind.

Eine EXKLUSIV-ODER-Operation (EOR) ergibt eins, wenn die beiden Operanden verschieden sind.

Soweit zu den Regeln. Viele sind es ja nicht. In BASIC-Programmen haben sie vermutlich schon logische Operatoren verwendet, vielleicht ohne es zu wissen. Die IF-Anweisung in BASIC basiert auf logischen Operatoren.

IF (Ausdruck wahr) THEN mache dies

IF A > B THEN GOTO 1000

Bei der Bearbeitung dieser Anweisung löst der BASIC-Interpreter zunächst die Bedingung (A ³ B) und erhält entweder Wahr oder Falsch. Wenn A kleiner oder gleich B ist, so ist die Bedingung falsch. Wenn A gößer als B ist, ist sie wahr. Definitionsgemäß bewirkt ein "Falsch", daß die auf THEN folgenden Anweisungen übersprungen werden, dagegen werden sie bei "Wahr" ausgeführt.

IF A OR B THEN GOTO 1000

verzweigt zu Zeile 1000, wenn eine der beiden Variablen, A oder B, von Null verschieden ist (BASIC behandelt alles, was nicht Null ist, als 1). Zur Darstellung komplexer Beziehungen können logische Operatoren beliebig gruppiert und kombiniert werden.

IF A > B OR (FLAG AND G < 14) THEN GOTO 1000

Dies erscheint jedermann ganz natürlich zu sein, da wir solche Ausdrücke täglich benutzen:

"Wenn ich es finden kann und du mir das Geld gibst, dann kaufe ich es"

"Wenn es morgen nicht regnet oder du mir den Wagen läßt, dann werde ich in die Stadt fahren."

"Wenn die Kopiermaschine läuft oder Hans die Flugblätter fertig gedruckt hat und ich sie rechtzeitig bekomme, dann erhalten sie ihre Broschüre."

Abschliessende Übung

Füllen Sie die Leerstellen aus.

BINÄR	HEX	BINÄR	HEX
1001 1010			$F0
1111 1011			$02
0000 0001			$CA
1111 0000			$0C
1100 1101			$DD
0101 1010			$11
1011 1011			$E6

Wandeln Sie 10111 in eine Dezimalzahl um.

Konvertieren Sie 69 ins Binäre.

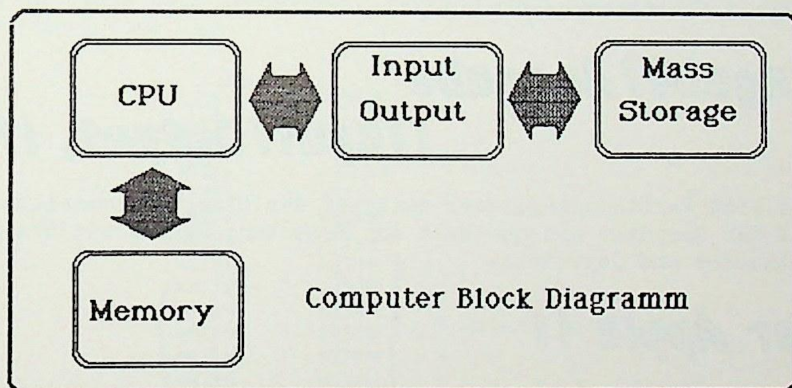
Berechnen Sie folgende logischen Ausdrücke:

0 UND 1 =	1 EOR 1 =
1 UND 1 =	0 EOR 0 =
0 ODER 1 =	1 UND 1 =
1 ODER 1 =	0 ODER 0 =
1 EOR 1 =	1 EOR 0 =
0 EOR 1 =	0 UND 0 =

Kapitel 3

Hardware

Ein 'Control Data Corporation Cyber 6600'-Computer ist so groß, daß er ein mittelgroßes Haus füllt. Ein APPLE II wiegt kaum 9 kg, wenn er völlig durchnäßt ist (daran darf man gar nicht denken!). Beide Maschinen haben jedoch vieles gemeinsam, und auf Blockdiagrammebene sind sie sogar identisch:



Zentrales Rechenwerk (Central Processing Unit, CPU)

Das zentrale Rechenwerk, die CPU, ist der absolute Herrscher jedes Computers. Die CPU trifft alle Entscheidungen und gibt allen anderen Komponenten das Tempo vor. Obwohl es annähernd genauso viele verschiedene CPUs wie Computer gibt, erfüllen alle CPUs die gleichen Aufgaben: Steuern, Entscheiden und Rechnen.

Hauptspeicher (Memory)

Neben der CPU spielt der Hauptspeicher die zweite Geige, ist jedoch ein unverzichtbares Mitglied im Team. Die CPU holt sich aus dem Hauptspeicher den Zahlenstrom, der ihre Arbeitsweise bestimmt, ein Maschinenprogramm. Die Grundoperation eines Computers besteht darin, daß die CPU Zahlen aus dem Hauptspeicher liest oder Zahlen in den Hauptspeicher schreibt. Gäbe es nicht die Notwendigkeit, mit dem Menschen zu kommunizieren, so könnten CPU und Hauptspeicher gut ohne die beiden anderen Komponenten auskommen.

Massenspeicher (Mass Storage)

Massenspeicher nehmen die Daten auf, die momentan nicht benötigt werden und für die im Moment kein Platz im Hauptspeicher vorhanden ist. Massenspeicher haben gegenüber dem Hauptspeicher üblicherweise einen finanziellen Vorteil: Sie kosten weniger pro Byte. Beispiele für Massenspeicher sind Diskettenlaufwerke und (Daten-) Cassettenrecorder.

Eingabe/Ausgabe (Input/Output, I/O)

Dies sind Verbindungsglieder zwischen der binären, numerischen Welt der Computer und der Welt der Menschen; Geräte wie Drucker, Tastaturen und Joysticks.

Der Apple II

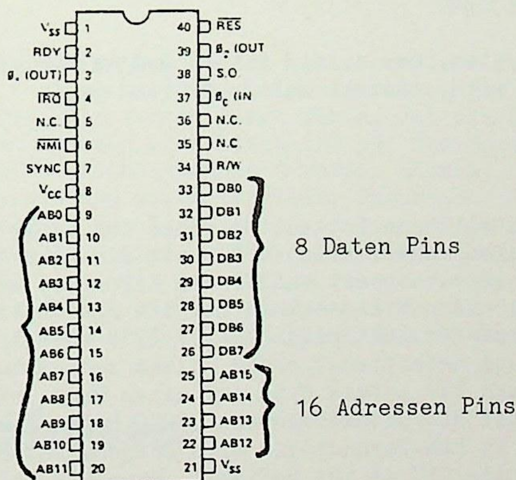
Alle Computer der APPLE II und III Serie benutzen einen 6502-Mikroprozessor als CPU. Ein Mikroprozessor ist ein integrierter Baustein (Integrated Circuit, IC oder Chip), der die gesamte CPU enthält. Die CPU eines Großrechners dagegen kann aus tausend oder mehr ICs bestehen. Der 6502 wurde 1975 von einer kleinen kalifornischen Firma, von MOS-Technology, eingeführt. Er ist eine Weiterentwicklung eines früheren Mikroprozessors, des Motorola 6800. Ein besonderes Qualitätsmerkmal dieser Entwicklungsarbeit ist es, daß der 6502 auch heute noch als leistungsstarker Prozessor für Minicomputer angesehen und verbreitet ist. Inzwischen wurde MOS-Technology von Commodore Business Machines aufgekauft, so daß der 6502 heute von Commodore und (unter Linzenz) von Rockwell und Synertek hergestellt wird.

Wenn Sie den Deckel eines II Plus lösen, so ist der große 40-beinige IC, der quer vor den Peripherieslots eingebaut ist, der 6502. Irgendwo in dem Irrgarten der aufgedruckten Buchstaben sollten Sie "6502" lesen können.

In einem //e ist der 6502 einer der drei 40-beinigen Chips. Die anderen beiden sind spezielle Hybrid-Bausteine, die die Funktionen vieler einzelner ICs des II Plus ersetzen. Diese Hybrid-Chips sind zwar genauso groß wie der 6502, jedoch längst nicht so intelligent. Der 6502 ist derjenige, der den Slots am nächsten liegt.

Der größte Herausforderer des 6502 um die Vorherrschaft in der Welt der 8-Bit-Mikroprozessor ist der Z-80 von Zilog. Obwohl beide mehr Gemeinsamkeiten als Unterschiede aufweisen, wird der Z-80 als register-orientierter und der 6502 als speicher-orientierter Prozessor geschätzt. Der Z-80 hat mehr eingebaute Speichermöglichkeit und der 6502 mehr Möglichkeiten im Umgang mit dem (externen) Speicher.

Der 6502 wird als 8-Bit-Mikroprozessor bezeichnet, weil er die gespeicherten Daten in 8 Bit großen Einheiten verarbeitet. Diese Anzahl schlägt sich auch in der Tatsache nieder, daß 8 der 40 Pins eines 6502 dem Datentransfer binärer Zahlen von und zum Chip dienen. Jedes Bein sendet oder empfängt das elektronische Äquivalent von 0 oder 1.



Mit 8 Bit kann man die Zahlen von 0 bis 255 darstellen. Obwohl sich dies wie eine ernste Einschränkung anhört, kann man bei der riesigen Geschwindigkeit des 6502 mit ein wenig Programmierung beliebig große Zahlen verarbeiten. Riesige Geschwindigkeit? In der Zeit, die man braucht, die RETURN-Taste zu drücken, kann der 6502 fünfzigtausend 8-Bit-Additionen ausführen.

16 der 40 Pins des 6502 werden dazu benutzt, die Speicheradresse zu spezifizieren. Dies läuft auf eine 16-ziffrige Binärzahl hinaus und bedeutet, daß der 6502 in der Lage ist, 65536 (216) verschiedene Speicherzellen zu adressieren. Den Speicher kann man sich als eine nummerierte Reihe von 65536 Fächern, etwa als einen gigantischen Karteischrank, vorstellen. Jedes Fach enthält eine Karteikarte mit einer Zahl von 0 bis 255 (eigentlich 8 kleine Karten mit einer Eins oder einer Null). Um den Speicher zu lesen, muß man zunächst einmal die Karteikarte heraussuchen und dann die Zahl lesen, die darauf geschrieben steht. Um in den Speicher zu schreiben, muß man zunächst die Karte heraussuchen und sie mit einer neuen Zahl beschreiben, wobei man die Zahl löscht, die ursprünglich dort geschrieben stand.

In einem eingeschalteten APPLE II steht der 6502 ständig in einem schnellen Dialog mit seinem Speicher. Wenn wir unser Ohr ganz dicht an das Tastenfeld halten und ganz still sind, so können wir vielleicht ungefähr folgendes hören:

6502: Speicher - Nenne mir den Inhalt von Zelle 45601.
Speicher: Okay. Diese Zahl ist.. äh, 134.
6502: Jetzt brauche ich den Inhalt von 45602.
Speicher: Das ist... 101.
6502: Hmmm... Sehr interessant. (Er überlegt für ein oder zwei Mikrosekunden.) OK, übernimm bitte 59 in Zelle 101.
Speicher: Hab' ich.

Das Speichersystem, das an die Daten- und Adresspins eines 6502 angeschlossen wird, besteht aus drei Grundtypen:

RAM

RAM ist die nützlichste Art von Speicher und daher nicht zufällig die am häufigsten anzutreffende. RAM ist die Abkürzung für Random Access Memory (Speicher mit wahlfreiem Zugriff), was bedeutet, daß man in der einen Mikrosekunde auf die Speicherzelle 3 und in der nächsten bereits auf Speicherzelle 6319 zugreifen kann. Ganz im Gegensatz zum Cassetten-Tonband, einem sequentiellen Speichermedium. Wenn man das letzte Byte auf einem Band lesen will, so muß man zunächst die davorliegenden 22000 Bytes überlesen. Eine Speicherzelle in RAM-Technologie wird folgsam Daten schreiben und lesen, so wie die CPU es ihr befiehlt. Wenn Sie die Versorgungsspannung des RAM Ihres Rechners ausschalten, so verlieren sie innerhalb weniger Mikrosekunden all die gespeicherten Informationen .(Wer von uns hat deswegen nicht schon mindestens einmal mit den Zähnen geknirscht?)

Sie sollten noch einmal den Deckel ihres Rechners abnehmen und sich die RAM-Bausteine ansehen. In einem APPLE II Plus gibt es drei Reihen zu je acht 16-beinigen Speicher-Chips, die insgesamt

ein RAM von 48K Bytes bilden. Diese Speicherbaugruppe befindet sich in der Mitte der Hauptplatine, eingerahmt von einer weißen Linie. Jeder einzelne Chip ist vom Typ 4116, der 16384 Ein-Bit-Zahlen speichern kann. Schaltet man 8 Chips parallel (d.h. steuert man sie mit derselben Adresse an), so erhält man 16384 Acht-Bit-Zahlen. Drei 'Bänke' zu je 8 Chips ergeben daher 48K Bytes.

Bitte tun Sie es nicht wirklich, aber lassen Sie uns einmal annehmen, wir würden einen Chip der mittleren Reihe herausziehen. Welche Speicherzellen wären betroffen? Antwort: Es wären die Zellen mit den Adressen 16384 bis 32767 ($4000 - $7FFF). Jedes Byte in diesem Bereich hätte dann eine Bit-Position, die immer 1 wäre, ganz egal, welche Daten Sie ursprünglich dorthin geschrieben haben. Welche Position dies ist, hinge davon ab, welchen Chip der mittleren Reihe Sie herausgezogen hätten.

Bei einem //e ist weniger mehr. 64K Speicher sind dort in nur acht 16-beinigen ICs enthalten, die sich in der Süd-Ost-Ecke der Hauptplatine befinden. Diese neueren Chips vom Typ 4164 können jeder 64K Bits speichern, so daß alle acht zusammen 64K Bytes ergeben. Würden wir einen dieser Chips herausnehmen, so fehlte in jedem einzelnen Byte des RAM ein Bit, und der Computer wäre mausetot.

ROM

Genauso wie im RAM enthalten die Speicherzellen im ROM Zahlen, die der 6502 (ebenso wahlfrei, also in beliebiger Reihenfolge) lesen kann. Der Unterschied liegt darin, daß die Zahlen in den Speicherzellen des ROM unauslöschlich bei der Herstellung eingebrannt wurden und nicht verändert werden können, egal wie oft die CPU versucht, sie neu zu beschreiben. Daher auch die Abkürzung: Read Only Memory = Nur Lese-Speicher. Diese Konzeption bringt Vor- und Nachteile zugleich. ROM ist nicht besonders flexibel (z.B. wenn Sie etwas mit dem Rechner machen wollen, wozu diese Zahlen nicht benötigt werden), dafür hat er die sympathische Eigenschaft, seine Informationen beim Ausschalten nicht zu verlieren.

E/A - Adressen

E/A(Eingabe/Ausgabe)-Adressen sind der dritte Typ der Speicherzellen. Es handelt sich um Adressen, die mit Teilen des Rechners außerhalb der CPU-ROM/RAM-Clique verbunden sind. Über die E/A-Adressen kommuniziert der 6502 mit dem Rest der Maschine. Einige E/A-Adressen erlauben es externen Geräten, Nachrichten für den 6502 zu hinterlassen. Um bei unserer Karteikartenanalogie von eben zu bleiben: es gibt einige Karteikästen mit einem Loch in der Rückwand, durch das sich neue beschriebene Karteikarten einlegen lassen.

Will der 6502 im APPLE z.B. wissen, welche Taste gedrückt ist, so schaut er auf die Karteikarte im Kasten mit der Aufschrift "Tastatur". Wenn er die Karte beschreiben will, so funktioniert dies nicht; nur die Tastatur kann die Zahl auf der Karte verändern.

Im APPLE gibt es noch andere E/A-Adressen, die wie adressenabhängige Schalter arbeiten. Diese Karteikästen enthalten gewissermaßen einen Stolperdraht, der immer dann einen verborgenen Mechanismus auslöst, wenn wir versuchen, diese Speicherzelle zu lesen oder sie zu beschreiben. Jedes Lesen oder Beschreiben der Zelle 49200 (eine E/A-Adresse mit der Aufschrift "Lautsprecher") bewirkt z.B., daß ein Lautsprecher irgendwo (in einer Schublade vermutlich) einen Ton von sich gibt. Allein die Tatsache der Adressierung dieser Zelle, das Herausnehmen der Karteikarte, egal ob sie gelesen oder beschrieben werden soll, löst den Stolperdraht und damit den gewünschten Effekt aus.

Teile und herrsche

Es ist nützlich, die 65536 (64K, wobei 1K = 2^{10} = 1024) Speicherzellen in 256 "Seiten" zu je 256 Zellen anzuordnen. Denken Sie sich den Speicher als ein Buch mit 256 Seiten und 256 Worten (Bytes) auf jeder Seite. Seite 3 besteht aus den Speicherzellen 768 bis 1023, oder $300 bis $3FF. In der hexadezimalen Zahlendarstellung wirkt das Seitenkonzept ganz natürlich, da jede Adresse sofort in eine Seite und einen Ort auf dieser Seite zerfällt: Die Speicherzelle $3411 ist das $11. Byte auf Seite $34.

Die Tatsache, daß jeder 6502 auf 65536 verschiedene Speicherzellen zugreifen kann, bedeutet noch lange nicht, daß jeder 6502 auf der Welt darauf zählen kann, diese Menge an Speicher zur Verfügung zu haben. Ein Ingenieur, der einen 6502 als das Gehirn eines Mikrowellenherdes benutzen will, mag z.B. entscheiden, daß 1000 Bytes ROM und 100 Bytes RAM ausreichen, um den intelligentesten Mikrowellenherd der Welt zu bauen.

Der 6502, der in solch einem Mikrowellengerät installiert ist, kann zwar immer noch 65536 verschiedene Speicherzellen adressieren, aber nur etwa 1000 sind tatsächlich vorhanden. Wenn er versucht, auf eine der nicht implementierten Adressen zuzugreifen, handelt er wie ein Roboter in einer Toyota-Fabrik, der blind versucht, einen Corolla zu schweißen, der 3 Meter neben ihm auf dem Fließband steht. Er meint, er lese eine Anweisung an der Adresse $C000, was er jedoch sieht, ist irgendwelcher zufälliger Müll.

Welche Art von Speicher liegt nun an welcher Adresse in einem APPLE? Der 6502-Programmierer muß das wissen, damit er nicht versucht, seine Daten im ROM zu speichern. Die Speicherliste ist ein nützliches Hilfsmittel, um den Grundaufbau des 64K-Adressbereiches Ihres Rechners auf einen Blick zu überschauen.

Die Speicherliste des Apple II

```
Seitennummer

Dez     Hex     Funktion
------------------------------------------
255     FF
.
.               ROM (12k; $D000 - $FFFF)
.
208     D0
------------------------------------------
207     CF      E/A ( 4k; $C000 - $CFFF)
192     C0
------------------------------------------
191     BF
.
.
.
.
.
.
.
.
.
.               RAM (48k; $0000 - $BFFF)
.
.
.
.
.
.
.
.
.
0       00
------------------------------------------
```

Drei Viertel des Adressbereiches sind RAM. In früheren Tagen (1977-79) haben die meisten APPLE-Computer das Fließband mit nur 16K Speicher im unteren Adressbereich verlassen (einige hatten sogar nur 4K - man stelle sich vor!). Die Rechner waren nicht anders; nur kostete damals jeder 16K-RAM-Chip etwa 160 DM. Versuchte man, eine Zahl in Speicherzelle $4010 zu schreiben, so hielt einen nichts davon ab, nur wurde die Zahl nicht gespeichert. Heutzutage sind 48K oder mehr üblich. Diese relativ große Menge RAM-Speicher verleiht dem APPLE eine große Flexibilität, da man nun im RAM alles machen kann, von BASIC angefangen über PASCAL bis hin zu Programmen, die Mikroprozessoren simulieren.

$C000 bis $CFFF sind 4096 Speicherzellen für Ein-/Ausgabe. Ohne diese Speicherzellen könnte der APPLE der Menschheit gar nicht mitteilen, was für erstaunliche Dinge er zustande bringt. Die Adressen in diesem Bereich sind mit APPLE-Hardware wie Lautsprecher, Tastatur, Spiele-Regler und Diskettenlaufwerk verbunden.

Im Bereich von $D000 bis $FFFF liegt ROM. In diesem ROM ist ein Maschinenprogramm gespeichert, das die sogenannten APPLESOFT-Programme laufen läßt. Ferner enthält es eine Reihe von Hilfsprogrammen, die die Tastatur lesen, Text darstellen, die Spiele-Regler abfragen können usw.; diese Hilfsprogramme werden unter dem Begriff APPLE-Monitor zusammengefaßt.

Ich dachte, mein IIe hätte 64K

Obwohl in den APPLE //e 64K RAM eingebaut sind, sind die oberen 16K so ausgelegt, daß sie sich normalerweise verhalten, als seien sie nicht vorhanden. Wäre dies nicht der Fall, so ergäben sich laufend Kämpfe auf dem Adress- und Datenbus, sobald man versuchte, im Bereich zwischen $C000 und $FFFF zu lesen. Versuchte die CPU z.B. die Speicherzelle $D341 zu lesen, so ergäbe sich ein Datenwirrwarr auf dem Bus, da ROM und RAM jeweils etwas anderes sagen würden. Um diese Situation zu entwirren, wird der RAM abgeschaltet (deselektiert) und der ROM eingeschaltet. Es besteht die Möglichkeit, diese Anordnung umzukehren. In einem II Plus ergibt sich eine ähnliche Situation bei Verwendung einer 16K-RAM-Karte.

Massenspeicher des APPLE

$C000 Bytes sind eine Menge RAM - manchmal jedoch nicht genug. Ein Disk II-Laufwerk kann 140000 Acht-Bit-Zahlen auf einer Diskette speichern, und wir können so viele Disketten benutzen, wie wir uns leisten können. Gelegentlich veranlaßt der 6502 das Diskettenlaufwerk, etwas von der Diskette ins RAM zu laden. Einmal im RAM, kann der 6502 mit den Bytes in der normalen, vertrauten, schnellen Weise arbeiten. Disketten haben darüber hinaus die nützliche Eigenschaft, ihre Daten nicht zu verlieren, wenn die Spannung abgeschaltet wird.

Aber wie funktioniert es?

Anders als in dem Film TRON dargestellt, ist die Welt des 6502 so weit von menschlicher Erfahrung entfernt wie nur irgend etwas. Sie gleicht eher den wirbelnden Nocken und Hebeln einer Flaschenabfüllanlage als Menschen, die in lustigen Hüten mit leuchtenden Frisbee-Scheiben Fangen spielen. Obwohl dies so ist, kann ein

Vergleich der Arbeitsweise eines 6502 mit der eines Menschen sehr nützlich sein bei der Erklärung, wie Maschinensprache funktioniert.

Betrachten wir hierzu die Versandabteilung des GIANT METROPOLITAN Software Verlages GmbH. Zahllose Lieferwagen fahren schwerfällig an Laderampen heran und von ihnen weg. Arbeiter mit Transportwagen und Gabelstaplern laden kühlschrankgroße Kartons mit leeren Disketten aus und fertige Programme und Bedienungsanleitungen ein.

Der unangefochtene Chef des Versandes ist der Vorarbeiter, eine imposante Erscheinung im himmelblauen Overall und orangem Schutzhelm, der die Arbeiter hin und her dirigiert und Versandpapiere unterzeichnet, die man ab und zu auf einem Klemmbrett sieht, das er in seiner linken Hand hält.

Durch die Macht DES BUCHES hat er die Dinge fest in der Hand. DAS BUCH ist ein ziemlich abgenutztes Notizbuch von vielleicht 150 Seiten. Der Titel auf der Vorderseite, inzwischen unleserlich von der jahrelangen Benutzung, lautete einst: "Verfahrenshandbuch der Versandabteilung". Die Seiten sind nummeriert. Einige enthalten nur drei Zeilen, andere 10 oder 15. Auf Seite 12 z.B. steht geschrieben:

> Anweisung 12 UPB -- UPS blauer Versand
>
> Schritte:
>
> 1. Die Packliste liegt hinter der Arbeitsanweisung
> 2. Bringe einen blauen Aufkleber an
> 3. Im UPS-Bereich entlassen
> 4. Fertig

Jeden Morgen findet der Vorarbeiter einen dicken Stapel Arbeitsanweisungen in seinem Korb vor. Auf jeder steht bürointernes Kauderwelsch und, in der linken oberen Ecke, die alles entscheidende Versandanweisungsnummer. Da die meisten Arbeitsanweisungen ein oder zwei weitere Papiere benötigen, um komplett zu sein, sind nicht alle Blätter in dem Stapel Arbeitsanweisungen.

Das wichtigste Werkzeug des Vorarbeiters der Versandabteilung ist sein graues Klemmbrett für die Arbeitsanweisungen. Nach seinem Morgenkaffee nimmt er die erste Anweisung vom Stapel und befestigt sie auf seinem Klemmbrett. Solange sie an diesem Platz ist, widmet er all seine Energie darauf, die Arbeiten durchzuführen, mit denen diese Anweisung erledigt wird.

Es gibt eine ganze Reihe von Schreibkram auf diesen Arbeitsanweisungen, er ist jedoch nur an der Anweisungsnummer interessiert. Heute trägt die erste Anweisung die Nummer 22. "TCP" murmelt er vor sich hin, als er Seite 22 des Verfahrensbuches aufschägt. (Er kennt Anweisung 22 auswendig, trotzdem sieht er nach. Er gehört zu dieser Sorte Mensch.)

Anweisung 22 TCP -- TEXPAK C.O.D

Schritte:

1. Die Packliste liegt hinter der Arbeitsanweisung
2. TEXPAK wegen der Übernahme anrufen
3. C.O.D. Formular ausfüllen
4. Paket in den Ausgabebereich bringen
5. Fertig

Wenn er den letzten Schritt von TCP erledigt hat, nach 5 oder 15 Minuten, geht er zu seinem Schreibtisch zurück, entnimmt die alte Arbeitsanweisung und die ihr folgenden Packzettel seinem Klemmbrett und legt sie mit der Schrift nach unten in den Ausgangskorb. Ohne eine Pause entnimmt er dem Eingangskorb die nächste Anweisung, befestigt sie auf seinem Klemmbrett und geht los, um sie zu bearbeiten. So geht das tagein tagaus: Arbeitsanweisung entnehmen, Anweisungsnummer nachschlagen, Anweisung durchführen. Arbeitsanweisung entnehmen, Anweisungsnummer nachschlagen ....

Ein 6502 arbeitet genauso: Nimm eine Speicherzelle, entschlüssele den Inhalt dieser Zelle. Die Anweisung benötigt eventuell das nächste oder die nächsten beiden Bytes zur Durchführung. Bearbeite das Kommando und gehe dann über zur nächsten Speicherzelle für die nächste Anweisung.

Die phantastische Reise

Erinnern Sie sich noch an den Film Die phantastische Reise? In dem einige unerschrockene Wissenschaftler/Soldaten auf die Größe einer Mikrobe verkleinert wurden, um bei der Beseitigung eines Tumores im Hirn eines wichtigen (so glaube ich jedenfalls) Wissenschaftlers zu helfen?

Durch den Zauber des geschriebenen Wortes werden wir dasselbe machen, allerdings ohne Raquel Welch in der Crew. Oder, nein! - sie kann ruhig mitkommen. Wir klettern in unser Unterseeboot, das wie ein Rochen aussieht, und machen uns bereit. Halten Sie sich fest! Soldaten bestrahlen uns mit einem seltsamen violetten Licht. Wir schrumpfen. Wir werden kleiner...kleiner...kleiner...

Wir sind jetzt so winzig, daß der Punkt am Ende dieses Satzes wie die Navigationskuppel aussieht. Wir heben ab (dieses Unterseeboot kann auch fliegen) und steuern geradewegs auf einen in der Nähe befindlichen Rechner zu. Er sieht so groß aus wie die Zugspitze. Wir schlüpfen einfach durch einen Schlitz im Tastenfeld hinein in eine bizarre, fremde Landschaft aus Kabeln und Taktkristallen. Gespenstergleich gleiten wir weiter, es kommt uns wie Stunden vor. Plötzlich, direkt vor uns, ein riesiger schwarzer Monolith, das Ziel unserer Reise: der 6502-Mikroprozessor.

Kapitel 4
Starten des Programms

Es ist Zeit sich mit VISIBLE COMPUTER vertraut zu machen. Nehmen Sie die mitgelieferte Diskette aus dem Umschlag, schieben Sie sie in Drive 1 und schalten Sie den Rechner ein. Besitzer eines APPLE II-Standard-Rechners müssen (anders als beim APPLE II Plus oder //e) einen Zwischenschritt tun: Booten Sie mit der DOS 3.3 Master-Diskette, damit APPLESOFT in die RAM- oder Language-Karte geladen wird. Legen Sie dann die VC-Diskette ein und booten Sie mit PR#6.

Nach wenigen Sekunden erscheint eine Copyright-Meldung von Software Masters, die für einige Sekunden, oder bis eine Taste gedrückt wird (je nachdem, was früher eintritt), auf dem Bildschirm stehen bleibt. Wenn Sie diese Meldung nicht sehen, so gibt es Probleme.

Wenn die Diskette nicht bootet ...

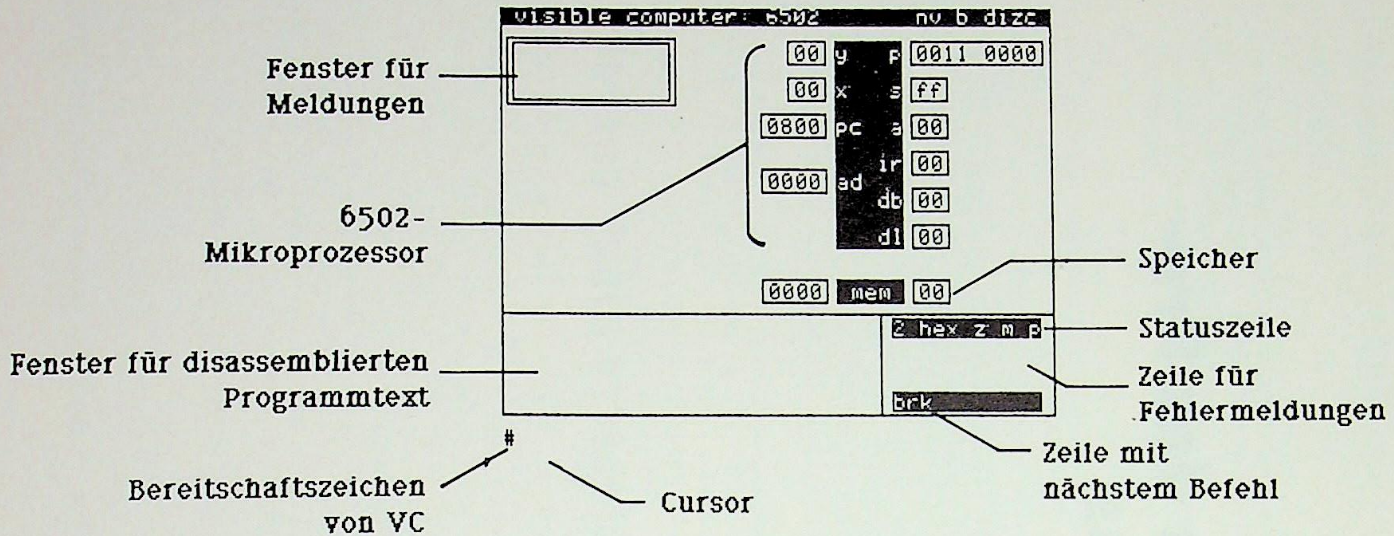
Hat Ihr APPLE wenigstens 48K? Hat er eine 16 Sektor (DOS 3.3) Disk-Controller-Karte? Ist es ein II Plus oder ein //e; oder, wenn es ein APPLE II Standard ist, hat er entweder eine APPLESOFT-ROM-Karte oder eine 16K-RAM-Karte (APPLE-Language-Karte, Microsoft-RAM-Karte)? Wenn Ihre Antwort auf eine dieser Fragen nein ist, so müssen Sie zunächst Abhilfe schaffen, ehe VC auf Ihrer Maschine laufen kann.

Wenn Ihr System den nötigen Anforderungen entspricht und trotzdem nicht bootet, so haben Sie vielleicht eine schadhafte Diskette. Wenden Sie sich dann an Ihren Händler oder schreiben Sie direkt an PANDABOOKS unter der auf der letzten Seite angegebenen Adresse.

Geht alles gut, so wird die Copyright-Meldung durch die Meldung

LOADING...

ersetzt, und nach etwa 15 Sekunden erscheint das Hauptdisplay von VISIBLE COMPUTER.



Die Bildschirmdarstellung von VISIBLE COMPUTER

Wir schauen jetzt auf etwas, was wenige je gesehen haben: die Innereien eines arbeitenden Mikroprozessors.

Jedes der Kästchen, die je eine Hexzahl enthalten, stellt ein Register dar. Ein Register ist eine Stelle innerhalb des Prozessors, an der wir binäre Zahlen speichern können, ähnlich wie in den Haupspeicherzellen. Der 6502 kann mit nur wenigen Operationen mit den Inhalten dieser 10 Register wahre Kunststücke anstellen. Wir beginnen im nächsten Kapitel damit, uns anzusehen, wie dies geschieht.

Da wir gerade vom Speicher sprachen, die Verbindung des 6502 mit seinen 65536 Speicherzellen geschieht über die beiden mit "mem" (für Memory = Speicher) bezeichneten Register. Das 16 Bit breite Register ist für die Adressen und das 8 Bit breite für die Daten, die an dieser Adresse gespeichert sind, zuständig. Während der Programmausführung erscheinen dort die Zahlen, die aus dem Speicher gelesen bzw. dorthin geschrieben werden.

In der oberen linken Ecke ist das Fenster für Meldungen, wo VISIBLE COMPUTER die einzelnen Schritte angibt, die er bei der Durchführung von 6502-Programmen macht. Wenn VC kein Programm ausführt (jetzt zum Beispiel), so ist dieses Fenster leer. Die Beschreibung des Fensters für disassemblierten Programmtext heben wir uns für später auf, wenn wir wissen, was "disassemblieren" ist. Im Statuskästchen in der rechten unteren Ecke des Bildschirmes werden verschiedene Besonderheiten dargestellt, die mit der Programmausführung von VC zu tun haben. Am unteren Bildrand befindet sich die Kommandozeile. Dort werden Sie Ihre Befehle eingeben, um VISIBLE COMPUTER zu steuern. Das '#'-Zeichen (Doppelkreuz oder Nummer-Zeichen) ist das Bereitschaftszeichen (der Prompt) des VC-Monitors. Es soll Sie daran erinnern, daß Ihre Eingaben Befehle sein sollten, die VC versteht.

Das rechts neben dem Prompt stehende Zeichen ist der Cursor, der, genauso wie das blinkende Rechteck in BASIC, anzeigt, wo auf dem Bildschirm das nächste Zeichen, das Sie eintippen, erscheint.

VISIBLE COMPUTER besteht aus zwei Hauptteilen: dem Monitor und dem 6502-Simulator. Der Monitor steuert (to monitor = überwachen, steuern) den Simulator. Der Simulator ist der Teil, der die 6502-Programme dann wirklich ausführt. In dieser Anleitung werden wir Phrasen wie "im Monitor" und "Rückkehr zum Monitor" gebrauchen. Sie sind "im Monitor", wenn Sie den Prompt (das #-Zeichen) in der Kommandozeile sehen. Sie sind "im Simulator", wenn Sie den Prompt nicht sehen.

Monitor-Kommandos

Das Vokabular von VISIBLE COMPUTER umfaßt mehr als 20 Kommandos. Sie sagen ihm etwas, und wenn das etwas ist, was er versteht, so wird er es tun.

Eingabe von Kommandos

Sie geben ein Kommando, indem Sie Ihre Anforderung eintippen und RETURN drücken. Besteht ein Kommando aus mehreren Teilen, so trennen Sie diese mit einem oder mehreren Leerzeichen. Wenn Sie einen APPLE //e benutzen, so achten Sie darauf, daß die SHIFT LOCK-Taste gedrückt ist (ironischerweise erzeugt die Eingabe dann kleine Buchstaben - kümmern Sie sich nicht darum).

Wenn Sie bei der Eingabe Fehler machen, so verwenden Sie zum Korrigieren die Linkspfeil-Taste. Alternativ dazu können Sie auch CTRL-X drücken und neu beginnen.

Versteht VC Ihre Anweisungen nicht, so wird er Ihnen dies durch Fehlermeldungen mitteilen.

VC-Kommandos haben die folgende generelle Form:

Kommando <Argument1> <Argument2>

"Argument" ist ein viel benutztes Computer-Wort, das Modifikator oder Parameter bedeutet. Einige VC-Kommandos haben keine Argumente; andere brauchen weitere Informationen, um vollständig zu sein. Genauso stehen einige BASIC-Kommandos ("END", "NEW") alleine, während andere ("IF", "GOSUB") wenig Sinn ergeben, wenn Sie nicht weitere Informationen hinzufügen. Beispiele für Monitor-Kommandos, die nur aus einem Wort bestehen, sind ERASE und RESTORE.

Das Monitor-Kommando BASE (d.h. ändere die Zahlenbasis eines Registers um in Hex, Binär oder Dezimal) benötigt zwei Argumente: das erste spezifiziert das Register, dessen Zahlenbasis wir ändern wollen, das zweite gibt die neue Basis an. BASE PC BIN stellt den Darstellungsmodus des PROGRAM COUNTER (Programmzählers) auf binär ein.

Ein Kommando und seine Argumente müssen durch ein oder mehrere Leerzeichen voneinander getrennt werden. Dagegen dürfen innerhalb eines Kommandos keine Leerzeichen vorkommen. Wenn Sie z.B. das Kommando ERASE als ER ASE eingeben, so würde der Kommandointerpreter von VISIBLE COMPUTER annehmen, Sie meinten: "Führe das Kommande ER aus und benutze das Argument ASE", was zu einem Fehler führen würde, da es das Kommando ER nicht gibt. Sollten Sie also einmal feststellen, daß ein Kommando nicht die gewünschte Funktion ausführt, so überprüfen Sie die Syntax Ihrer Eingabe und achten dabei auf die Leerzeichen in Kommandos, die Argumente haben.

Wir wollen uns nun mit einigen Kommandos näher befassen. Zuerst werden wir den Kalkulator aufrufen und sehen, ob zwei plus zwei vier ergibt. Ich bin sicher, daß dies eine Frage ist, die viele von Ihnen sehr interessiert.

Tippen Sie CALC ein und drücken Sie RETURN (das Eintippen einer Anweisung und das abschließende Drücken der RETURN-Taste werden wir in Zukunft einfach Eingabe nennen). Wenn Sie einen

Fehler machen, so fahren Sie den Cursor einfach mit der Linkspfeil-Taste zurück und korrigieren ihn. Wenn im Statusfenster "Command" erscheint, so hat VC Ihre Eingabe nicht verstanden.
Vielleicht sehen Sie aber auch dies in der Kommandozeile:

<HEX><CALC>00

Das "HEX" bedeutet, daß die momentane Zahlenbasis des Kalkulators hexadezimal ist: Alle Zahlen, die er erzeugt, werden hexadezimal dargestellt (ohne Dollarzeichen), und alle Zahlen, die Sie eingeben, dürfen nur Zeichen enthalten, die in dieser Zahlenbasis gültig sind. Gleiches gilt analog für die anderen beiden Zahlenbasen. Mit anderen Worten, geben Sie keine Hex-Zahlen wie G3#B und keine Dezimalzahlen wie FC3 ein. Geben Sie bitte auch kein Dollarzeichen ein, wenn die Zahlenbasis hexadezimal ist; das Dollarzeichen ist sozusagen implizit enthalten. Sie können die Zahlenbasis des Kalkulators jederzeit durch CTRL-B in binär und durch CTRL-D in dezimal umändern (geben Sie CTRL-B bzw. CTRL-D ein, indem Sie die CTRL-Taste gedrückt halten, während Sie B bzw. D drücken). Im Moment belassen Sie jedoch die hexadezimale Basis (CTRL-H) und geben folgendes ein:

2 + 2

2 + 2 wird ersetzt durch 04 (haben Sie RETURN gedrückt?). Wenn Sie nicht vier als Antwort erhalten haben, so prüfen Sie, ob Sie die Leerzeichen vor und hinter dem Plus-Zeichen eingegeben haben.
Versuchen Sie sich nun an folgenden Problemen:

3 + 3
4 * 4
6 / 2
3EA * C (versuchen Sie das mal auf Ihrem Taschenrechner!)

Um zwischen den einzelnen Zahlenbasen umzurechnen, tun Sie folgendes.
Konvertierung von 65000 dezimal ins Hexadezimale:

CTRL-D
65000<RETURN>
CTRL-H

Konvertieren Sie mit CTRL-B ins Binäre und zurück ins Dezimale mit CTRL-D. Versuchen Sie in der Basis Dezimal FF und 3 zu addieren. Mit der BASE Fehlermeldung teilt Ihnen VC mit, daß Sie Zeichen eingegeben haben, die mit der momentanen Basis nicht verträglich sind. FF ist keine gültige dezimale Zahl.
Die Ergebnisse werden mit führenden Nullen angezeigt. Bei binären Zahlen werden zusätzlich die einzelnen Nibbles mit Leer-

zeichen getrennt. Der Kalkulator hat einige Eigenschaften, die ihn für die täglichen Berechnungen Ihres Kontostandes und der Verbrauchswerte Ihres Autos untauglich machen. Er ist ein Ganzzahlenrechner; Zahlen mit Dezimalpunkt sind als Eingaben nicht erlaubt. Divisionen ergeben ein abgehacktes (im Gegensatz zu einem gerundeten) Resultat. So ist z.B. 6 / 2 = 3; 5 / 2 = 2; 9 / 10 = 0.

Sie dürfen auch keine negativen Zahlen eingeben. Das Minuszeichen wird als unzulässiges Zeichen behandelt, es sei denn, Sie verwenden es als Operator. Wenn Sie eine Subtraktion ausführen, die ein negatives Resultat erzeugt, indem Sie eine große Zahl von einer kleineren abziehen, so wird das Resultat im Zweierkomplement dargestellt (wir werden später besprechen, was das ist). Schließlich dürfen Sie auch keine Zahlen eingeben oder durch Rechnungen erzeugen, die größer sind als 65535. Notwendig sind diese Einschränkungen aufgrung der eigentlichen Aufgabe des Kalkulators, nämlich Ihnen zu helfen, Maschinenprogramme zu schreiben.

Wenn Sie Ihren Spaß beim Umrechnen von Zahlen zwischen den verschiedenen Zahlensystemen hatten, so drücken Sie bitte ESC, oder falls dies nicht funktioniert, RETURN ESC hintereinander, so daß Sie zum Monitor zurückgelangen. Haben Sie den Monitor-Prompt auf Ihrem Bildschirm? Dann versuchen Sie jetzt folgendes nette, kurze Kommando:

ERASE

Wow, wie spektakulär! ERASE löscht den Bildschirm, so daß Sie mit dem Display experimentieren können. Da es jedoch traurig ist, den armen kleinen Monitor-Prompt ganz allein auf dem Bildschirm zu sehen, bringen Sie die alte Anzeige mit dem Kommando RESTORE zurück auf den Schirm. Wenn Sie wollen, können Sie diese beiden Kommandos immer und immer wieder geben. Für die mehr abenteuerlich Veranlagten wollen wir jedoch weitermachen.

Geben Sie ein: WINDOW OPEN (Fenster offen). Jetzt haben wir nur einen Teil der Anzeige gelöscht. Wenn Sie mehr über die Programmierung des 6502 wissen, werden Sie es zu würdigen wissen, daß Sie wählen können, welche Register auf dem Bildschirm verbleiben, während das Fenster geöffnet ist. Auch jetzt wollen wir unsere Anzeige nicht so leer stehen lassen. Stellen wir also den ursprünglichen Zustand mit WINDOW CLOSE wieder her.

Was ist so grossartig an Kleinbuchstaben?

Haben Sie eine Abneigung gegen Kleinbuchstaben? Wenn ja, so verbannen Sie diese Pest mit dem Kommando CASE UPPER vom Bildschirm. Obwohl man sich zunächst etwas daran gewöhnen muß, haben "Experten" bewiesen, daß der Mensch Kleinbuchstaben schneller erfassen kann als Großbuchstaben. Nun, da auch Sie von der Überlegenheit der Kleinbuchstaben überzeugt sind, schalten Sie mit dem Kommando CASE LOWER zurück.

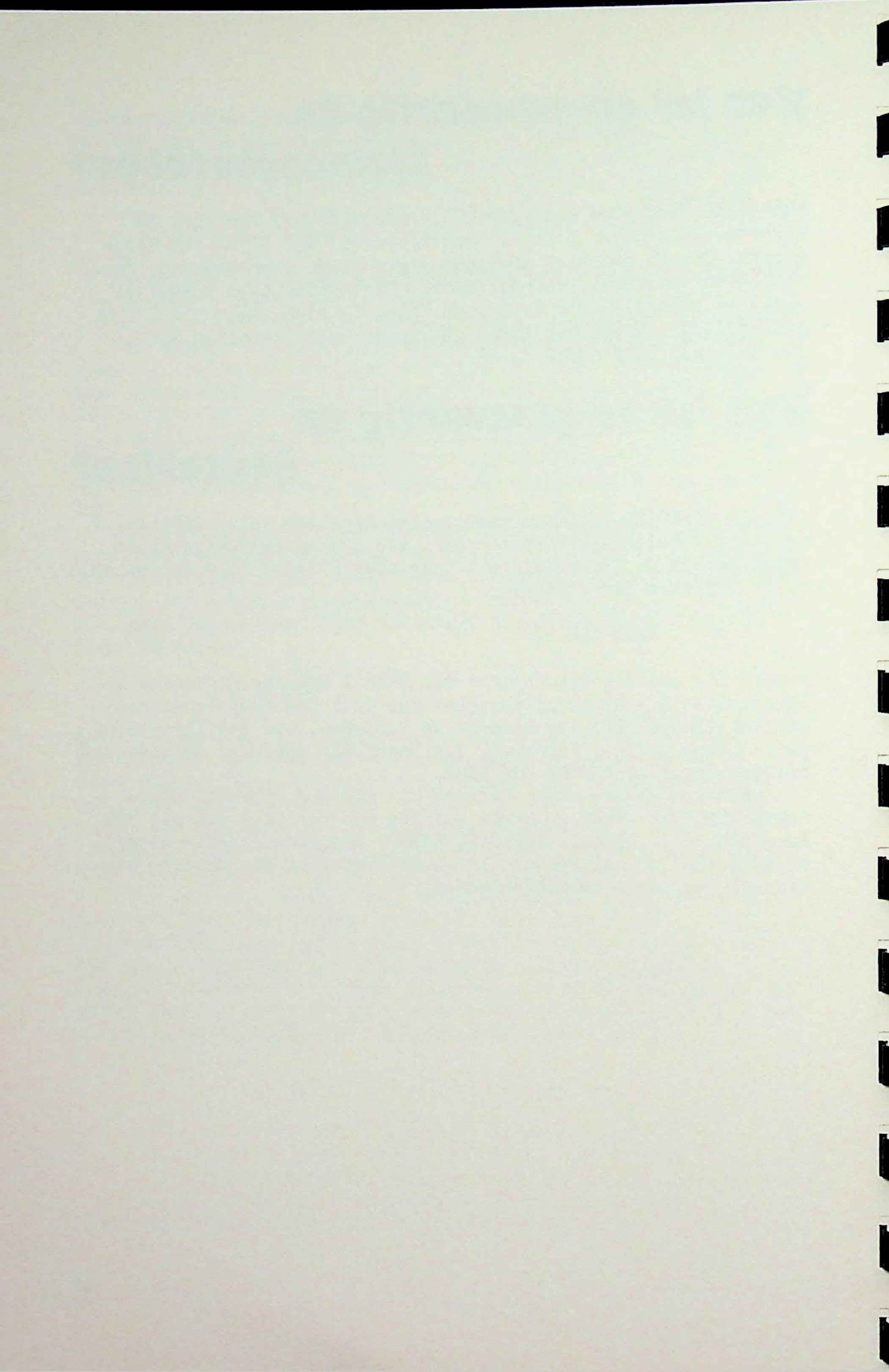
Was ist so grossartig an Hexzahlen?

VISIBLE COMPUTER stellt zunächst alle Register außer Register P im hexadezimalen Zahlensystem dar. Sie müssen es jedoch nicht dabei belassen. Ändern Sie mit folgendem Kommando die Zahlenbasis aller Register ins Binäre:

```
BASE ALL BIN
```

Ändern Sie nur den Akkumulator mit BASE A DEC ins Dezimale. So können Sie jede Mischung zwischen den drei Zahlensystemen einstellen. Lassen Sie die Anzeige so aussehen, wie Sie Ihren 6502 gerne haben wollen. Nebenbei, die Namen der Speicher Adress- und Datenregister sind MEMA und MEMD.

Damit ist unsere erste Sitzung mit VISIBLE COMPUTER abgeschlossen. Wir haben gelernt, was der Monitor ist, und mit den Kommandos CALC, ERASE, RESTORE, WINDOW und BASE experimentiert. Im nächsten Kapitel werden wir etwas tiefer in die Materie eintauchen und darin herumplantschen.



Kapitel 5

Das Arbeiten mit dem Speicher

In diesem Kapitel werden wir lernen, wie die Speicheraufteilung in Ihrem Rechner aussieht, wenn das VC-Programm läuft. Anschließend werden wir das Prüfen und Ändern von Speicherinhalten mit dem Monitor üben.

Seitennummer

```
BF---------------------------------------------
::
:: Reserviert für VC ($0C00 - $BFFF)
::
0C
::
:: Primärer Benutzerbereich ($0800 - $0BFF)
::
08
::
:: Textseite 1 des APPLE
::
04
:: Seite 3 Benutzerbereich ($0300 - $03CF)
03
:: Seite 2 (Eingabepuffer)
02
:: Seite 1 (Stack)
01
:: Seite 0 (Zero Page)
00---------------------------------------------
```

Obwohl Sie die Inhalte nahezu jeder Speicherstelle lesen können, dürfen Sie den Bereich von $0C00 - $BFFF (mehr als 90% des RAM-Bereiches) nicht beschreiben. Wenn es Ihnen erlaubt wäre, diese Region mit Zahlen Ihrer Wahl zu belegen, so könnten Sie VC verletzen; vielleicht würden Sie das Programm zum Absturz bringen,

oder Sie würden nur eine einzelne Funktion leicht verändern. VISIBLE COMPUTER scheint Ihren Auftrag, einen Wert an die Speicherstelle $4003 zu schreiben, zu akzeptieren (keine Fehlermeldung), wird ihn jedoch nicht ausführen. Er behandelt den ROM und die E/A-Adressen auf die gleiche Weise.

Der 2K große Bereich, der für Schreiboperationen zur Verfügung steht ($0000 - $03CF und $0800 - $0BFF), ist jedoch groß genug für die meisten Maschinenprogramme. Der als "Textseite 1 des APPLE" bezeichnete Bereich von $0400 - $07FF sollte nicht benutzt werden, obwohl Sie VC dabei nicht schaden können. Sie werden nämlich die Werte, die Sie dort gespeichert haben, nicht wieder vorfinden, wenn Sie später darauf zurückgreifen wollen.

Später, wenn Sie einige Meilensteine in Ihrem Studium passiert haben werden, werden Sie ein Kommando kennen lernen, das es Ihnen gestattet, auf Adressen im gesamten Speicherraum zu schreiben (obwohl Sie niemals Glück haben werden bei dem Versuch, das ROM zu beschreiben, egal wieviel Sie lernen).

Ein Fenster zum Speicher

Sie können den Inhalt von 16 Speicherstellen mit dem Kommando:

```
WINDOW MEM
```

zur Anzeige bringen. Sofern Sie die Kommandos RC oder LC nicht anders definieren, lassen sie die Speicherbereiche $0800 - $0807 und $00 - $07 im Speicherfenster erscheinen. Die Zahlenbasis der dargestellten Werte läßt sich mit dem Kommando BASE MEM DEC auf dezimal umstellen. BASE MEM HEX schaltet zurück auf hex. Wenn Sie den Wert einer Speicherzelle ändern möchten, so gibt es drei Möglichkeiten:

Direktes Laden

Die schnellste Art, eine Speicherstelle zu beschreiben, ist das direkte Laden. Geben Sie die Adresse und den Wert, den Sie dort speichern wollen, durch ein Leerzeichen getrennt ein. Beides, die Adresse und der Wert, müssen in der momentanen Zahlenbasis des Monitors gültige Zahlen sein (die Zahlenbasis wird durch den zweiten Eintrag in der Statuszeile wiedergegeben - hex, falls Sie sie seit Programmstart nicht verändert haben). Um z.B. $CA in die Speicherstelle $806 zu schreiben, geben Sie ein:

```
806 CA
```

Haben Sie gesehen, wie sich der Wert im Speicherfenster geändert hat? Nett, nicht wahr? 804 3 schreibt eine 3 in die Speicherstel-

le $804. Um dezimale Zahlen zu benutzen, müssen Sie die Zahlenbasis des Monitors mit BASE MON DEC umstellen. Adressen und Daten müssen jetzt dezimal eingegeben werden. 2054 4 schreibt dann eine 4 in die Speicherzelle $806. Da wir jedoch fast ausschließlich hexadezimale Zahlen verwenden werden, schalten Sie bitte die Zahlenbasis zurück auf hex (BASE MON HEX).

Editieren des Speichers

Das direkte Laden funktioniert ganz gut für das schnelle Beschreiben einiger Speicherzellen. Wollen wir jedoch Daten in 50 aufeinanderfolgende Speicherzellen schreiben, ist es lästig, jedesmal die Adresse angeben zu müssen. Ein effizienterer Weg, mehrere aufeinanderfolgende Speicherzellen zu ändern, ergibt sich durch die EDIT-Funktion. Rufen Sie sie auf mit:

```
EDIT 800
```

Dies bewirkt das Editieren des Speichers beginnend bei Adresse $0800. Um nun den Inhalt der Adresse zu ändern, brauchen Sie nur eine Zahl (dies muß natürlich eine 8-Bit-Zahl sein, die in der momentanen Monitor-Zahlenbasis gültig ist) einzugeben. Die eingegebene Zahl ersetzt den vorherigen Wert, sowohl im Speicher als auch auf dem Bildschirm. Anschließend wird die Adresse um eins erhöht, und der Vorgang wiederholt sich. Es gibt eine Reihe von Tricks, die Sie beim ersten Tastendruck anwenden können: Die Linkspfeil-Taste bringt den Wert der vorherigen Speicherzelle in die Anzeige, die Rechtspfeil-Taste (oder RETURN) springt zur nächsten Zelle, ohne die Zahl zu ändern, die an der ersten Adresse gespeichert ist.

Ändern Sie die Werte, die an den Adressen $0800 - $0807 gespeichert sind, um in $11, $22, $33, $44, $55, $66, $77 und $88. Bewegen Sie sich in diesem Bereich mit den Pfeiltasten vor und zurück. Stimmen die Werte, die die EDIT-Funktion Ihnen liefert, mit den Angaben im Speicherfenster überein?

ESC beendet EDIT und bringt Sie zurück zum Monitor.

Wenn Sie gültige Speicherzellen beschreiben, die nicht im Fenster dargestellt werden, so erfolgt die Änderung zwar im Speicher, nicht jedoch auf dem Bildschirm. Es gibt 2 Kommandos, mit denen Sie die darzustellenden Speicherzellen bestimmen können: LC (change left column = ändere die linke Spalte) und RC (change right column = ändere die rechte Spalte).

Um die Speicherzellen $FF00 - $FF07 darzustellen, geben Sie ein:

```
LC FF00
```

Wenn Sie wollen, können Sie dieselben Speicherzellen in der rechten Spalte anzeigen lassen. Wir leben in einem freien Land.

Laden von der Diskette

Beim APPLE lädt das BLOAD-Kommando Daten in den Speicher, die in DOS 3.3 B-Dateien auf der Diskette vorliegen. Es kann auch dazu benutzt werden, auf schnelle Art und Weise Nullen in den Arbeitsbereich zu schreiben, um diesen aufzuräumen. Dies geschieht durch das Laden eines Files von der VC-Diskette, das nichts als Nullen enthält und daher ZEROS (Nullen) heißt. Versuchen Sie es jetzt einmal.

```
BLOAD ZEROS
```

Damit sind alle Bytes im Bereich von $0800 - $0BFF, dem Hauptarbeitsbereich von VISIBLE COMPUTER, sowie zusätzlich die Seiten 0 und 1 auf Null gesetzt worden. Im Speicherfenster sollte die Änderung sichtbar geworden sein.

Wie Sie sich sicherlich schon gedacht haben, gibt es ein Gegenstück zu BLOAD, nämlich BSAVE. Mit diesem Kommando können Sie wählbare Ausschnitte der Arbeitsbereiche in eine Datei Ihrer Wahl schreiben.

Das BSAVE-Kommando funktioniert jedoch nur im MASTER MODE, der später behandelt wird. Solange Sie noch kein VC-Meister sind, gibt es kein BSAVE. Versuchen Sie es ruhig.

Laden der Register

Wir haben bereits das BASE-Kommando kennengelernt, mit dem wir die Darstellung der Register verändern können. Es gibt auch eine Möglichkeit, den Inhalt der Register zu verändern. Geben Sie direkt neben dem Monitor-Prompt den Namen des Registers an, gefolgt von dem Wert, den Sie dort speichern wollen. Wie bei allen Monitor-Kommandos müssen Sie auch hier den Wert in der momentanen Zahlenbasis eingeben. Um z.B. den Programmzähler PC auf Adresse $089F zu setzen, geben Sie ein:

```
PC 89F
```

Sie dürfen keine Zahlen, die größer als $FF sind, in ein 8-Bit-Register und keine Zahlen, die größer als $FFFF sind, in ein 16-Bit-Register schreiben. Ändern Sie die Basis der Register, die Sie beschrieben haben. Erhalten Sie die gleichen Werte, die sich auf dem Papier oder bei Verwendung des Kalkulators ergeben?

Im Anhang finden Sie eine Liste aller VC-Kommandos. Obwohl darunter einige sind, die wir zunächst nicht benutzen werden, sollten Sie sich diese Liste jetzt schnell einmal ansehen.

Kapitel 6

Erste Programme

Lassen Sie uns zunächst einen sauberen Start für unser erstes Programm durchführen. Geben Sie hierzu das RESTART-Kommando ein, das den 6502-Simulator in die Ausgangsstellung versetzt. Dies ist nicht nötig, falls Sie VC gerade geladen haben.

VISIBLE COMPUTER bewirkt, daß sich der Bildschirm Ihres Rechners wie ein 6502-Mikroprozessor verhält. Der Simulator arbeitet sehr viel langsamer als der wirkliche 6502, genauer gesagt etwa eine Million mal langsamer. Darüberhinaus ist er transparent, so daß wir in ihn hineinsehen können, während er arbeitet. Dabei bleibt der reale 6502, der den simulierten 6502 arbeiten läßt, im Hintergrund.

00	y	p	0011 0000
00	x	s	ff
0800	pc	a	00
		ir	00
0000	ad		
		db	00
		dl	00

Eine Reise durch den 6502

Der 6502 hat acht 8-Bit-Register. Sie erinnern sich, ein Register ist wie ein Kästchen, in dem Zahlen gespeichert werden können. Die Abkürzungen ihrer Namen sind: A, S, P, X, Y, DL, DB und IR. Außerdem gibt es noch zwei 16-Bit-Register, PC und AD. Wir werden die Register einzeln besprechen, da sie mehr Individualität haben als eine typische Speicherzelle.

A	Das A-Register oder der Akkumulator ist, obwohl nicht besonders groß, das wohl wichtigste Register eines 6502. Es wird in nahezu jedem Programm eingesetzt.
S	Direkt darüber befindet sich S, das "Stack-Zeiger"-Register. S wird für Stack-Operationen benutzt.
P	Das P-Register (Prozessor Status) hat vermutlich die unnatürlichste Abkürzung aller 6502-Register. Es ist auch das einzige Register, das standardmäßig binär dargestellt wird, da wir mehr an den einzelnen Bits von P interessiert sind als an deren gemeinsamem Wert.
X, Y	Als nächstes kommen die 6502-Zwillinge, das X- und das Y-Register. Sie sind scheinbar austauschbar und werden sehr häufig verwendet, wenn auch nicht so oft wie das A-Register. Wegen ihrer Verwendung bei etwas, das indexierte Adressierung genannt wird, heißen sie auch Index-Register.
PC	Das große Register unter X ist der Program Counter (Programmzähler). Er dient dem 6502 als eine Ortsmarkierung, die ihn daran erinnert, wo im Speicher er sich befindet und welche Anweisung er als nächste ausführen soll. Fans des Program Counters hätten gute Gründe, zu behaupten, er wäre das wichtigste Register des 6502. Zumindest ist er doppelt so groß wie der Akkumulator.
DL	DL (Data Latch), der Daten-Zwischenspeicher, ist die Bus-Station des 6502, der Übergang zwischen den Daten, die vom und zum Speicher in den Prozessor hinein oder aus ihm hinausgehen. Es gibt kein Zeichen, das in den 6502 hineingelangen oder ihn verlassen könnte, ohne dieses Register passiert zu haben.
DB	DB, der Datenpuffer (Data Buffer), ist ein Platz, an dem wir eine Zahl mitten in einem Befehl für eine begrenzte Zeit unterbringen können, bis wir sie wieder brauchen.
IR	IR ist das Befehlsregister (Instruction Register). Hier legt der 6502 den Befehl ab, den er gerade ausführt. Es ist das Äquivalent zum Klemmbrett des Vorarbeiters der Versandabteilung, ein Platz, an dem eine Anweisung studiert ("dekodiert") wird, um herauszufinden, was sie bedeutet und wie sie ausgeführt werden kann.
AD	AD ist der Adressen-Zwischenspeicher (Adress Latch); es ist das Register, das die Adresse der Speicherzelle enthält, auf die bei Schreib- oder Leseoperationen zugegriffen wird.

Die beiden 16-Bit Register des 6502, AD und PC, haben die Eigenheit, sich manchmal wie ein großes 16-Bit-Register und dann wieder wie ein Paar 8-Bit-Register zu verhalten. Wenn sie in dieser Weise benutzt werden, nennt man die höherwertigen Hälften PCH und ADH (H von High) und die niederwertigen Hälften PCL und ADL (L von Low).

A, S, P, X, Y und PC sind Abkürzungen, auf die sich alle 6502-Programmierer geeinigt haben. Für die Namen der anderen Register, DL, DB, IR und AD, gilt dies nicht, sie wurden vom Autor dieses Buches vergeben. Das ist jedoch nicht weiter schlimm. Sie können 11 Bücher über 6502-Maschinensprache kaufen, von denen keines die Register DL, DB, IR und AD erwähnt. Dies liegt daran, daß der Programmierer sich um diese Register nicht zu kümmern braucht, wenn er 6502-Programme schreibt. Es gibt sie in jedem 6502, und sie sind unerläßlich für dessen Funktionieren; da sie jedoch temporäre Zwischenspeicherfunktionen erfüllen, braucht sich der Programmierer nicht mit ihnen zu befassen. VISIBLE COMPUTER dagegen simuliert die innere Arbeitsweise eines 6502 und kann deshalb nicht auf sie verzichten.

Wir wollen nun einiges von diesem Wissen in die Tat umsetzen. Lassen Sie uns das erste der Demonstrationsprogramme laden und ausführen, das, wie nicht anders zu erwarten, PROG1 heißt. Laden Sie PROG1 (zwischen dem "G" und der "1" ist kein Leerzeichen), die ganzen 2 Bytes, aus denen es besteht, mit dem folgenden Kommando in den Speicher:

```
BLOAD PROG1
```

Wenn Sie nichts anderes angeben, so lädt BLOAD die Daten ab Adresse $0800. PROG1 ist eine einfache Sache, die ein kleines Kunststück vollbringen wird: Es wird bewirken, daß eine $33 (51 dezimal, 0011 0011 binär) im Akkumulator erscheint. Ich weiß, daß Sie das einfach mit dem Monitor-Kommando A 33 erreichen können; - haben Sie bitte noch etwas Geduld mit mir.

Lassen Sie uns die Daten ansehen, aus denen PROG1 besteht. Setzen Sie das Fenster mit dem WINDOW MEM-Kommando auf den Speicher. PROG1 besteht aus einer $A9 an der Adresse $0800 gefolgt von einer $33. Hmmm... Das Programm soll den Akkumulator mit $33 laden, und eines der beiden Bytes im Programm ist eine $33. Da könnte eine Verbindung bestehen.

Die folgenden Nullen markieren das Ende von PROG1. Schließen Sie das Fenster mit CLOSE WINDOW. Wir wollen den ganzen Prozessor/Speicheraufbau bei unserem ersten Programm sehen. Als nächstes bringen Sie VC in seinen langsamsten, hilfreichsten Status mit dem Kommando: STEP 3.

Ich weiß, Sie sind ungeduldig anzufangen, aber bevor wir den Simulator auf PROG1 loslassen, betrachten wir die Inhalte der Register. Da wir VC frisch geladen haben, sind sie in ihrer Grundeinstellung. Die meisten, wenn auch nicht alle, enthalten Nullen. Kümmern Sie sich zunächst weder um das schlecht abgekürz-

te P mit seinem binären Inhalt noch um die $FF im Stack Pointer. Ich möchte Ihre Aufmerksamkeit lieber auf die $0800 richten, die im Program Counter gespeichert ist.

Diese Adresse gibt an, wo im Speicher der zum Leben erwachende 6502 den Befehl finden wird, den er ausführen soll. Dies hat nichts damit zu tun, daß wir PROG1 an dieser Stelle geladen habe. Würden wir eine andere Zahl in den Program Counter laden, etwa $1AFF, so würde der Simulator nicht PROG1 ausführen, sondern die unbekannten Daten, die er bei $1AFF vorfindet. Um den Simulator zu aktivieren, drücken Sie bitte RETURN, ohne vorher irgend etwas auf den Monitor-Prompt hin eingegeben zu haben. Die Dinge entwickeln sich jetzt schnell, seien Sie also aufmerksam.

Der Simulator

Dies ist unserer erster Ausflug in den 6502-Simulator. Er hat bei weitem nicht so viele Kommandos, um die Sie sich kümmern müssen, wie der Monitor. Er führt im wesentlichen Programme aus, während Sie einfach zusehen. Im Meldungsfenster wird "fetch" dargestellt (in "Computerschrift", was andeuten soll, daß es die "Gedanken" des 6502 sind, die hier dargestellt werden). Fetch (= holen) bedeutet, daß der 6502 einen Instruction Fetch, das Holen eines neuen Befehls, die erste Phase jeder Befehlsausführung, durchführt. Wenn der 6502 sich ein Byte geholt und in das Instruction Register gelesen hat, endet der Fetch-Zyklus, und die Ausführungsphase beginnt.

Die zweite Zeile im Meldungsfenster enthält eine mehr verkürzte Meldung:

T:PC->AD

Ins Deutsche übersetzt bedeutet dies: Transferiere: Den Inhalt des Program Counters in den Adresszwischenspeicher. Dies ist einer der Mikroschritte (microsteps), der Funktionsblöcke, aus denen der Maschinenbefehl zusammengesetzt ist. Die neun Mikroschritte von VISIBLE COMPUTER sind im Anhang zusammengestellt. Der Transfer-Mikroschritt, der kurz auf das Quell- und dann auf das Zielregister zugreift, ist der verbreitetste; er wird bei jedem Befehl mindestens zweimal verwendet.

Der Transfer wird in dem Moment durchgeführt, in dem Sie die Pause beenden, in der Sie sich gerade befinden. Wenn Sie "C" drücken, gelangen Sie in den Kalkulator, der einzigen Monitorfunktion, die vom Simulator her aufgerufen werden kann. Das Drücken der ESC-Taste versetzt Sie in den Pausenzustand zurück.

Starten Sie nun den Simulator neu mit der Leertaste. AD enthält jetzt $0800. Beachten Sie bitte, daß PC immer noch $0800 enthält. Ein Transfer beeinflußt niemals das Quellregister.

READ (Lesen) ist der nächste Mikroschritt des FETCH-Prozesses.

Ein Read geht sehr schnell, seien Sie also bereit. READ besteht aus folgenden Schritten:

1. Der Inhalt des Adresszwischenspeichers wird an den Adressbus des Speichers gelegt.
2. Der Speicher holt den Inhalt dieser Adresse und transferiert ihn auf den Speicherdatenbus.
3. Der Wert wird zum Datenzwischenspeicher des 6502 transferiert.

Bevor wir dieses READ stattfinden lassen, eine kurze Quizfrage: Welcher Wert wird von der Adresse $0800 gelesen werden? Antwort: $A9. Er wird sich in der einen Minute nicht geändert haben, seit wir ihn mit dem Monitor gelesen haben. Okay, drücken Sie die Leertaste.

Der 6502 befindet sich weiterhin im Fetch-Zyklus. Er hat dem Eingangskorb eine Anweisung entnommen, sie aber bislang nicht auf seinem Klemmbrett befestigt. Solange er das Byte nicht in seinem Instruktionsregister (IR) hat, weiß der 6502 nicht, um was für einen Befehl es sich handelt, geschweige denn, wie er zu beenden ist. Der nächste Schritt ist daher, die Anweisung ins Instruktionsregister zu laden, damit sie entschlüsselt und ausgeführt werden kann. Sobald die $A9 in IR ist, endet die Fetch-Phase, und die Ausführungsphase beginnt. Das "Fetch" im Meldungsfenster wird durch "LDA IMMED" ersetzt. Der 6502 sagt zu sich selbst, "Ich muß meinen Akkumulator mit dem nächsten Byte im Speicher laden". Es ist dies ein Load Accumulator, Immediate (direktes Laden des Akkumulators), auch bekannt als Anweisung Nummer $A9.

Er weiß nun, was er als nächstes tun muß. Erstens: Den Program Counter inkrementieren. Er enthält jetzt $0801. Den Counter zum Adressbus transferieren. Fühlen Sie schon das kommende READ? Hier ist es: Lesen des Inhalts von Stelle $0801 in den Daten-Zwischenspeicher. Kopieren der dort gefundenen Zahl, einer $33 (welche Überraschung!), in den Akkumulator.

Wir sind fast fertig. Es passiert noch etwas, das sich "CONDition FLAGS" nennt und das P-Register beeinflußt. Wir werden dieses Phänomen später besprechen. Der letzte Schritt besteht darin, den Program Counter zu inkrementieren. Dies dient nicht der momentanen Anweisung, sondern ist Vorbereitung für den nächsten Befehl. Wenn Sie sich im Monitor befinden, zeigt der Program Counter immer auf den nächsten auszuführenden Befehl, nicht auf das letzte Byte der gerade ausgeführten Anweisung.

Ein richtiger 6502 leistet sich nicht den Luxus, untätig herumzusitzen, während ein Monitor eine halbe Stunde lang aktiv ist. Er muß eine Anweisung nach der anderen ausführen, tick-tick-tick, hunderttausende von Malen in der Sekunde, ohne so etwas wie eine Pause, in der er sich auf die faule Haut legen kann. Nach jedem Befehl ist der Wert des Program Counters die Adresse des ersten Bytes der nächsten Anweisung.

Mit dem abschließenden Erhöhen des Program Counters ist der Befehl beendet, und wir landen wieder im Monitor. Die letzte Tat

des Simulators ist, uns die eben ausgeführte Anweisung im Fenster für disassemblierten Programmtext anzuzeigen. Betrachten Sie dieses Fenster zunächst als einen mysteriösen Schwanz der letzten fünf Anweisungen, die der Simulator ausgeführt hat.

Was würde wohl passieren, wenn wir jetzt den Simulator aktivieren würden? Drücken Sie RETURN und finden Sie es heraus. Er wird auf die $00 zugreifen, die in $0802 steht, sie sich ins IR-Register laden und untersuchen. $00 ist ein 6502-Maschinenbefehl, der sich BRK (Software Break = programmierbarer Abbruch) nennt und Sie gleich wieder zum Monitor zurückschickt, ohne etwas zu tun. Er ist ein Signal an den Simulator, daß das Programm zu Ende ist und wir zum Monitor zurückkehren wollen. Wir werden später noch mehr über diese einzigartige Anweisung lernen.

Ich gratuliere! Sie haben gerade Ihr erstes Programm beobachtet. Wenn Sie alles mitbekommen haben, haben Sie etwa 90% der Grundlagen der Maschinensprache gelernt. Bemühen Sie sich nicht, die genaue Reihenfolge der Mikroschritte im Kopf zu behalten. Wichtig zu wissen ist: Wenn der 6502 die Bytefolge $A9 $33 ausführt, so lädt er seinen Akkumulator mit $33.

Bevor Sie zur nächste Simulatorsitzung und komplexeren Programmen übergehen, vergewissern Sie sich, daß Sie verstanden haben, wie dieses Programm arbeitet. Wenn Sie den Program Counter mit PC 800 auf den Anfang von PROG1 zurücksetzen, können Sie dieses Programm noch einmal ablaufen lassen. Wenn Sie schon dabei sind, so ändern Sie doch den Inhalt der Speicherzelle $0801 von $33 zu - sagen wir - Ihrem Alter, so daß PROG1 die nützliche Aufgabe erfüllen kann, dem Akkumulator mitzuteilen, wie alt Sie sind.

Es geht direkt weiter

Bis jetzt kennen wir genau zwei der sechsundfünfzig 6502-Befehle: LDA, auch bekannt als $A9: "Lade den Akkumulator mit dem Byte, das diesem folgt", und BRK, $00: "Brich das Programm ab und kehre zum Monitor zurück". "LDA" und "BRK" sind keine zufällig gewählten Abkürzungen; es sind offizielle 6502-Mnemonics (sprich: Nehmon-iks). Ein Mnemonic ist eine Eselsbrücke, basierend auf der Theorie, daß es für den Menschen einfacher ist, das Laden des Akkumulators mit "LDA" zu verbinden als mit $A9. Selbstverständlich kann der 6502 mit "LDA" nichts anfangen; wenn Sie wollen, daß er seinen Akkumulator lädt, so müssen Sie ihm den Opcode (von operation code) $A9 geben. Jedem 6502-Befehl ist ein 3-buchstabiges Mnemonic zugeordnet. Einige dieser Abkürzungen sind besser als andere, alle sind sie jedoch besser zu behalten als eine Zahl.

Erinnern Sie sich daran, daß ich sagte, der Akkumulator sei das wichtigste Register des 6502? Dies macht LDA - $A9 zu einem Befehl, den man kennen muß. Laden ist gut und schön, aber wie

steht es damit, einen Wert, der im Akkumulator steht, irgendwo im Speicher abzulegen? Gibt es eine Möglichkeit, dies zu tun? Sie haben es erraten. Laden Sie PROG2.

PROG2 führt das Gegenteil von LDA ein: STA (Store Accumulator, Opcode $85). Dieser Befehl bewirkt, daß der Inhalt des Akkumulators in einer Speicherzelle Ihrer Wahl abgelegt wird. PROG2 wird zunächst den Akkumulator mit $66 laden (LDA $66) und dann in die Speicherzelle $43 schreiben (eine Adresse auf Seite Null). PROG2 ist länger als PROG1, es besteht aus gewaltigen 2 Befehlen, 4 Bytes. Sehen Sie es sich an, entweder mit WINDOW MEM oder mit EDIT. Es beginnt, genau wie PROG1, bei $0800. Achten Sie auf die $43 in $0803. Ein Zufall? Sie wissen es besser.

Nachdem Sie den Program Counter auf $0800 zurückgesetzt und sich vergewissert haben, daß das Speicherfenster geschlossen (WINDOW CLOSE) und STEP 3 aktiv ist, arbeiten Sie sich nun Schritt für Schritt durch dieses aus 2 Befehlen bestehende Programm. Wenn Sie das erste Mal zum Monitor zurückkehren, wird PROG2 erst zur Hälfte durchgeführt sein, da der Simulator nach jeder vollständigen Anweisung zum Monitor zurückkehrt. Drücken Sie RETURN, um die nächste Anweisung ausführen zu lassen. Achten Sie ganz besonders auf STA - $85. STA ist ein klein wenig komplizierter als LDA - $A9.

Wenn der 6502 die Zahl $85 auf seinem Klemmbrett-Register sieht, so weiß er, er muß, genau wie bei LDA, ein weiteres Byte lesen. Was er jedoch mit diesem zweiten Byte (der $43) tut, ist etwas ganz anderes.

Zunächst transferiert er es auf den Adressbus. Da AD ein 16 Bit breites Register ist und wir es mit einer 8 Bit breiten Zahl laden, wird das höherwertige Byte Null gesetzt. Auf diese Weise haben wir die Adresse $0043, eine Zero-Page-Adresse (Adresse auf Seite Null), gebildet. Das Anlegen einer Zahl an den Adressbus des Prozessors ist immer die Vorstufe zu einer Lese- oder Schreiboperation. Anschließend wird der Inhalt des Akkumulators in das Kreuzungsregister DL übertragen. Nun ist alles bereit für den WRITE-Mikroschritt.

Ein WRITE besteht aus den folgenden Schritten:

1. Der Inhalt des Adress-Zwischenspeichers wird auf den Speicher-Adressbus gelegt.
2. Der Inhalt des Daten-Zwischenspeichers wird an den Speicher-Datenbus gesandt.
3. Die Zahl wird an der gewählten Stelle im Speicher eingefügt.

Nach der Schreiboperation ist der STA-Befehl bis auf die abschließende Erhöhung des Program Counters, die diesen auf die nächste auszuführende Anweisung zeigen läßt, abgeschlossen. Diesmal erfolgt also keine Konditionierung der FLAGS.

Wenn Sie wieder im Monitor sind, so prüfen Sie bitte entweder mit WINDOW MEM (und LC 40) oder mit EDIT den Inhalt von Speicherzelle $0043, ob sie wirklich den Wert enthält, den PROG2 dorthin geschrieben hat. Lassen Sie das Programm einige Male laufen. Benutzen Sie dabei verschiedene Werte in $0801 und $0803. Die Werte der Opcodes in $0800 und $0802 dürfen Sie jedoch nicht ändern. Tun Sie dies trotzdem, so ändern Sie damit den Befehl LDA zu wer weiß was.

Das waren nun 3 Befehle, 53 verbleiben noch. Zehn weitere davon werden wir jedoch jetzt auf ganz einfache Weise kennenlernen.

Laden und Speichern mit anderen Registern

Der Akkumulator ist der Leithammel des 6502, manchmal jedoch ist es notwendig, einige der anderen Register zu laden oder aus ihnen zu speichern. Tatsächlich gibt es Befehle, die genau dies tun: LDY und STY für das Y-Register und LDX und STX für das X-Register.

Mnemonic	Opcode	Operation
LDX	$A2	Lade das X-Register
LDY	$A0	Lade das Y-Register
STX	$86	Speichere das X-Register
STY	$84	Speichere das Y-Register

PROG3 demonstriert all die gelernten Befehle. Laden Sie es, setzen Sie den Program Counter auf $0800 und führen Sie es schrittweise aus. Jede der neuen Anweisungen arbeitet genauso wie die entsprechende Anweisung für den Akkumulator. Wir fangen nun an, richtige Aufgaben zu lösen; drei aufeinanderfolgende Speicherzellen wurden mit $FF geladen. Großartig.

Alle bisher behandelten Befehle bestanden aus 2 Bytes: Ein Opcode-Byte, um dem 6502 mitzuteilen, was er tun soll, und ein zweites Byte, das die Anweisung vervollständigt. Der 6502 kennt auch Ein-Byte-Befehle, Anweisungen, die selbsterklärend sind, so daß kein weiteres Byte benötigt wird, um sie auszuführen. Von solchen Befehlen sagt man, sie seien "implizit" oder implied.

Die sechs Transfer-Befehle des 6502 sind Repräsentanten der Gruppe der impliziten Befehle. Sie dienen dazu, Daten zwischen den Registern auszutauschen.

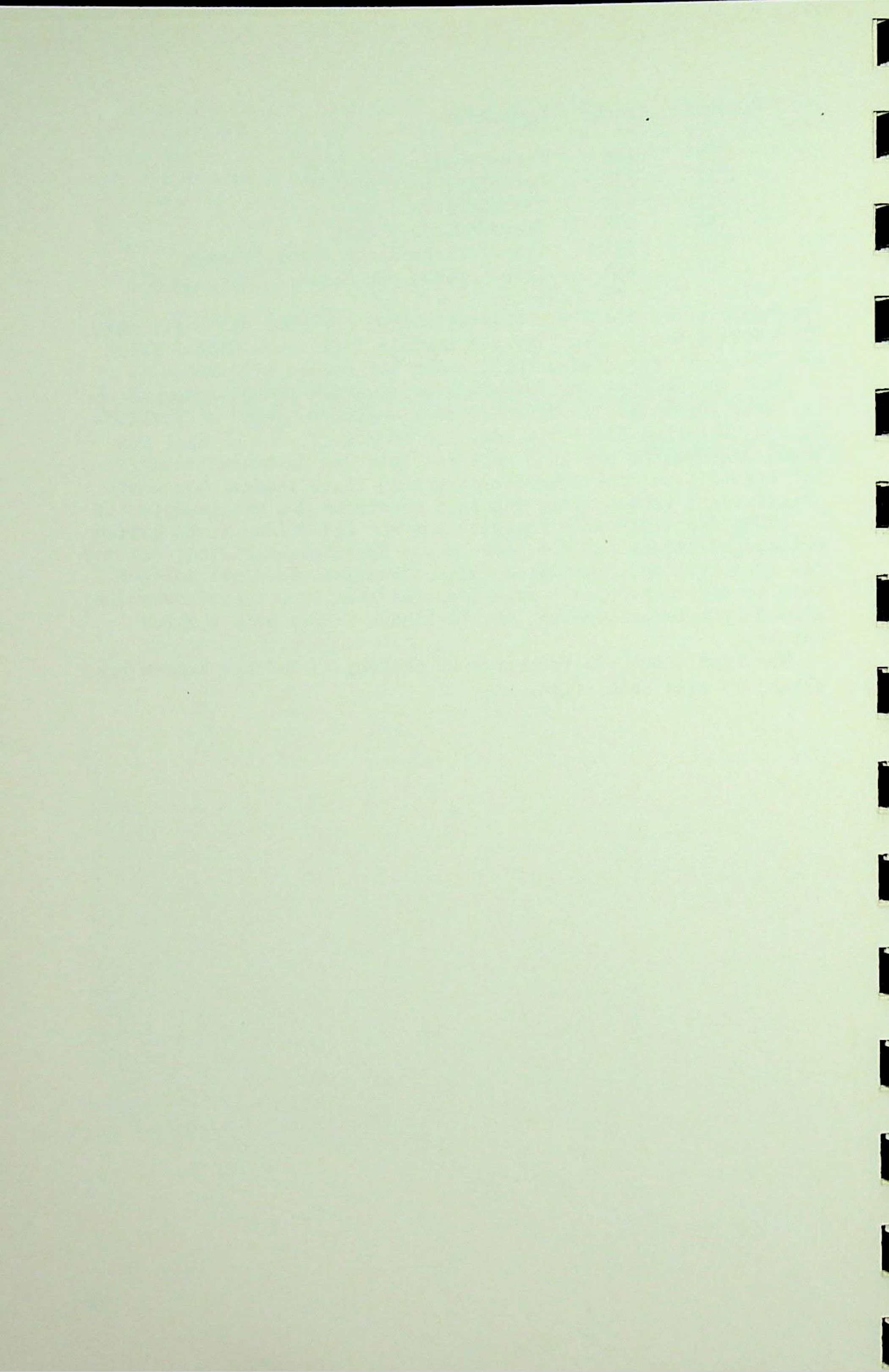
Mnemonic	Opcode	Operation
TAX	$AA	Transferiere A nach X
TAY	$A8	Transferiere A nach Y
TXA	$8A	Transferiere X nach A
TYA	$98	Transferiere Y nach A
TXS	$9A	Transferiere X zum Stack Pointer
TSX	$BA	Transferiere den Stack Pointer nach X

Verwechseln Sie die Transfer-Anweisungen des 6502 nicht mit dem "T:"-Mikroschritt. Ein "T:" ist nur ein Teil eines 6502-Befehls wie TYA oder, für diesen Fall, jedes beliebigen 6502-Befehls.

Zwei der Befehle der Transfer-Serie werden in PROG4 demonstriert. Vertrauen Sie darauf, daß die anderen genau so funktionieren. Bemerken Sie bitte, daß der 6502 weiß, daß er nach dem FETCH des Opcodes auf kein weiteres Byte des Speichers zuzugreifen braucht, um eine Transfer-Anweisung zu vollenden. Was wohin transferiert werden soll, entnimmt er direkt dem Opcode-Byte.

PROG4 führt dieselbe Funktion aus wie PROG3 (die nicht gerade welterschütternde Aufgabe, $FF in die Speicherzellen $40, $41 und $42 zu schreiben), nur tut es dies schneller. Es nimmt weniger Zeit in Anspruch, eine 1-Byte-Transfer-Anweisung auszuführen als eine 2-Byte-Ladeanweisung. Darüberhinaus ist es auch 2 Bytes kürzer.

Wir sind dabei, Fortschritte zu machen; 13 Befehle kennen wir schon, 43 sind noch offen.



Kapitel 7

Das Prozessor-Status-Register

Das Prozessor-Status-Register (P-Register) ist in gewisser Hinsicht eine Kuriosität innerhalb der 6502-Familie. Es hat nicht nur eine verwirrende Abkürzung, es ist darüber hinaus das einzige Register, dessen Inhalt uns mehr auf der Bit- als auf der Byte-Ebene interessiert. Mit anderen Worten, wenn sowohl P als auch A den Wert 0011 0011 enthalten, so werden wir den Inhalt von A als die Zahl $33 interpretieren; dagegen werden wir sagen, P enthält Einsen in den Positionen 0, 1, 4 und 5 und Nullen in den Positionen 2, 3, 6 und 7. Diese Bits sind so bedeutend, daß sie eigene Namen haben. Das P-Register wird daher standardmäßig binär dargestellt, so daß jedes Bit unter seiner Abkürzung steht.

Da wir gerade von Standards sprechen: Warum sind die Bits 4 und 5 gesetzt? Weil das der Wert ist, den Sie üblicherweise in diesem Register finden, wenn ein 6502 in einem APPLE läuft. Die vollen Namen dieser ausgeprägten Individualisten lauten:

Position		Name
7	N	Negativ-Flag
6	V	oVerflow-Flag
5		
4	B	Break-Flag
3	D	Dezimal-Modus-Flag
2	I	Interrupt-Disable-Flag
1	Z	Zero-Flag
0	C	Carry-Flag

Zwei Bemerkungen: Ein FLAG ist ein Bit von besonderer Bedeutung. Bit 5 des Prozessor-Status-Registers wird nicht benutzt. Es ist zwar offensichtlich vorhanden, nur können wir es nicht kontrollieren und sind auch nie an seinem Wert interessiert.

Befehle, die das P-Register beeinflussen

Es gibt implizite (aus einem Byte bestehende) Befehle, um viele, wenn auch nicht alle Flags des P-Registers zu setzen und zu löschen. "Setzen" und "löschen" sind handliche Verben, die den Akt beschreiben, ein Bit 1 oder 0 werden zu lassen. Statt vom "Löschen" werden wir in diesem Buch auch oft vom "Zurücksetzen" sprechen.

Mnemonic	Opcode	Operation
CLC	$18	Lösche das Carry-Flag
CLD	$D8	Lösche das Dezimal-Modus-Flag
CLI	$58	Lösche das Interrupt-Disable-Flag
CLV	$B8	Lösche das Overflow-Flag
SEC	$38	Setze das Carry-Flag
SED	$F8	Setze das Dezimal-Modus-Flag
SEI	$78	Setze das Interrupt-Disable-Flag

Diese Liste der Kommandos scheint unvollständig zu sein, da es keine Befehle zum Setzen und Löschen der Negativ- und Nullbits gibt und keinen zum Setzen des Overflow-Flags. Wie wir sehen werden, brauchen wir diese auch nicht.

Das Zero-Flag: Der 6502-Geschichtsschreiber

Das Z-Flag (Null-Flag) enthält eine einfach binäre Tatsache über früher ausgeführte Anweisungen. Es wird jedesmal konditioniert (gesetzt oder gelöscht), wenn der 6502 eine Lade- oder Transferanweisung ausführt. Wenn Sie eine Null in den Akkumulator laden, so wird Z gesetzt. Dies steht dem normalen Verständnis entgegen. Ich wiederhole also: Wenn Sie X, Y oder A mit Null ($00; 0000 0000) laden, so wird das Z-Bit gesetzt. Es bleibt gesetzt, bis ein anderer Lade- oder Transferbefehl ausgeführt wird, der einen Wert ungleich Null in ein Register lädt. Einmal gelöscht, bleibt es in diesem Zustand, bis das nächste Laden einer Null es wieder setzt.

Das Negativ-Flag

Auch das N-Flag wird bei jedem Lade- oder Transferbefehl konditioniert. Wenn Sie eine Zahl laden oder transferieren, bei der

Bit 7, das höchstwertige Bit, gesetzt ist, so wird N gesetzt. Jede Acht-Bit-Zahl, die größer als $7F ist, hat dieses Bit gesetzt (prüfen Sie es nach!). Entsprechend wird beim Laden oder Transferieren von Zahlen, bei denen dieses Bit 0 ist, das N-Flag gelöscht. N hat seinen Namen aufgrund der Tatsache erhalten, daß Bit 7 häufig dazu benutzt wird, negative Zahlen zu kennzeichnen. Wir werden die Situation bei Zahlen mit Vorzeichen später detaillierter beschreiben. Kurz gesagt lautet die Konvention: Wenn Bit 7 gesetzt ist, so ist die Zahl negativ; ist es gelöscht, so handelt es sich um eine positive Zahl. Wenn Sie keine Zahlen mit Vorzeichen benutzen, so können Sie das Verhalten des N-Flags außer acht lassen.

Dieser Konditionierungseffekt erlaubt es dem Programmierer, unter Verwendung von Befehlen, die wir im nächsten Kapitel kennenlernen werden, Bedingungen zu testen, die bei vorausgegangenen Lade- oder Transferbefehlen aufgetreten sind. Die Vorgehensweise besteht darin, den Status des Z- oder N-Bits zu prüfen und aufgrund dieser Prüfung zu entscheiden, was als nächstes zu tun ist. Es ist genauso wie beim IF/THEN-Befehl in BASIC.

```
          BASIC:  IF A=0 THEN GOTO 1000
                  A = A + 1
                  usw.

Maschinensprache: TXA
                  (If Accumulator = 0, GOTO XXXX)
```

Im nächsten Kapitel werden wir einen Befehl kennenlernen, der in die Klammer eingesetzt werden kann.

PROG5 demonstriert sowohl die impliziten Setz-/Lösch-Befehle als auch den Konditionierungseffekt beim Laden und Transferieren. Laden Sie PROG5. Bevor Sie es jedoch ausführen, werden wir eine Eigenschaft von VISIBLE COMPUTER kennenlernen, die es Ihnen ermöglicht, herauszufinden, was ein Programm tun wird, ohne es wirklich auszuführen.

Disassemblieren: Das L(ist)-Kommando

Nachdem Sie PROG5 geladen haben, geben Sie bitte ein: 800 L. Wie bei allen Monitor-Kommandos müssen Sie die Adresse von dem "L" durch ein Leerzeichen trennen. Das, was nun im Fenster für disassemblierten Programmtext erscheint, ist eine Vorschau der ersten fünf Befehle von PROG5. Im Gegensatz zu den Anweisungen, die der Simulator dort nach der Ausführung eines Befehles darstellt, wird die Adresse nicht invers dargestellt.

```
0800: 38    SEC
0801: F8    SED
0802: 78    SEI
0803: 18    CLC
0804: D8    CLD
```

'Disassemblieren' ist ein schreckliches Wort für den extrem nützlichen Vorgang, ein Maschinensprachprogramm in einer angenehmeren Form darzustellen als in schlichtem Hex. Die hexadezimale Darstellung ist auch vorhanden, Adresse und Inhalt - wir interessieren uns jedoch mehr für die für den Menschen aufbereitete Version des Befehls.

Ein disassemblierter Befehl besteht üblicherweise aus zwei Teilen: dem Mnemonic und dem Operanden (implizite Befehle wie z.B. DEX haben keinen Operanden). In "LDA 33" ist LDA das Mnemonic und 33 der Operand. Auf manchmal verzwickte Weise helfen beide dem Programmierer, herauszufinden, was der Befehl tut.

Sie können überall im Speicher disassemblieren; Sie dürfen jedoch keine vernünftigen Resultate erwarten, wenn die Zahlen, die dort gespeichert sind, nicht gerade ein Maschinenprogramm darstellen. Geben Sie z.B. ein: 4100 L. An dieser Adresse befindet sich ein Teil des VC-Programmes selbst. VISIBLE COMPUTER ist größtenteils ein BASIC-Programm, und BASIC-Programme ergeben weder für einen Disassembler noch für den 6502 einen direkten Sinn. Eine andere Möglichkeit, unsinnige Resultate zu erhalten, ist, das Disassemblieren an der falschen Stelle zu beginnen. Obwohl PROG5, das bei $0800 beginnt, ein gültiges Maschinenprogramm ist, erhalten Sie beim Listen Fragezeichen, wenn Sie an der Adresse $080A (APPLE) bzw. $C00A (C64) beginnen. Dies liegt daran, daß das erste Byte an dieser Adresse (das Daten-Byte der LDX-Anweisung, die bei $0809 (APPLE) bzw. $C009 (C64) beginnt) ein nicht definierter Opcode ist.

Wenn als Mnemonic "???" erscheint, so ist das immer ein Hinweis darauf, daß Sie etwas anderes als Maschinensprache disassemblieren. Von den 256 möglichen Acht-Bit-Werten, die 6502-Opcodes sein könnten, haben die Entwickler des 6502 nur 151 mit definierten Bedeutungen versehen. Wenn Sie dem Diasassembler nun einen der nicht implementierten Opcodes geben, etwa $02 oder $FF, so denkt er: "Nanu?" Probieren Sie das Disassemblieren in Bereichen, die bekanntermaßen Maschinensprache enthalten, wie etwa im Monitorbereich des APPLE ($F800 - $FFFF). Obwohl diese Bereiche nicht durchgehend aus Maschinensprachbefehlen bestehen, werden Sie viele Mnemonics und Operanden sehen, darunter eine ganze Reihe, die wir noch nicht besprochen haben.

Die Zeile für die nächste auszuführende Anweisung enthält, wie Sie sicher bereits vermutet haben, in disassemblierter Form entweder die Anweisung, die als nächste ausgeführt werden wird (wenn Sie im Monitor sind), oder die Anweisung, die gerade ausgeführt wird (wenn Sie im Simulator sind). Hierbei werden die Adresse

(die ja sowieso durch den Program Counter definiert ist) und der hexadezimale Wert der Anweisung weggelassen.

Die Anzeige in dieser Zeile wechselt immer dann, wenn der Program Counter oder der Speicherinhalt, auf den der Program Counter zeigt, geändert wird. Ändern Sie den Program Counter zu $0900. Die Befehlszeile sollte sich geändert haben. Schreiben Sie nun $18, den Opcode für CLC, an diese Adresse. Wieder sollte eine Änderung eintreten. Da wir als nächstes PROG5 ablaufen lassen wollen, setzen Sie PC zurück auf $0800.

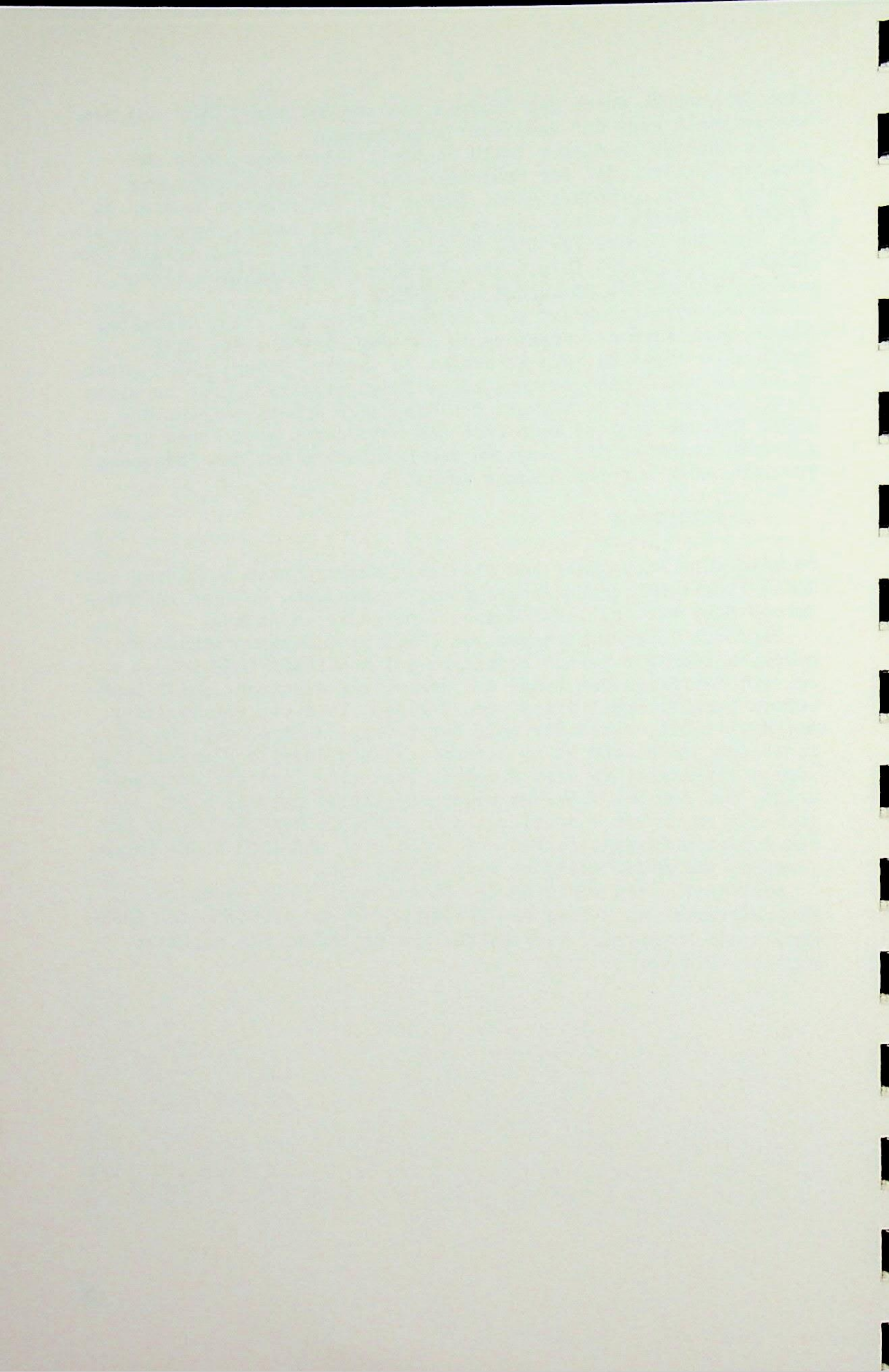
Ein Maschinenprogramm zu disassemblieren ist nicht dasselbe, wie es auszuführen - ebensowenig ist das Listen eines BASIC-Programmes dasselbe, wie es laufen zu lassen. Obwohl PROG5 jetzt, da wir seinen Inhalt kennen, keine Schwierigkeiten mehr bereiten wird, sollten Sie es nun mit dem Simulator durcharbeiten. Wer einen Drucker besitzt kann sich den Spaß daran noch etwas vergrößern, indem er die Ausgaben des Simulators mit dem folgenden Kommando auch auf dem Drucker ausgibt:

```
PRINTER ON
```

VC wird dann nach jeder vom Simulator ausgeführten Anweisung eine Zeile disassemblierten Programmtext ausdrucken. Weitere Informationen über das PRINTER-Kommando finden Sie im Anhang.

Die Set/Clear-Anweisungen von PROG5 sind unmißverständlich genug, achten Sie jedoch besonders auf den COND FLAGS-Mikroschritt der folgenden Lade- und Speicheranweisungen. Jeder Ladebefehl konditioniert die N- und Z-Flags. Wenn wir ein Register mit Null laden, setzt der 6502 das Z-Flag. Wenn es bereits gesetzt ist, so bleibt es in diesem Zustand. Jeweils gleichzeitig wird N verändert. Es wird gesetzt, wenn eine Ladeanweisung auftritt, bei der Bit 7 des geladenen Registers gesetzt wird, und gelöscht, wenn Bit 7 dabei gelöscht wird. Im Moment sollten Sie den Konditionierungsprozess nur beobachten und sich nicht darum kümmern, warum das alles so kompliziert ist.

Basteln Sie ein wenig an den Datenanteilen der Ladebefehle. Wie reagieren das Z- und das N-Flag auf Werte wie $FF oder $31? Finden Sie es heraus, wir treffen uns am Anfang des nächsten Kapitels wieder.



Kapitel 8

Verzweigungen: Entscheidungen treffen

Wenn es Ihnen ergangen ist wie mir, so hat Ihr erstes BASIC-Programm nicht gerade sehr viel für Sie getan. Es sah vermutlich etwa so aus:

```
100 INPUT "WIE HEISSEN SIE ";A$
110 PRINT "DAS IST EIN SCHÖNER NAME, ";A$
120 END
```

Wenn Sie bei Ihren Eingaben nicht gerade außerordentlich kreativ waren (DAS IST EIN SCHÖNER NAME, GROSSER MEISTER ALLER KLASSEN), so wurde es schnell langweilig. Meine erste Begegnung mit Entscheidungen und Schleifen war dagegen eine fast religiöse Erfahrung.

```
100 N = 0
110 PRINT N, N * N
120 N = N + 1
130 IF N <= 10 THEN 110
140 END
```

Irgendwie war das Konzept, eine Reihe von Anweisungen zu testen und, wenn nötig, zu wiederholen, faszinierend: "Mann, ich könnte die 10 in Zeile 130 ändern in eine 1000.. oder 1000000... Oder Zeile 110 so ändern, daß auch die 3. Wurzel gedruckt wird!"

Vereinfacht ausgedrückt, sind es das Testen und die Schleifenbildung, die das Programmieren von Computern ausmachen. Dies gilt für Maschinensprache genauso wie für BASIC. Um unser erstes "Entscheide und Wiederhole"-Programm auszuführen, benötigen wir einige neue Befehle: die Gruppe der Inkrement/Dekrement-Befehle und eine oder zwei Verzweigungsanweisungen.

Es gibt vier implizite Befehle, mit denen der Inhalt des X- und des Y-Registers inkrementiert (um eins erhöht) und dekrementiert (um eins vermindert) werden kann. Es sind die Befehle: DEX, DEY, INX und INY.

Mnemonic	Opcode	Operation
DEX	$CA	Dekrementiere das X-Register
DEY	$88	Dekrementiere das Y-Register
INX	$E8	Inkrementiere das X-Register
INY	$C8	Inkrementiere das Y-Register

All diese Anweisungen haben einen "Umklapp-Effekt". Wenn Sie ein Register dekrementieren, das $00 enthält, so erhalten Sie $FF. Inkrementieren Sie ein Register, das $FF enthält, so ergibt sich $00. Es besteht auch die Möglichkeit, Speicherzellen zu inkrementieren und zu dekrementieren. Seltsamerweise gibt es kein INC/DEC-Paar für den Akkumulator, obwohl es einen Weg gibt, dasselbe zu erreichen.

Genau wie die Lade- und Transferanweisungen konditionieren diese Befehle das N- und das Z-Flag. Wenn wir einen DEX-Befehl in einem Moment ausführen, in dem das X-Register $01 enthält, so erhalten wir $00, ein gesetztes Z- und ein gelöschtes N-Flag. Diese Eigenschaft der Dekrement- und Inkrementbefehle ist besonders bei Zählschleifen nützlich. Laden Sie das X- oder Y-Register mit der Zahl der gewünschten Schleifendurchläufe. Programmieren Sie anschließend die zu wiederholende(n) Operation(en). Dekrementieren Sie nun X, um anzuzeigen, daß Sie die Schleife einmal durchlaufen haben. Zum Schluß kommt ein Befehl, der das Z-Bit daraufhin testet, ob X bereits gleich null ist, und dann in Abhängigkeit vom Resultat dieses Testes auf den Anfang der Schleife zurückverzweigt, um den Vorgang zu wiederholen.

Diese Anforderungen werden von der BNE-Anweisung erfüllt. Sie spricht sich "Branch if Not Equal" (verzweige wenn ungleich), hat den Opcode $D0 und ist der folgenden BASIC-Zeile äquivalent:

```
IF A<>0 THEN GOTO 1000
```

BNE gehört zur Gruppe der 6502-Verzweigungsbefehle, von denen es noch sieben weitere gibt, jeweils zwei für jedes der vier testbaren Flags des P-Registers (C, N, Z und V). Ein Befehl testet, ob das jeweilige Bit gesetzt ist, der andere, ob es gelöscht ist:

Mnemonic	Opcode	Operation
BCC	$90	Verzweige wenn C gelöscht
BCS	$B0	Verzweige wenn C gesetzt
BEQ	$F0	Verzweige wenn Z gesetzt
BNE	$D0	Verzweige wenn Z gelöscht
BMI	$30	Verzweige wenn N gesetzt
BPL	$10	Verzweige wenn N gelöscht
BVC	$50	Verzweige wenn V gesetzt
BVS	$70	Verzweige wenn V gelöscht

Aufgrund der Art und Weise, wie diese Verzweigungsbefehle ausgeführt werden, sagt man, daß sie relative Adressierung verwenden. Eine Verzweigungsanweisung ist zwei Bytes lang; sie besteht aus einem Opcode-Byte (das dem 6502 sagt, welches Bit er auf welchen Zustand hin testen soll) und einem Offset-Byte, das ihm mitteilt, wohin er springen soll, wenn der Test positiv ausfällt. Dieses "Mitteilen, wohin er springen soll" ist ziemlich vertrackt und hat etwas damit zu tun, warum die Adressierungsart dieser Befehle relativ genannt wird.

Wenn die Bedingung, die bei dem Test spezifiziert wurde, wahr ist, so wird das zweite Byte der Anweisung dazu benutzt, einen neuen Wert für den Programmzeiger zu berechnen. Sie erinnern sich, der Programmzeiger ist die Stelle im Speicher, die bewirkt, daß der 6502 die Befehle nacheinander abarbeitet. Wenn der Test "falsch" ergibt (z.B. bei einer BNE-Anweisung, wenn das Z-Bit gesetzt ist), so wird der Programmzeiger ganz normal um eins erhöht, und die Dinge entwickeln sich so weiter, als hätte es überhaupt keine Verzweigungsanweisung gegeben.

Fällt der Test jedoch positiv aus, so wird der Programmzeiger modifiziert, indem das zweite Byte der Anweisung zu ihm hinzuaddiert wird. Enthält der Programmzeiger beispielsweise den Wert $805 (nachdem er gerade das zweite Byte einer BNE-Anweisung, sagen wir eine $10, aus dem Speicher gelesen hat) und der 6502 entscheidet, daß der Test positiv ausfällt, so bildet er einen neuen Wert für PC, indem er $10 zu der bereits in PC stehenden $805 addiert. Die nächste auszuführende Anweisung ist dann die bei $815 ($805 + $10) gespeicherte.

Bedeutet dies nun, daß Sprünge nur vorwärts erfolgen können? Nein, negative Sprungweiten sind auch möglich, es ist nur ein wenig schwieriger zu verstehen, wie sie berechnet werden. Wenn das Daten-Byte einer Verzweigungsanweisung $80 oder größer ist (beachten Sie: Bit 7, das Vorzeichen-Bit ist gesetzt), so weiß der 6502, daß er vom Programmzeiger etwas abziehen und nicht hinzuaddieren soll. Die Einzelheiten dieser Subtraktion heben wir uns für einen späteren Abschnitt auf. (Im Vorgriff sei nur soviel gesagt: $FF = -1, $FE = -2, $FD = -3...) Die Verzweigung kann also abhängig vom Daten-Byte in beide Richtungen erfolgen, indem der Programmzeiger entweder verkleinert oder vergrößert wird. Wir können in beide Richtungen 128 Bytes weit springen.

Dieses Springen wird in PROG6 demonstriert. Laden Sie es mit (B)LOAD und sehen Sie es sich mit dem "L"-Kommando an. Es beginnt damit, daß X mit einer $04 geladen wird; offensichtlich soll irgendetwas viermal getan werden. Anschließend folgen zwei Set/Clear-Befehle, die lediglich dazu dienen, dem Programm etwas Arbeit in der Schleife zu verschaffen. Danach kommt der neue DEX-Befehl. DEX konditioniert das Z-Flag; täte er das nicht, so würde das Programm nicht funktionieren. Die Verzweigungsanweisung BNE besteht aus einem Opcode-Byte ($D0) bei $0805 und einem Offset-Byte ($FB) bei $0806. Da $FB größer als $80 ist und daher Bit 7

gesetzt hat, handelt es sich um einen Rückwärtssprung; der Programmzeiger wird also um einen gewissen Betrag vermindert, wenn der Test "wahr" ergibt.

```
0800:a2 04     ldx #$04
0802:38        sec
0803:18        clc
0804:ca        dex
0805:d0 fb     bne $0802
```

Der VC-Disassembler weicht etwas von seinem normalen Weg ab, um Ihnen deutlicher zu machen, wo die Verzweigung enden wird, wenn der Test positiv ausfällt. BNE $0802 bedeutet: "Verzweige zu Speicherstelle $0802, wenn das Z-Flag gelöscht ist". Dies ist leichter zu verstehen, als wenn einfach nur BNE $FB angezeigt und es damit Ihnen überlassen würde, herauszufinden, wohin die Verzweigung weist.

Bei der ersten Ausführung der DEX-Anweisung wird X auf den Wert $03 reduziert. Dies ist ungleich null, daher wird das Z-Flag gelöscht, und der Test ergibt "wahr", wodurch die Schleife wiederholt wird. Schließlich, nach vier Wiederholungen, ergibt der Test "falsch", und das folgende BRK beendet das Programm.

Da dieses Programm zu seiner Durchführung erheblich länger braucht (obwohl es nicht länger ist) als die bisherigen Programme, will ich Ihnen nun erklären, wie die Geschwindigkeit des Simulators gesteuert werden kann. Bisher haben wir ausschließlich STEP 3 benutzt. Was bewirken nun die anderen STEP-Werte?

STEP 2 führt eine vollständige Anweisung aus, ohne bei jedem Mikroschritt anzuhalten. Sie können jedoch durch Drücken der Leertaste eine Pause einlegen. Nach einem vollständigen Befehl landen Sie wieder im Simulator.

STEP 1 entspricht STEP 2 bis auf die Tatsache, daß Sie zwischen den Befehlen nicht in den Monitormodus versetzt werden; stattdessen wird sofort mit der nächsten Anweisung begonnen. ESCape veranlaßt den Simulator dazu, nach Beendigung der laufenden Anweisung in den Monitormodus überzugehen.

STEP 0 ist der "Vollgas"- oder Hochgeschwindigkeitsmodus von VISIBLE COMPUTER. Bei ihm wird dadurch Zeit gespart, daß während der Ausführung nicht auf den Bildschirm geschrieben wird. Nur das Fenster für den disassemblierten Programmtext und der Bereich mit der nächsten Anweisung zeigen den aktuellen Stand an. Die Register zeigen nicht ihren wahren Wert, bis Sie zum Monitormodus zurückkehren. "Vollgas" und "Hochgeschwindigkeit" sind relative Begriffe. Der richtige 6502 arbeitet etwa eine Million mal so schnell wie VISIBLE COMPUTER bei "Vollgas".

Wenn Sie STEP 1, 2 oder 3 gewählt haben, so können Sie den Ablauf durch Drücken einer der Zahlentasten 1 bis 9 beschleunigen oder verlangsamen. 1 ist die schnellste und 9 die langsamste Arbeitsgeschwindigkeit. Drücken Sie nun RETURN; ich wünsche Ihnen fröhliche Schleifenabarbeitung.

Kapitel 9
Adressierungsarten

Wir sind so schnell vorgegangen, daß wir einige sehr gute Fragen übergangen haben, die Ihnen vielleicht gekommen sind. Eine davon ist: "Wenn es nur 56 Befehle gibt, warum existieren dann 151 verschiedene Opcodes?" Die Antwort ist mit etwas verknüpft, das man "Adressierungsarten" nennt.

Der 6502 ist ein Meister der Adressierungsarten; ja, er läßt einige seiner Artgenossen (wie den Z-80) in dieser Hinsicht reichlich kurzatmig erscheinen. Kurz gesagt, die Adressierungsarten bestimmen nicht, welche Anweisung ausgeführt werden soll, sondern wo und wie die Daten zu finden sind, die der Befehl benutzt. Bisher haben die Demonstrationsprogramme nur mit einem kleinen Teil der vielen möglichen Adressierungsarten des 6502 gearbeitet. Alle Ladebefehle benutzten direkte (immediate) Adressierung, das ist diejenige Form, die den 6502 anweist, ein Register mit dem nächsten im Speicher stehenden Byte zu laden. Alle Speicherbefehle benutzten die Zero-Page-Form, bei der eine Speicherstelle auf der Seite Null ($00 bis $FF) spezifiziert wird.

Was tun wir, wenn wir den Akkumulator nicht mit einer Zahl laden wollen, die wir schon kannten, noch ehe das Programm geschrieben wurde, sondern mit dem Inhalt einer Speicherstelle außerhalb des Programms? Der BASIC-Befehl

```
LET A = 14
```

entspricht der Art, in der wir den Akkumulator bisher geladen haben. Häufiger trifft man in BASIC jedoch auf die folgende Art von Befehl:

```
LET A = B
```

Die Durchführung eines solchen Ladebefehls in Maschinensprache erfordert ein LDA ganz anderer Art. Es gibt einen weiteren Opcode, der als LDA dekodiert wird (ich meine nicht den LDA-$A9, der das Laden des folgenden Bytes bewirkt). Es ist LDA-$A5; er bewirkt das Laden des Akkumulators aus der Speicherzelle, die im

nächsten Byte spezifiziert wird. Dies ist, wie ich zugebe, ein gerissener Gedanke, der jedoch entscheidend für Ihre Zukunft als weltberühmter Maschinensprachprogrammierer sein könnte.

PROG7, eine weitere Zwei-Byte-Spezialität, wird etwas Licht in dieses Dunkel bringen. Achten Sie darauf, daß der disassemblierte Text nicht genau dergleiche ist wie bei PROG1.

```
PROG1

0800(C000):A9 33     LDA #$33


PROG7

0800(C000):A5 33     LDA $33
```

Die Opcodes $A9 und $A5 übersetzt der Disassembler mit demselben Mnemonic, nämlich LDA, jedoch mit verschiedenen Operanden. Das "#" ist der Anhaltspunkt, um herauszufinden, was für eine Art LDA vorliegt. Nach 6502-Konvention bedeutet ein Nummernzeichen im Operanden, daß der Wert, der in der nächsten Speicherzelle enthalten ist, "direkt" geladen werden soll. Die Abwesenheit des Nummernzeichens im zweiten Beispiel sagt uns, daß die Zahl geladen werden soll, die in der durch das nächste Byte im Programm angegebenen Speicherzelle steht (in diesem Falle aus der Speicherzelle $0033).

Wir haben gerade einen neuen Opcode, $A5, kennengelernt, aber keinen neuen Befehl. $A5 ist LDA unter Verwendung der Zero-Page-Adressierung. $A9 ist LDA bei direkter Adressierung. Führen Sie nun PROG7 aus. Achten Sie besonders darauf, wie es die $0033 in AD erhält, nämlich entsprechend der STA $33-Anweisung in PROG2.

Was, es gibt noch mehr? Es folgt eine dritte Art LDA. Einige von Ihnen werden sich sicher schon gefragt haben: "Was muß ich tun, wenn ich den Akkumulator mit einem Wert laden will, der irgendwo im Speicher steht, aber nicht auf der Seite Null (Zero Page)? Etwa aus der Speicherzelle $A09 oder von $BFFF?"

Eine sehr gute Frage. Natürlich gibt es eine Möglichkeit, dies zu tun. Sie können jede der 65536 Speicherzellen mit absoluter Adressierung spezifizieren. Ein Befehl, bei dem absolute Adressierung benutzt wird, benötigt drei Bytes: ein Opcode-Byte und zwei Bytes, die die Speicherzelle bestimmen, die der Befehl benutzen soll.

Laden und disassemblieren Sie PROG8. PROG8 lädt den Akkumulator aus $B1C, wenn wir es gleich laufen lassen. Zunächst wollen wir uns jedoch den disassemblierten Programmtext genau ansehen.

```
0800(C000):AD 1C 0B   LDA $0B1C
```

Beachten Sie, daß das niederwertige Byte der Adresse zuerst erscheint. Beim 6502 werden Zwei-Byte-Werte immer so in aufeinanderfolgenden Speicherzellen abgelegt, daß das niederwertige Byte zuerst erscheint (an der niedrigeren Adresse). Es gibt hierfür keinen speziellen Grund; man hat lediglich einmal eine Abmachung getroffen und hält nun an ihr fest. Wieder arbeitet der Disassembler schwer, um uns das Leben zu erleichtern. Er ordnet den Operanden neu in die gewohnte Links-rechts-Form. Es ist nun einmal einfacher, den Sinn von "LDA $0B1C" zu erfassen als den von "AD 1C 0B".

Führen Sie nun PROG8 aus. Wie Sie sicherlich vermutet haben, läuft es länger als die anderen Formen von LDA, die wir benutzt haben. Der Datenpuffer wird als temporärer Speicher für das erste Byte der Adresse benutzt, bis wir bereit für es sind. Abgesehen von dem zusätzlichen Speicherzugriff und der Übertragung auf den Adressbus, läuft es jedoch genau wie die anderen beiden und endet mit dem Konditionieren der Flags und der abschließenden Erhöhung des Programmzeigers.

Das wär's also. Ein Befehl, LDA, und drei verschiedene Opcodes ($A9 für direkte Adressierung; $A5 für Zero Page und $AD für absolut). Kann man nun die absolute Adressierung auch dazu benutzen, um auf eine Speicherzelle auf der Zero Page zuzugreifen? Ja, das geht. Es gibt keine Regel gegen den Befehl:

```
0800(C000):AD 12 00  LDA $0012
```

Warum gibt es dann überhaupt die Zero-Page-Adressierung? Weil nur zwei statt drei Bytes benötigt werden. Absolute Adressierung benötigt mehr Speicherplatz und längere Zeit zur Ausführung. Zur Erzielung einer höheren Effizienz plazieren die 6502-Programmierer ihre am häufigsten verwendeten Variablen auf der Zero Page. Das Ergebnis ist, daß die Zero Page einen herausragenden Platz in der Speichertabelle des 6502 einnimmt. Obwohl es theoretisch möglich ist, die Zero Page zur Programmspeicherung zu benutzen, wird dies selten getan. Es wäre so, als verwendete man ein Viertel im Herzen Berlins dazu, Tomaten anzubauen.

Es gibt auf der Zero Page nur 256 Speicherzellen, und jeder will sie benutzen. Wenn Sie ein Maschinenprogramm schreiben, das von BASIC her aufgerufen wird, so müssen sie peinlich darauf achten, daß Sie nur solche Zero-Page-Adressen verwenden, die beim APPLE weder von APPLESOFT noch von DOS und dem MONITOR benutzt werden. APPLE II-Benutzer können auf dem Diagramm auf Seite 74 und 75 des 'APPLE Reference Manual' nachschauen, welche Speicherzellen sicher sind. Haben Sie einen APPLE //e, so schlagen Sie bitte die Seiten 66-67 des 'APPLE //e Reference Manual' auf.

Wenn Sie keines der genannten Bücher für Ihre Maschine besitzen, so sollten Sie sich schnellstens eines besorgen. Sie sind voller Fakten und Hinweise, die Sie unbedingt brauchen werden, wenn Sie 6502-Maschinenprogramme schreiben.

Die Lade- und Speicherbefehle für die Indexregister haben ebenso die besprochenen Adressierungsarten. Die folgende Tabelle faßt alle drei Modi für LDA, STA, LDX, STX, LDY und STY zusammen.

Mnemonic	Adressierungsart			Operation
	Abs.	Imm.	Seite 0	
LDA	$AD	$A9	$A5	Laden des Akkumulators
STA	$8D		$85	Speichern des Akkumulators
LDX	$AE	$A2	$A6	Laden des X-Registers
STX	$8E		$86	Speichern des X-Registers
LDY	$AC	$A0	$A4	Laden des Y-Registers
STY	$8C		$84	Speichern des Y-Registers

Es gibt keine Speicherbefehle mit direkter Adressierung, sie würden keinen Sinn ergeben. Sie wären wie der BASIC-Befehl: LET 14 = A.

Ein ominöses Wort zum Abschluß, bevor wir zum nächsten Kapitel übergehen. Ich sagte vorhin, daß der 6502 ein Meister der Adressierungsarten ist. Man wird kein Meister, wenn man nur drei Adressierungsarten für eine so bekannte Anweisung wie LDA beherrscht. Man wird es, wenn man acht davon verarbeiten kann.

Kapitel 10
Unterprogramme

Als nächstes sehen wir uns drei Instruktionen an, die, ähnlich wie die Verzweigungsinstruktionen, den Programmablauf verändern. Diesmal gibt es jedoch für den 6502 nichts zu entscheiden.

Die neuen Instruktionen sind: JMP (Jump, $4C), JSR (Jump to Subroutine, $20) und RTS (Return from Subroutine, $60). Zu allen dreien gibt es in BASIC direkte Gegenstücke:

Mnemonic	Opcode	Operation	BASIC
JMP	$4C	Springe zu einer anderen Adresse	GOTO
JSR	$20	Springe zum Unterprogramm	GOSUB
RTS	$60	Kehre aus dem Unterprogramm zurück	RETURN

JMP ist ein drei Byte langer Befehl, der die angegebene Adresse in den Programmzähler lädt und uns so zu jeder beliebigen Stelle im Speicher führen kann. Wie in allen Fällen absoluter Adressierung wird auch hier das Lo-Byte der Adresse zuerst angegeben.

Mit einem JMP können wir einen BRANCH-Befehl "verlängern". Verzweigungen sind auf 128 Bytes (vorwärts oder rückwärts) beschränkt, mit einem anschließenden JMP können wir jedoch soweit springen, wie wir wollen.

```
Statt          0810: BEQ $F000  (das geht leider nicht!)
               0812: ...

benutzt man    0810: BNE $0815
               0812: JMP $F000
               0815: ...
```

Laden Sie jetzt PROG9 und schauen Sie es sich an: Es steckt voller JMP-Instruktionen. Wenn Sie es starten, werden Sie feststellen, daß es in eine Endlosschleife mündet. Wie eine Katze,

die sich in den eigenen Schwanz beißen will, kommt dieses Programm nirgendwo an.

Solange Sie das Programm mit einem Simulator ausführen, ist das nicht weiter schlimm: Sie können den Kreislauf jederzeit mit ESC verlassen. Ein echtes 6502-Programm kann in so einem Fall jedoch nur noch mit der RESET-Taste unterbrochen werden. (Die RESET-Taste ist im Gegensatz zu den anderen Tasten direkt mit dem 6502 verbunden und hat eine starken Einfluß auf diesen).

Die Kombination JSR/RTS entspricht dem GOSUB/RETURN in BASIC (fast jede Programmiersprache hat ein Befehlspaar in dieser Art). Genau wie JMP veranlaßt auch JSR den Prozessor, mit seiner Tätigkeit an einer anderen Adresse im Speicher fortzufahren. Es gibt jedoch einen wichtigen Unterschied: Bevor die neue Adresse angesprungen wird, merkt sich der Prozessor die alte Anschrift: Der Programmzähler wird an einer sicheren Stelle im Speicher abgelegt. Nur dadurch wird es möglich, am Ende des Unterprogramms zu der alten Adresse zurückzukehren.

Eigentlich ist es nicht unbedingt notwendig, den genauen Mechanismus von JSR und RTS zu verstehen. Das gilt zumindest für BASIC-Programmierer, die so ein Geschenk annehmen, ohne lange nach dem Woher zu fragen. Assembler-Programmierer kommen nicht so leicht davon: sie müssen dem geschenkten Gaul schon mal ins Maul sehen, um auch die Fallstricke bei der Sache zu erkennen.

JSR und RTS benutzen eine Vorrichtung, die als "Stack" (Stapel) bezeichnet wird. Beim 6502 besteht der Stack aus dem Stackzeiger S und dem Speicherbereich von $100 - $1FF. Niemand kann Sie daran hindern, diesen Bereich für Ihre Programme und Daten zu verwenden - wir empfehlen jedoch, diesen Teil des Speichers dem Stack zu überlassen.

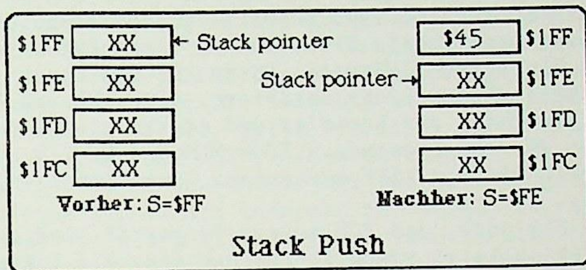
Die klassische Tellerstapel-Analogie

Den Stack können Sie sich wie einen dieser Tellerstapel vorstellen, die in manchen Cafeterias in die Theke eingelassen sind. Eine starke Feder unter den Tellern sorgt dafür, daß immer nur zwei oder drei Teller oben rausgucken. Wenn Sie einen Teller wegnehmen, dann schiebt die Feder den Rest ein Stück nach oben; wenn frische Teller draufgepackt werden, wird alles nach unten gedrückt. (Der unterste Teller sieht wahrscheinlich nie das Tageslicht.)

Sie haben hier eine reinrassige LIFO-Struktur (Last In, First Out: Als letzer 'rein, als erster 'raus) im Gegensatz zur Warteschlange vor der Theke, die dem FIFO-Prinzip folgt (First In, First Out: Wer zuerst kommt, mahlt zuerst). Um einen Stack als Zwischenspeicher zu benutzen, braucht man zwei Funktionen: PUSH (Leg einen Teller oben drauf) und PULL (Nimm den obersten 'runter). Dabei gilt es, immer daran zu denken, daß der erste PULL

den zuletzt aufgelegten Teller zurückgibt, der zweite PULL den vorletzten, usw.

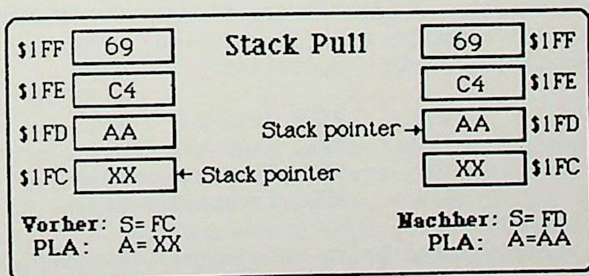
Was haben unser S-Register und die Speicherseite mit einer Cafeteria zu tun? Die Teller entsprechen den Bytes, die auf den Stack gepackt werden (ein 6502 kann maximal 256 Bytes verkraften), und anstelle einer Feder, die die Bytes im Bereich $100 - $1FF verschieben würde, benutzen wir einen Zeiger (Stack-Pointer), der auf das zuletzt abgelegte Byte zeigt. (Genaugenommen zeigt er auf den ersten freien Platz im Stack.)



In dieser Abbildung zeigt der Stack nach unten, aber wenn Sie das Buch einfach umdrehen, ist die Welt wieder in Ordnung. Wenn das Stack-Register $FF enthält, dann schreibt ein PUSH-Befehl das nächste Byte in Adresse $1FF, und der Wert des Stack-Pointers wird um eins (auf $FE) vermindert. Eine PULL-Operation erhöht zuerst den Wert des Stack-Pointers um eins und liest dann das Byte wieder aus.

PUSH in Mikroschritten

1) Übertragen des Stack-Pointers nach ADL (bei Stack-Operationen ist ADH immer gleich 1)
2) Übertragen des Registerinhalts, der "gepusht" werden soll, in den DATA LATCH
3) Schreiben
4) Stack-Pointer dekrementieren



PULL in Mikroschritten

1) Stack-Pointer inkrementieren
2) Stack-Pointer nach ADL
3) Lesen
4) Inhalt des DATA LATCH ins entsprechende Register

Jetzt sind Sie verwirrt genug, lassen Sie uns also PROG10 ansehen: Dieses Programm ruft ein Unterprogramm an der Adresse $A00 auf, das die Register X und Y mit dem Wert $FF lädt. Ein zweites Unterprogramm an Adresse $900 schreibt anschließend den Registerinhalt in zwei benachbarte Bytes auf der Zero Page.

Beachten Sie bitte: JSR packt Daten auf den Stack (und zwar die beiden Hälften des Programmzählers, also PCH und PCL), die RTS-Instruktion holt sie herunter und steckt sie wieder in den PC. Benutzen Sie das Kommando WINDOW MEM und RC 1F8, um sich den Speicherbereich $1F8 - 1$FF anzusehen. Hier findet die Action statt!

Beachten Sie auch, daß PCH zuerst "gepusht" wird und deshalb vom RTS-Befehl zuletzt "gepullt" werden muß. RTS inkrementiert auch den Programmzähler um eins, damit er auf die nächste Instruktion nach dem JSR-Befehl zeigt. Ein PULL-Befehl löscht übrigens das Byte nicht aus dem Speicher; es bleibt da stehen, bis es durch das nächste PUSH überschrieben wird.

Verschachtelung

Das Unterprogramm an Adresse $900 zeigt, wie eine Subroutine eine zweite aufrufen kann. Das RTS an $982 läßt uns zunächst in die Routine an $900 zurückkehren, mit dem nächsten RTS geht's zurück ins Hauptprogramm. Mit seinem 256-Byte-Stack erlaubt der 6502 bis zu 128 verschachtelte Unterprogrammaufrufe, ohne den Rückweg zu verlieren.

Noch mehr Stack

Es gibt vier weitere Instruktionen, die den Stack benutzen:

Mnemonic	Opcode	Operation
PHA	$48	PUSH Akku auf Stack
PLA	$68	PULL Akku vom Stack
PHP	$08	PUSH Prozessorstatus
PLP	$28	PULL Prozessorstatus

Wir werden vorerst woohl keine Gelegenheit bekommen, mit PHP und PLP herumzuspielen, obwohl es die einzige Möglichkeit ist, das Statusregister zu lesen oder zu verändern.

Aber PHA und PLA eignen sich prima dazu, Zwischenergebnisse abzulegen, ohne gleich ein Register oder eine Speicherstelle zu verschwenden. Wenn Sie einen Wert im Akkumulator haben, den Sie noch weiterverarbeiten wollen, Sie aber den Akku zwischendurch für etwas anderes brauchen, dann ist der Stack die ideale Zwischenablage.

Zwei Dinge müssen jedoch beachtet werden: Erstens, die Größe des Stacks ist begrenzt. Mehr als 256 Bytes können Sie nicht ablegen, sonst überschreibt er sich selbst. Wenn Sie sich den Stack auch noch mit DOS und BASIC teilen müssen (wenn Sie z.B. Maschinenprogramme von BASIC aus aufrufen), dann ist dort noch weniger Platz. Zweitens, wenn Sie sich innerhalb eines Unterprogramms befinden, dann dürfen Sie die Rücksprungadresse für das RTS nicht gefährden. Wenn Sie mehr auf den Stack packen, als Sie nachher herunterholen (oder Bytes "pullen", die Ihnen gar nicht gehören), dann führt das RTS Sie anschließend in die Wüste!

PROG11 zeigt die Stack-Behandlung. Das Unterprogramm an Adresse $900 ist eine grauenhaft langsame Warteschleife. Wie können wir sie wieder verlassen (zum Hauptprogramm zurückkehren)?

Offensichtlich soll das X-Register auf null heruntergezählt werden. Wir könnten das Programm mit Gewalt anhalten und eine 1 ins Register schreiben (von 1 ist es dann nicht mehr weit bis zur 0) oder im Stack rumwühlen, bis wir die Rückkehradresse finden und sie in den Programmzähler laden (plus 1 natürlich).

Eine bessere Möglichkeit bietet das POP-Kommando von VC. POP lädt die obersten beiden Bytes vom Stack in den Programmzähler und korrigiert den Stack-Zeiger entsprechend. Wenn Sie sich beim Testen eines Programmes in einer Subroutine verirrt haben und nicht mehr wissen, wo Sie hergekommen sind, dann hilft Ihnen POP aus dieser mißlichen Lage.

Das Unterprogramm an Adresse $A00 demonstriert, wie man PHA und PLA nicht benutzen sollte! Zum Zeitpunkt des RTS enthält der Stack eine Mischung aus Rückkehradresse und altem Akku-Müll. Pfui Teufel!

Indirekte Sprünge

Für den JMP-Befehl gibt es noch eine zweite Adressierungsart, die sogenannte indirekte (Opcode $6C). Das Mnemonic hierfür ist:

```
JMP ($2000)
```

Wie beim normalen JMP wird eine andere Adresse angesprungen und dort weiter gearbeitet. JMP ($2000) springt jedoch nicht zur Adresse $2000, sondern zu der Adresse, die an $2000 und $2001 abgespeichert ist. Wenn Adresse $2000 z.B. den Wert $F0 enthält und $2001 den Wert $FE, dann wird der Programmzähler mit $FEF0 geladen, und es wird dort weitergearbeitet.

Die ganze Sache ist eine Stufe raffinierter als beim normalen JMP; Sie dürfen sich daher jetzt zu Recht ein wenig schwindlig fühlen!

Der 6502 ist ein großartiger Mikroprozessor, doch die Jungs bei MOS TECHNOLOGY haben einen kleinen Fehler in die JMP-Indirekt-Instruktion eingebaut. (Ja, Hardware kann auch Fehler enthalten!) Wenn die indirekte Adresse genau auf einer Seitengrenze liegt, dann wird der Sprung falsch berechnet: JMP ($20FF) holt sich die Sprungadresse von $20FF und $2000 statt von $20FF und $2100. Diesen Fehler haben wir originalgetreu auch im VC eingebaut!

PROG12 enthält einen indirekten Sprung, JMP ($A00). Beim ersten Mal landen wir bei $810, beim nächsten Mal aber ganz woanders.

Warum sollten Assembler-Programmierer überhaupt Unterprogramme verwenden? Aus den gleichen Gründen, aus denen sie auch der BASIC-Programmierer benutzt: Erstens ist es effizient, gleiche Aufgaben an verschiedenen Stellen des Programms erledigen zu können, ohne den betreffenden Programmteil immer duplizieren zu müssen. Und zweitens dient es der Übersichtlichkeit!

Beim typischen Spaghetti-Code, voller JMP-Befehle (oder GOTO), haben Sie schon am nächsten Tag Schwierigkeiten, Ihr eigenes Programm nachzuvollziehen. Ein guter Programmierer benutzt Unterprogramme sehr großzügig, auch wenn sie nur einmal aufgerufen werden.

Das ideale BASIC-Programm:

```
100 GOSUB 1000
110 GOSUB 2000
120 GOSUB 3000
130 GOSUB 4000
140 GOTO 110
```

Das ideale Assembler-Programm:

```
      JSR $1000
LOOP: JSR $2000
      JSR $3000
      JSR $4000
      JMP LOOP
```

Kapitel 11
Zwei nützliche Befehle: ADC/SBC

Bisher haben wir Befehle kennengelernt, die hauptsächlich dem Vergnügen des Programmierers dienen. Wir haben Register geladen und wieder abgespeichert, sind hoch und runter, weit weg und wieder zurückgesprungen, aber ein Ergebnis haben wir nicht vorzuzeigen. Ein 6502, der ausschließlich die bisher bekannten Befehlen hätte, wäre wie ein Auto mit einer tollen Stereoanlage, plüschigen Sitzen, aber ohne Motor!

In diesem Kapitel werden wir zwei echte Arbeitspferde kennenlernen: ADC (Add with Carry, Addiere mit Übertrag) und SBC (Subtract with Borrow, Subtrahiere mit Anleihe). Es gibt sie in allen drei Adressierungsarten, die wir bisher kennen:

Mnemonic	Absolut	Direkt	Zero Page	Operation
ADC	$6D	$69	$65	Addieren
SBC	$ED	$E9	$E5	Subtrahieren

Wir haben die besondere Bedeutung des Akkumulators schon erwähnt, ohne sie jedoch richtig begründen zu können. Jetzt erfahren Sie es: ADC und SBC funktionieren nur mit dem Akku und mit keinem der anderen Register!

Um $23 und $14 zu addieren, laden wir den Akku mit $23 und benutzen ADC 14. Das Ergebnis, $37, steht anschließend im Akkumulator. (Ergebnisse von Rechenoperationen akkumulieren hier). Der Akku ist immer zur Hälfte an der Operation beteiligt und enthält schließlich das Ergebnis.

ADC addiert einfach zwei Acht-Bit-Zahlen. Kinder lernen das schon in der zweiten Klasse. Ein bißchen komplizierter wird es durch das CARRY-Bit (Überlauf-Bit). Der 6502 braucht das Carry-Bit, falls das Ergebnis der Addition größer wird, als es der Akkumulator fassen kann.

Reicht ein Bit dazu wirklich aus? Addieren Sie einmal die beiden größten Acht-Bit-Zahlen:

```
   $FF
+  $FF
------
= $1FE    (1 1111 1110)
```

Offensichtlich ja! Jedesmal, wenn die Addition ein Ergebnis größer als 255 liefert, wird das Carry-Bit gesetzt.

```
   $7F                      $31
 + $82                    + $16
 -----                    -----
 = $01, Carry = 1         = $47, Carry = 0
```

ADC verändert auch das Z- und das N-Flag entsprechend. Wenn als Ergebnis eine Null im Akku erscheint, dann wird das Z-Flag gesetzt; wenn das 7. Bit im Akku gesetzt ist, dann wird das N-Flag auf 1 gesetzt. In allen anderen Fällen werden die beiden Status-Bits gelöscht.

Das Carry-Bit wird nicht nur für das Ergebnis der Addition benutzt, sondern es wird auch selbst mitaddiert. Wenn das Carry-Bit vorher gesetzt war, dann fällt das Ergebnis um eins höher aus als sonst. Das ist ein wenig lästig, wenn wir nur mal schnell zwei Bytes addieren wollen, denn wir müssen vorher das Carry-Bit mit Hilfe eines CLC (Clear Carry) explizit löschen. Bei komplexeren Operationen ist dies jedoch eine sehr nützliche Einrichtung, wie wir gleich sehen werden.

PROG13 demonstriert die ADC-Instruktion. Spielen Sie ein wenig damit herum, bis Sie die Funktion des Carry-Bits und die Veränderungen der Status-Bits N und Z wirklich verstehen.

Arithmetik der grossen Zahlen

Wenn wir Zahlen größer als 255 addieren wollen, dann erweist sich das Carry-Bit als ausgesprochen nützlich. Obwohl der Akku nur 8 Bits verarbeitet, können wir größere Zahlen darstellen, in dem wir zwei oder mehr zusammenhängende Bytes dafür benutzen. Die größte Zahl, die sich mit zwei Bytes darstellen läßt, ist:

$$2^{16} - 1 = 65535$$

und mit dreien:

$$2^{24} - 1 = 16777215$$

Wir kommen so schnell in brauchbare Größenordnungen.

PROG14 addiert zwei Zwei-Byte-Werte, es benutzt dazu die Zero-Page-Varianten von ADC, LDA und STA. Bevor Sie es starten, müssen Sie (mit EDIT) zwei Zahlen in die Zero Page schreiben. Probieren Sie:

```
   $13FC  (A)
 + $4597  (B)
 -------
 = $????  (C)
```

Sie können das ganze vorher mit dem Kalkulator (CALC) ausrechnen, um das Ergebnis nachher vergleichen zu können.

Wir benutzen die Adressen $00 und $01, um A zu speichern, und $02 und $03 für B. Setzen Sie $04 und $05 auf null, sie sollen anschließend das Ergebnis (C) enthalten. (Nicht vergessen: das LSB immer zuerst, also $00 = $FC und $01 = $13, etc.)

Probieren Sie das Programm auch mit anderen Werten aus. Was passiert, wenn das Ergebnis größer wird, als 16 Bits verkraften können? Reicht das Carry-Bit auch in diesem Fall aus?

Subtraktion

Die Instruktion SBC (Subtrahiere mit Borrow) funktioniert ähnlich wie ADC. Der erste Operand befindet sich im Akku, der zweite irgendwo im Speicher, das Ergebnis steht anschließend im Akkumulator:

```
LDA 14
SBC 02
```

Kompliziert wird es erst durch das BORROW-Bit (Anleihe), - vor allen Dingen deswegen, weil es überhaupt kein Borrow gibt! Borrow ist das 'Gegenteil' vom Carry-Bit. Ist C = 1, dann ist Borrow per Definition gleich 0 und umgekehrt. (Verwirrend? Allerdings!)

Nehmen sie folgende Subtraktion:

```
   7
 - 2
 ---
```

Um sie auf dem 6502 auszuführen, laden wir 7 in den Akkumulator und benutzen SBC 02. Wenn das C-Bit vorher gesetzt war, dann ist das Ergebnis 5, denn C-Bit gesetzt bedeutet Borrow = 0. Wenn C vorher 0 war, dann erhalten wir eine 4 als Ergebnis.

Auch hier wird das Carry-Bit durch das Ergebnis verändert. Wann immer wir eine größere Zahl von einer kleineren subtrahieren, erhalten wir einen Borrow (sprich: das C-Bit wird gelöscht).

```
  $14            $14            $14
- $22          - $12          - $14
-----          -----          -----
 -$0E            $02            $00

Borrow = 1     Borrow = 0     Borrow = 0
(Carry = 0)    (Carry = 1)    (Carry = 1)
```

PROG15 demonstriert den SBC-Befehl und seine "verdrehte" Benutzung des Carry-Bits.

```
PROG15

SEC                 (Borrow gelöscht, da Carry gesetzt)
LDA #$07
SBC #$02
CLC                 (Borrow gesetzt)
LDA #$07
SBC #$02
LDA #$14
SBC #$22
```

Experimentieren Sie mit verschiedenen Werten.

Die wahre Schönheit der Borrow-Hilfskonstruktion wird erst bei mehrstelligen Subtraktionen sichtbar. PROG16 subtrahiert die Zwei-Byte-Zahl an Adresse $02 und $03 von der Zahl in $00 und $01. Das Ergebnis steht wieder in $04,$05. Mit Hilfe von EDIT geben Sie folgende Aufgabe ein:

```
  $73A1
- $46B1
-------
```

Starten Sie das Programm. Das Carry-Bit ist nach Ausführung des Programms gesetzt, Borrow ist also gleich null. Da $46B1 kleiner als $73A1 ist, haben Sie das natürlich gar nicht anders erwartet, oder? Spielen Sie solange mit verschiedenen Zahlen herum, bis Sie den Zustand des (imaginären) Borrow-Bits richtig vorhersagen können.

Multiplikation und Division

Der 6502 hat leider keine eingebauten Multiplikations- oder Divisionsbefehle wie manche der neueren Mikroprozessoren (68000, 8086, etc.). Aber durch geschickte Anwendung von Addition und Subtraktion können wir dasgleiche erreichen.

Um n mit m zu multiplizieren, können wir m n-mal zu sich selbst addieren. Um n durch m zu dividieren, zählen wir, wie oft wir m von n subtrahieren können. Das klingt umständlich und ist selbst für schnelle Kopfrechner nicht zu empfehlen, aber so ein flinker Kobold wie der 6502 macht das hundertmal, bevor Sie einmal mit den Augen gezwinkert haben.

Beispiel: 12 * 4 entspricht

12 + 12 + 12 + 12

oder

4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4

PROG17 ist ein Acht-Bit-Multiplikationsprogramm. Die Werte in $00 und $01 werden miteinander multipliziert und in $02,$03 abgelegt. Probieren Sie selber aus, daß zwei Bytes für das größte Produkt aus zwei Acht-Bit-Zahlen ausreichen. Beim Ausprobieren sollten Sie den Wert in $01 ziemlich klein halten (unter $10), sonst schlafen Sie während der Ausführung ein.

PROG18 dividiert den Wert in $00 durch den Inhalt von $01. $02 enthält anschließend den Quotienten und $03 den Rest.

Diese beiden Programme sind armselige Beispiele für Multiplikation und Division in Maschinensprache. Es gibt viele andere Verfahren, bessere und vor allem schnellere.

Vergleiche

Ein wichtiger Befehl zum Aufbau von Programmschleifen ist CMP (Compare, Vergleiche Akkumulator mit Speicher). CMP subtrahiert den Speicherinhalt vom Akku (oder vom X- oder Y-Register, es gibt auch ein CPX und ein CPY). Es setzt die N-, Z- und C-Flags entsprechend, verändert den Akku jedoch nicht. Wofür kann eine Subtraktion gut sein, die gar nichts subtrahiert? Wir werden es gleich sehen!

Mnemonic	Absolut	Direkt	Zero Page	Operation
CMP	$CD	$C9	$C5	Vergl. Speicher mit Akkumulator
CPX	$EC	$E0	$E4	Vergl. Speicher mit X-Register
CPY	$CC	$C0	$C4	Vergl. Speicher mit Y-Register

Bevor wir den CMP-Befehl demonstrieren, ein kleiner Exkurs.

Sie haben sicher schon einmal den Befehl PEEK(-16384) in einem BASIC-Programm gesehen. Speicheradresse -16384 (oder auch $C000)

ist die Stelle, an der die Tastatur des APPLE II angeschlossen ist. Wenn ein Programm (BASIC oder Assemblersprache) dort nachsieht, kann es feststellen, welche Taste zuletzt gedrückt wurde. Fast allen Tasten ist eine bestimmte Zahl zugeordnet. $0D ist die RETURN-Taste, $1B ist ESCAPE, $32 eine "2", $54 ein "T". Manche Tasten (SHIFT, CTRL) haben keinen eigenen Wert, sondern verändern den Code, der von den anderen generiert wird.

Wenn der 6502 die Adresse $C000 ausliest, dann findet er den ASCII-Code (ASCII = American Standard Code for Information Interchanging) der zuletzt gedrückten Taste. Den kompletten ASCII-Zeichsatz finden Sie im Anhang. (Sie haben ihn wahrscheinlich auch in 4 oder 5 anderen Handbüchern abgedruckt Zweck stehen, aber man kann nie genug davon haben.)

Der ASCII-Code wird übrigens auch von den meisten anderen Computern und Peripheriegeräten benutzt, es ist deshalb ziemlich einfach, z.B. einen EPSON-Drucker an den APPLE anzuschließen. Beide sind sich sofort darüber einig, welcher Code welchen Buchstaben darstellen soll.

Die Geschichte mit der Tastatur hat aber noch einen kleinen Haken. Wenn Sie ein Taste betätigen, dann bleibt sie etwa eine Zehntelsekunde gedrückt. Der 6502 ist so schnell, daß er die Taste lesen und das Ergebnis verarbeiten kann (z.B. ein Semikolon auf den Bildschirm schreiben), und wenn er zurückkommt, ist die Taste immer noch gedrückt. Auf diese Art und Weise können Sie statt eines leicht 38 Semikolons auf den Bildchirm zaubern.

Damit das nicht passiert, hat man den Tastatur-Strobe erfunden. Das höchste Bit an Adresse $C000 wird immer dann auf 1 gesetzt, wenn gerade eine Taste gedrückt wurde, und bleibt stehen, bis es vom 6502 explizit gelöscht wird. Das geschieht durch einen Zugriff auf Adresse $C010, den Strobe. $C010 ist ein Softwareschalter: ein Zugriff auf diese Adresse, egal ob Schreib- oder Lesebefehl, setzt den Strobe zurück.

Mit Hilfe dieser Konstruktion kann der 6502 unterscheiden, ob eine Taste neu gedrückt wurde oder nicht. Das kann dann so aussehen:

```
LOOP: Lade Akku mit Inhalt von $C000
      Bit 7 gesetzt?
      Nein? Gehe nach LOOP!
      Lösche Tastatur-Strobe
      Return
```

Durch das Löschen des Strobe können wir sicher gehen, das gleiche Zeichen nicht noch einmal einzulesen, es sei denn, die gleiche Taste wurde noch einmal betätigt.

PROG19 demonstriert die Anwendung der CMP-Instruktion. Zunächst wird ein Unterprogramm aufgerufen, das ein Zeichen von der Tastatur einliest. Es entspricht dem oben skizzierten Ablauf und enthält beim Rücksprung das gelesene Zeichen im Akku. Anschließend wird mit Hilfe von CMP festgestellt, ob es sich bei

dem gelesenen Zeichen um die ESC-Taste gehandelt hat. Wenn ja, endet das Programm, wenn nicht, wiederholen wir den ganzen Vorgang.

Da es im APPLE nur eine Tastaturadresse ($C000) gibt, die sowohl von VC (unserem Simulatorprogramm) als auch von PROG19 benutzt wird, müssen Sie darauf achten, die Taste genau dann zu drücken, wenn in PROG19 der entsprechende Mikroschritt erfolgt; ansonsten wird VC den Tastendruck als Aufforderung zum Pausieren verstehen.

Kapitel 12

Weitere nützliche Befehle

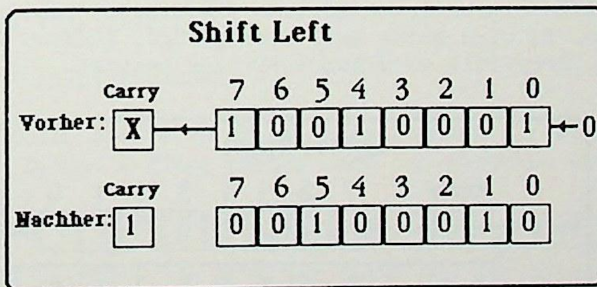
Bisher haben wir zwei Gruppen von Befehlen kennengelernt, die wirklich zupacken und sich die Hände schmutzig machen: ADC/SBC und die Inkrement/Dekrement-Befehle.

In diesem Kapitel werden wir die SHIFT- und die LOGICAL-Befehle vorstellen. Im Gegensatz zu den ersteren behandeln sie die Register nicht als ganzes, sondern verändern einzelne Bits.

SHIFT und ROTATE

Der 6502 hat besondere Befehle, um die Bits im Akkumulator um eine Position nach rechts oder links zu verschieben. Ähnlich wie ADC und SBC betrachten sie dabei das Carry-Bit als neuntes Bit des Akkus.

Der ASL-Befehl (Arithmetic Shift Left - Arithmetisches Verschieben nach links) schiebt alle Bits im Akkumulator oder in einer Speicherstelle um eine Stelle nach links. Bit 7 landet dabei im Carry-Bit (der alte Inhalt geht verloren), und rechts wird eine Null in Bit 0 gesetzt.

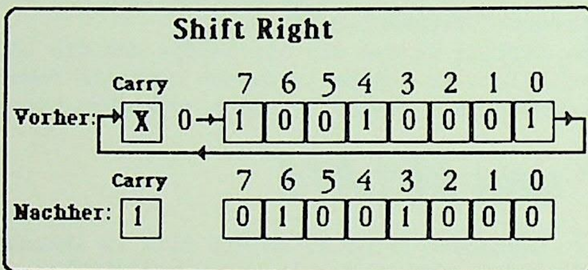


Interessant, aber wofür ist der Befehl gut? Zunächst gibt er uns die Möglichkeit, jedes einzelne Bit im Akku zu testen. Wollen wir z,B. Bit 5 untersuchen, dann benutzen wir drei aufeinanderfolgende ASL-Instruktionen, um Bit 5 ins Carry-Bit zu bringen. Mit einem BCC können wir dann testen und entsprechend verzweigen.

Ein Links-Shift hat darüberhinaus den überraschenden Nebeneffekt, den Akku-Inhalt mit 2 zu multiplizieren! Probieren Sie es aus:

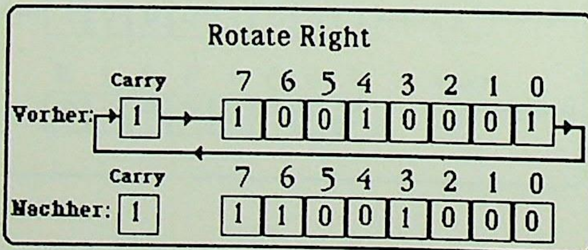
```
$20 (0010 0000) * 2 = $40 (0100 0000)
$37 (0011 0111) * 2 = $6E (0110 1110)
$64 (0110 0100) * 2 = $C8 (1100 1000)
```

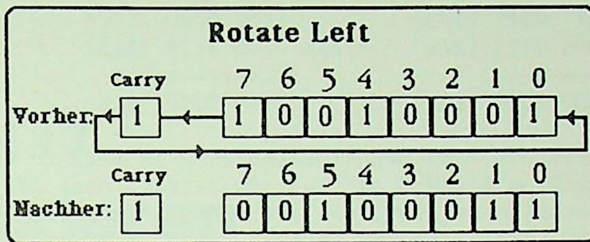
Um mit 4 zu multiplizieren, benutzen Sie zwei ASL-Instruktionen, drei ASL multiplizieren mit 8 usw.



LSR (Logical Shift Right - Logisches Verschieben nach rechts) ist das Gegenstück zu ASL. Diesmal geht Bit 0 ins Carry-Bit, und die Null wandert in Bit 7. Der Akkumulator wird dabei durch 2 dividiert. Wenn Sie mir das nicht glauben wollen, dann probieren Sie es mit dem VC-Kalkulator aus. Der Wert im Akku bildet den Quotienten, die Bits, die aus Position 0 herausfallen, bilden den Rest.

Die ROTATE-Instruktionen (Rotate - Rotieren) sind nur wenig verschieden. Es wird keine Null hinzugefügt, sondern der alte Inhalt des Carry-Bits wird zum Auffüllen benutzt.





Rotationen bewirken keine Multiplikation oder Division mit 2, es sei denn, das Carry-Bit wird immer rechtzeitig gelöscht.

(Eine geschichtliche Randbemerkung: ROR war die letzte Instruktion, die der 6502 verpaßt bekam, die ersten Prototypen des Mikroprozessors besaßen diesen Befehl noch nicht.)

Die anderen Register können nicht "geshiftet" oder rotiert werden, mit Speicherstellen dagegen ist es möglich.

Mnemonic	Absolut	Akku	Zero Page	Operation
ASL	$0E	$0A	$06	Links-Shift
LSR	$4E	$4A	$46	Rechts-Shift
ROL	$2E	$2A	$26	Links rotieren
ROR	$6E	$6A	$66	Rechts rotieren

PROG20 führt eine mehrstellige Verschiebung durch. Der Wert an $900,$901 (Lo-Byte zuerst!) wird mit 4 multipliziert, indem Sie eine ASL/ROR-Kombination zweimal ausführen: Das Lo-Byte wird nach links geshiftet, Bit 7 landet im Carry und wird mit Hilfe des ROL-Befehls ins Hi-Byte gepackt.

Auch die üblichen logischen Verknüpfungen gibt es in 6502-Assemblersprache:

AND (logisches UND)

ORA (Inklusiv-ODER)

EOR (Exklusiv-ODER)

Wie SBC und ADC operieren diese Instruktionen nur mit dem Akkumulator. Die Z- und N-Flags werden nach den gleichen Regeln verändert.

Auch hier sind die individuellen Bits wichtiger als das ganze Kollektiv. Die Instruktion AND 33 führt acht simultane AND-Operationen zwischen den Bits des Akkus und den Bits der Speicherstelle aus:

```
      0011 0011 ($33)           1100 0000 ($C0)
  AND 0100 0110 ($46)       AND 0100 1111 ($4F)
      ---------                 ---------
      0000 0010 ($02)           0100 0000 ($40)
```

Mit Hilfe von AND kann man einzelne Bits im Akku in die Knie zwingen. Um Bits 6 und 7 auf null zu setzen, ohne die anderen Bits zu beeinflussen, benutzen wir AND 3F. Um alle Bits außer Bit 2 zu löschen, probieren Sie AND 04.

Umgekehrt kann ORA verwendet werden, um bestimmte Bits auf 1 zu setzen. Mit ORA F0 werden die Bits 4 - 7 gesetzt, Bits 0 - 3 bleiben unverändert, mit ORA 88 werden Bits 3 und 7 gesetzt.

Mit EOR FF können alle Bits umgedreht werden: Aus Einsen werden Nullen und umgekehrt. Zwei aufeinanderfolgende EOR-Befehle mit dem gleichen Wert lassen den Akkumulator unverändert:

```
      0011 0111  ($37)
  EOR 1100 1000  ($C8)
      ---------
      1111 1111  ($FF)
  EOR 1100 1000  ($C8)
      ---------
      0011 0111  ($37)
```

Die BIT-Instruktion

Eine Mischung aus AND und CMP bietet die BIT-Instruktion (Bit-Test). Sie führt ein logisches UND zwischen Akku und Speicher durch, wie bei CMP bleibt der Akku jedoch unverändert, nur werden die Flags entsprechend gesetzt. Als Bonus wird Bit 6 in das V-Flag und Bit 7 in das N-Flag kopiert, sie können dort bequem abgefragt werden. Mit BIT können I/O-Statusadressen abgefragt werden, besonders wenn die Bits 6 und 7 interessieren.

Die logischen Instruktionen sind:

Mnemonic	Absolut	Direkt	Zero Page	Operation
AND	$2D	$29	$25	Logisches UND
ORA	$0D	$09	$05	Logisches ODER
EOR	$4D	$49	$45	Exklusiv-ODER
BIT	$2C	---	$24	Teste Bits

PROG21 zeigt, wie mit AND und ORA Bits im Akku verändert werden können. Das Unterprogramm an Adresse $A00 benutzt AND, um festzustellen, ob $900 und $901 beide $FF enthalten. Falls ja, enthält der Akku beim Rücksprung ebenfalls $FF, ansonsten $00.

Kapitel 13

Indexieren

Wir haben schon erwähnt, daß X und Y als Index-Register verwendet werden können. Es ist jetzt an der Zeit, zu erklären, was es damit auf sich hat. Bisher haben wir fünf Adressierungsarten kennengelernt. Zwei davon, Relativ und Implizit, sind Sonderfälle. Relative Adressierung wird nur bei den BRANCH-Befehlen benutzt, und implizite Instruktionen (TAX, CLC) besitzen gar keinen Adressteil.

Die anderen drei Modi, Direkt, Absolut und Zero Page, sind allgemeiner und erlauben es, die gleiche Instruktion in verschiedenen Situationen anzuwenden. Wir können wählen, wie wir den Akkumulator laden wollen: mit einer Zahl, die direkt auf den Opcode folgt (direkte Adressierung), oder mit dem Inhalt der angegebenen Adresse (Absolut oder Zero Page).

Vier weitere Möglichkeiten wollen wir jetzt einführen:

```
LDA $4000,X     Absolut,X
LDA $4000,Y     Absolut,Y
LDA $00,X       Zero Page,X
LDX $00,Y       Zero Page,Y
```

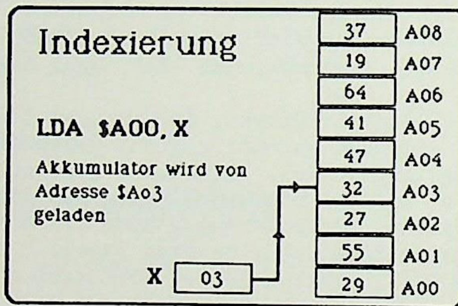
Indexierung kann am besten anhand eines Problems erläutert werden, das mit den bisher bekannten Adressierungsarten schwer zu lösen ist. Angenommen, wir wollen eine Gruppe von $10 Bytes von $900 nach $A00 verschieben, um Platz für etwas anderes zu machen. Mit unseren bisherigen Mitteln könnte das so aussehen:

```
LDA $0900
STA $0A00
LDA $0901
STA $0A01
LDA $0902
STA $0A02
LDA $0903
STA $0A03
etc......
```

Um 16 Bytes zu verschieben, brauchen wir 32 Instruktionen zu je drei Bytes. Nicht sehr effizient, 96 Programm-Bytes zu verwenden, um Platz für 16 Daten-Bytes zu machen. Was, wenn wir $100 oder $8000 Bytes verschieben wollen?

Wäre es nicht eleganter, diese Aufgabe in einer Schleife und unter Verwendung von ein wenig Inkrementierung zu lösen? Mit Indexierung, wobei X und Y als Offset zu einer Basisadresse benutzt werden?

Basis? Offset? Was zum Teufel ist das schon wieder? Bisher kannten wir nur eine Möglichkeit, den Akku mit dem Inhalt der Speicherzelle $0A03 zu laden: LDA $0A03 (Absolut). Bei der neuen Adressierungsart benutzen wir $0A00 als Basisadresse und lassen den 6502 den Inhalt des X-Registers hinzuaddieren. Haben wir vorher eine 3 ins X-Register gebracht, dann laden wir mit LDA $0A00,X ($BD $00 $0A) genau den Inhalt von $0A03. Inkrementieren wir anschließend das X-Register, dann laden wir mit dergleichen Instruktion von Adresse $0A04.



Indexierte Adressierung macht aus Speicherverschiebungen ein Kinderspiel. PROG22 demonstriert das an dem gleichen Problem:

```
$800 LDX #$00
$802 LDA $0900,X
     STA $0A00,X
     INX
     CPX #$10
     BNE $0802
     BRK
```

Aus 32 Instruktionen und 96 Bytes wurden 6 Instruktionen und 13 Bytes. Nicht schlecht! Und wir können mit diesem Programm bis zu 256 Bytes verschieben, ohne daß es deswegen auch nur ein um Byte länger wird.

Wenn Sie das Programm starten, dann achten Sie auf einen neuen Mikroschritt: CALC ADDRS (Adresse berechnen) ändert die Adresse auf dem Bus entsprechend dem X-Register. In allen anderen Aspekten (Ergebnis, Flags) verhält sich der neue Befehl genau wie LDA Absolut.

Arrays

Besonders nützlich ist die indexierte Adressierung bei der Benutzung von Arrays (Felder). Die Absolut,X-Adressierung erlaubt uns einen bequemen Zugriff auf Arrays mit bis zu 256 Elementen.

Stellen wir uns einmal vor, wir hätten ein komplettes Lohnabrechnungsprogramm in Maschinensprache geschrieben (vergessen Sie für einen Moment unsere Bemerkung aus Kapitel 1 über die Unsinnigkeit eines solchen Unternehmens). Für die korrekte Abrechnung braucht die Buchhaltung der BINÄR GmbH das genaue Alter jedes Mitarbeiters. So etwas speichert man am besten in einem Array:

Arrayelement (Personal-Nr.)	Inhalt (Alter)	
0	$40	Schwarz, Geschäftsführer
1	$16	Müller, Assistent
2	$20	Schulze, Buchhaltung
.	.	
.	.	
.	.	
255	$10	Schneider, Azubi

Solange die Firma nicht mehr als 256 Mitarbeiter beschäftigt und keiner älter als 255 ist, kann man das Alter eines Angestellten leicht ermitteln (die Liste ist ab Adresse $900 gespeichert):

```
LDX (Personal-Nr.)
LDA $0900,X
```

Die BINÄR GmbH ist laut Verordnung 218/4711 verpflichtet, der Oberfinanzdirektion das Gesamtalter der Belegschaft vierteljährlich zu melden. Nicht das Durchschnittsalter (das wäre Nummer 218/4712) sondern die Summe der Alter aller Mitarbeiter. COUNTPROG tut genau das, laden Sie es und sehen Sie es sich an:

```
START: LDX #$00
       STX $FB
       STX $FC
TOP:   LDA $0900,X
       CLC
       ADD $FB
       STA $FB
       BCC SKIP
       INC $FC
SKIP:  INX
       BNE TOP
       BRK
```

COUNTPROG durchläuft alle 256 Elemente in der Tabelle und addiert die einzelnen Werte an der Adresse $FB,$FC. Das X-Register dient dabei als Schleifenzähler und als Index-Register.

Zunächst werden das X-Register und die beiden Speicherzellen, die am Ende die Summe enthalten sollen, auf null gesetzt. Dann beginnen wir eine Schleife, in der das Alter von Mitarbeiter Nummer X in den Akku geladen wird und dieser (8-Bit-)Wert anschließend zu dem 16-Bit-Wert in $FB,$FC addiert wird. Schauen wir und dieses Aufsummieren noch etwas genauer an: Zunächst wird das Carry-Bit gelöscht, sonst geraten uns leicht ein paar Jahre zuviel in die Rechnung. Dann addieren wir das Alter von Mitarbeiter X zu dem Lo-Byte der Summe. Produziert diese Addition ein Carry (z.B.: Summe = $3F2 und Alter = $21), dann muß das Hi-Byte der Summe um eins inkrementiert werden. Nach 256 Durchläufen gestattet uns das BNE, zum BRK-Befehl weiterzulaufen und das Programm zu beenden.

Starten Sie COUNTPROG. Das Programm braucht eine Weile, hören Sie sich in der Zwischenzeit eine gute Schallplatte an, oder waschen Sie Ihren Wagen. Wenn es fertig ist, können Sie sich das Durchschnittsalter der BINÄR GmbH auch mit dem Taschenrechner ausrechnen.

Bei den meisten indexierten Befehlen kann statt des X- auch das Y-Register verwendet werden. Die folgende Tabelle gibt eine Übersicht der LOAD- und STORE-Befehle:

Befehl	ABS,X	ABS,Y
LDA	$BD	$B9
STA	$9D	$99
LDX		$BE
STX		
LDY	$BC	
STY		

Zum erstenmal haben wir Löcher in unserer Tabelle. Es gibt kein "STX ABS,Y" und auch kein "LDY ABS,Y". Die Lücken sind hauptsächlich durch den Stand der Technik im Jahre 1975 bedingt. So gerne wir die neuen Adressierungsarten für alle Befehle hätten - es war einfach nicht genug Platz auf dem Chip.

Die wichtigsten Instruktionen (ADC, SBC, EOR, AND, ORA, CMP, LDA, STA) haben die meisten Adressierungsarten zur Auswahl bekommen. Im Anhang finden Sie eine Zusammenstellung aller zulässigen Instruktionen.

Indexierung auf der Zero Page

Sie sind unverzichtbar, die neuen Adressierungsarten, aber mit jeweils drei Bytes Länge nicht gerade platzsparend. Es gibt sie deshalb auch in einer Zwei-Byte-Version für die Zero Page:

Befehl	ZP,X	ZP,Y
LDA	$B5	$B9
STA	$95	$99
LDX		$B6
STX		$96
LDY	$B4	
STY	$94	

PROG23 zeigt die indexierte Zero-Page-Adressierung. Beachten Sie den "Umklappeffekt": Wenn während des Mikroschrittes CALC ADDRSS die Addition von X und AD einen Wert größer $00FF liefert, dann wird der Überlauf abgeschnitten, so daß auf jeden Fall wieder eine Zero-Page-Adresse herauskommt. Falls das X-Register den Wert $90 enthält, dann lädt der Befehl LDA $80,X den Akkumulator mit dem Inhalt der Adresse $10.

Indexierung, zweiter Akt

In diesem Abschnitt werden wir zwei weitere Adressierungsarten einführen: die indirekte indexierte und die indexierte indirekte Adressierung.

Erinnern Sie sich noch an das Konzept der indirekten Adressierung? Wir haben es weiter vorne bei dem Befehl JMP Indirekt kennengelernt. Ein ganz gewöhnliches JMP $B136 springt an die Adresse $B136, ein JMP ($B136) dagegen holt zunächst den Inhalt der Adressen $B136 und $B137 und springt dann an die dadurch definierte Adresse. Das erlaubt uns, die Sprungadresse unter Programmkontrolle zu verändern.

Der 6502 hat zwei Adressierungsarten, die das Indirekt-Konzept mit der Indexierung kombinieren: die indirekte indexierte, die nur mit dem Y-Register funktioniert, und die indexierte indirekte für das X-Register. Zugegeben, die Namen sind ein klein wenig verwirrend.

Die indirekte indexierte Adressierung

Stellen Sie sich vor, daß Sie einen Block von mehr als 256 Bytes im Speicher verschieben wollen. Sie können unsere alte Verschiebe-Routine zweimal anwenden, um einen Block von $200 Bytes von $3000 nach $4000 zu verschieben:

```
         LDX #$00
LOOP1:   LDA $3000,X
         STA $4000,X
         INX
         BNE LOOP1
LOOP2:   LDA $3100,X
         STA $4100,X
         INX
         BNE LOOP2
         BRK
```

Obwohl das Programm gut funktioniert, wird schnell klar, daß es zum Verschieben größerer Speicherblöcke nicht geeignet ist. Damit kommen wir zur indirekten indexierten Adressierung:

```
LDA ($45),Y
```

Die Anordnung der Klammern gibt uns eine kleinen Hinweis auf die Reihenfolge der Bearbeitung. Da Y sich außerhalb der Klammern befindet, wird zuerst der indirekte Teil der Adresse berechnet und anschließend mit Y indexiert.

Zum Beispiel werden bei LDA ($45),Y Speicheradresse $0045 und $0046 gelesen. Angenommen, $0045 enthält $00, und $0046 enthält $20, dann bilden wir daraus "indirekt" die Adresse $2000 (wie immer, Lo-Byte zuerst). Anschließend folgt die Indexierung. Wenn Y z.B. 6 enthält, dann laden wir schließlich und endlich den Akkumulator mit dem Inhalt von $2006. Wenn wir Speicherzelle $46 um eins inkrementieren, dann greifen wir mit dem nächsten LDA ($45),Y auf Adresse $2106 zu.

Obwohl die indirekte indexierte Adressierung wegen der vielen Speicherzugriffe langsamer arbeitet als andere Adressierungsarten, ist sie jedoch hinsichtlich der Befehlslänge nicht zu schlagen. IND (Y) benötigt nur zwei Bytes, nämlich den Instruktionscode und die Adresse des ersten von zwei Zero-Page-Bytes, die die Basisadresse enthalten.

Die indirekte indexierte Adressierung ist jedoch hauptsächlich verantwortlich für das Gedrängel auf der Zero Page. Jedes Programm braucht auf der Zero Page Platz für ein paar Pointer (Zeiger, - so genannt, weil sie auf eine beliebige Adresse im Speicher "zeigen").

Für diese Adressierungsart darf nur das Y-Register benutzt werden, es gibt kein LDA ($45),X. Laden Sie CLEARPROG von der Diskette und schauen Sie es sich an. Es schreibt Nullen in alle $2000 Speicherzellen, die zusammen den Hi-Res-Schirm bilden, und löscht ihn damit vollständig.

```
0800-  20 04 08    JSR $0804
0803-  00          BRK
0804-  A9 00       LDA #$00
0806-  A8          TAY
```

```
0807-  85 FA       STA $FA
0809-  A9 20       LDA #$20
080B-  85 FD       STA $FB
080D-  A9 00       LDA #$00
080F-  91 FA       STA ($FA),Y
0811-  C8          INY
0812-  D0 F9       BNE $080D
0814-  E6 FB       INC $FB
0816-  A5 FB       LDA $FB
0818-  C9 40       CMP #$40
081A-  D0 F1       BNE $080D
081C-  60          RTS
081D-  00          BRK
```

Zunächst wird ein Zeiger ($FA,$FB) auf den Anfang der Hi-Res-Seite gerichtet ($2000, das sind die 7 Bits in der oberen linken Ecke des Bildschirmes). In der folgenden Schleife werden die ersten 256 Bytes gelöscht, die entscheidende Instruktion ist STA ($FA),Y. Anschließend wird der Hi-Byte-Teil des Zeigers inkrementiert und mit $40 verglichen. Ist dieser Wert erreicht, dann sind wir fertig; ist das nicht der Fall, werden die nächsten 256 Bytes gelöscht.

Lassen Sie von diesem Programm nicht mehr als ein paar Probeschritte ablaufen. Wenn Sie im Simulations-Modus warten wollen, bis der Bildschirm vollständig gelöscht worden ist, dann könnten Sie leicht graue Haare dabei bekommen. Es wird Zeit für uns, ein paar neue Monitor-Kommandos aus dem Hut zu zaubern.

Der Master-Modus

Bisher haben wir uns ausschließlich im Non-Master-Modus bewegt. Für Anfänger ist er am sichersten; es ist fast unmöglich, den Computer darin zum Absturz zu bringen. Zugriffe auf Speicherbereiche, die das VC-Programm enthalten, sind ebenso verboten wie bestimmte I/O-Befehle, die gefährlich werden könnten. Vor allem aber ist das GO-Kommando blockiert.

GO

Der GO-Befehl ermöglicht es, das Programm im Speicher durch den 6502 selbst (statt durch den Simulator) ausführen zu lassen. Wenn der Master-Modus eingeschaltet ist und die erste Instruktion des Programms ist ein JSR-Befehl, dann übergibt VC die Ausführung des Programmes direkt an den 6502. Durch einen RTS-Befehl am Ende des Programms wird die Kontrolle wieder an VC zurückgegeben. Der Bildschirm zeigt anschließend die Registerinhalte, wie sie am Ende des Programms gültig sind.

Es gibt eine Million Möglichkeiten (vorsichtige Schätzung), in einem Assemblerprogramm Fehler zu machen, und die meisten bewirken, daß Sie die Kontrolle über den Computer verlieren. Ihr Computer kann dann "abstürzen", "sich aufhängen", "in die Wüste laufen" oder ähnlich schreckliche Dinge tun. Wenn Sie den 6502 in eine Endlosschleife schicken (z.B. $2000: 4C 00 20 JMP $2000), dann hilft nur noch die RESET-Taste. In schlimmeren Fällen (etwa wenn Sie VISIBLE COMPUTER überschreiben) müssen Sie das Gerät aus- und wieder einschalten.

Glücklicherweise gibt es CLEARPROG; bevor Sie es starten, schalten Sie den Master-Modus ein:

```
        MASTER ON
```

Ein "M" erscheint auf der Statuszeile. Endlich sind Sie ein Visible-Computer-Meister (gedämpfter Applaus)!

Im Master-Modus können sie nicht länger auf die VC-Diskette zugreifen (wenn Sie es dennoch probieren, bekommen Sie einen I/O-ERROR). Dafür können Sie jetzt normale DOS 3.3-Disketten benutzen; das werden Sie in Zukunft auch reichlich tun.

Wenn Sie ein Programm von der VC-Diskette laden wollen, dann müssen Sie den Master-Modus vorher abschalten: MASTER OFF. Gehen wir jetzt zurück zu CLEARPROG. Stellen Sie den Programmzähler auf $800 (Master-Modus arbeitet nur, wenn die erste Instruktion ein JSR ist), und GO.

Hat nicht lange gedauert, oder? Stellen Sie mit RESTORE den alten Bildschirminhalt wieder her und ändern Sie den Wert an Adresse $080E. $FF macht den ganzen Bildschirm weiß, $55 bewirkt ein interessantes Streifenmuster.

REVERSEPROG ist eine Variante von CLEARPROG. Mit Hilfe von EOR FF wird jedes Byte auf der Hi-Res-Seite umgedreht, so daß Sie eine Schwarz-auf-Weiß-Darstellung des VC erhalten. Das Programm läuft in einer Endlosschleife, bis es durch einen beliebigen Tastendruck gestoppt wird.

```
        GO
```

Funktioniert nicht? Kann auch nicht, denn es fehlt die JSR-Instruktion am Anfang des Programms. Schreiben Sie mit Hilfe von EDIT folgenden Befehl an Adresse $0300:

```
        0300: 20 00 08 JSR $0800
```

Setzen Sie dann den PC auf $0300, und los geht's! Wenn Sie eine Taste drücken, um das Programm anzuhalten, dann haben Sie eine 50:50 Chance, daß der Bildschirm in der Negativ-Darstellung stehen bleibt. Macht aber nichts, mit RESTORE kehren Sie zu der gewohnten Darstellung zurück.

Die indexierte indirekte Adressierung

Wenn Sie die indirekte indexierte Adressierung mögen, dann werden Sie auch die indexierte indirekte lieben. Während erstere nur mit dem Y-Register möglich war, funktioniert diese nur mit dem X-Register:

```
LDA ($45,X)
```

Auch hier kann man nach einem scharfen Blick die Vorgehensweise erahnen. Das X-Register wird benutzt, um einen von mehreren Zeigern zu indexieren, der dann auf die richtige Adresse im Speicher deutet. Angenommen, die Adressen $10 - $17 haben folgenden Inhalt:

```
$10: $00        $11: $C0
$12: $10        $13: $C1
$14: $00        $15: $C2
$16: $00        $17: $C3
```

Acht Speicherplätze stellen vier Zeiger dar: Der erste zeigt auf $C000, der zweite auf $C110, der dritte auf $C200 und der vierte auf $C300. Indexierte indirekte Adressierung wird benutzt, um einen dieser Zeiger auszuwählen. LDA ($10,X) lädt den Akkumulator aus Adresse $C000, falls X = 0, aus Adresse $C110, falls X = 2, aus $C200, falls X = 4, aus $C300, falls X = 6. Die Zeiger brauchen übrigens nicht unbedingt auf einer geraden Adresse zu stehen. Falls X = 3, dann wird der Akku mit dem Inhalt von $00C1 geladen.

Die indexierte indirekte Adressierung wird hauptsächlich genutzt, um unter Programmkontrolle unter mehreren Tabellen auszuwählen. Es wird in der Praxis nicht so häufig gebraucht wie (IND),Y, aber es wird der Tag kommen, an dem Sie froh sind, daß es (IND,X) gibt.

Neue Probleme bei der BINÄR GmbH

Aufgrund der schlechten Ertragslage hat die Geschäftsführung beschlossen, das Urlaubsgeld für alle Betriebsangehörigen unter 48 zu streichen. (Zufällig ist das jüngste Mitglied der Geschäftsleitung gerade 48). Die Programmierer sehen sich vor folgende Aufgabe gestellt:

Aus der Liste der Mitarbeiter (ab Adresse $0900) soll eine neue Liste gewonnen werden (ab $0A00). Erfüllt der Mitarbeiter die Bedingung (ist er also 47 oder jünger), so wird sein Alter in

die neue Tabelle eingetragen, wenn nicht, dann wird der Eintrag 0 vorgenommen. Außerdem interessiert die Zahl der Mitarbeiter, die weiterhin Urlaubsgeld erhalten werden.

```
        BTABLEPROG

        START:  LDA #$09        ; zwei Zeiger:
                LDY #$0A        ; $FB,$FC zeigt auf
                LDX #$00        ; $0900 (alte Tabelle)
                STX $FB         ; $FD,$FE zeigt auf
                STX $FD         ; neue Liste ($0A00)
                STA $FC
                STA $FE         ; Y zählt die Leute über 47
                LDY #$00        ; auf 0 initialisieren
        LOOP:   LDA ($FB,X)     ; Lies von alter Liste (X=0)
                CMP #$30        ; $30 = 48 dezimal
                BCC YOUNG       ; Carry Clear bedeutet
                LDA #$00        ; jünger als 48
                INY             ; Zähler inkrementieren
        YOUNG:  LDX #$02        ; in der neuen Tabelle
                STA ($FB,X)     ; ablegen (X=2)
                LDX #$00        ; X löschen und
                INC $FB         ; Zeiger inkrementieren
                INC $FD         ; für nächsten
                BNE LOOP        ; Durchgang
                RTS
```

Die entscheidende Stelle in BTABLEPROG ist CMP #$30. Wenn der Wert im Akku kleiner als 47 ist, dann wird das Carry-Bit gelöscht und der folgende BRANCH-Befehl überspringt das Löschen des Akkus. Im unteren Teil wird das Alter in die zweite Tabelle eingetragen (oder eine Null, wenn der Mitarbeiter älter als 47 ist und deshalb von der neuen Regelung nicht betroffen ist).

Laden Sie BTABLEPROG und studieren Sie die wichtigsten Schritte im Simulator. Bevor Sie es mit GO laufen lassen können, müssen Sie irgendwo ein JSR $800 unterbringen (z.B. an Adresse $300; vielleicht steht sogar noch eins vom vorigen Programm da).

Und jetzt: GO. Schauen Sie ins Y-Register, wieviele Mitarbeiter der BINÄR GmbH älter als 47 sind. (Antwort: 12. Elf Mitglieder der Geschäftsführung und der Schwager vom Chef.)

Nur die wichtigsten der 6502-Instruktionen haben diese indexierten/indirekten Adressierungsarten: ADC, AND, EOR, SBC, ORA, CMP und natürlich STA und LDA.

Kapitel 14

Verschiedenes

In den vorangegangenen Kapiteln haben wir öfters Behauptungen aufgestellt wie: "Diese Instruktion ist sehr wichtig", oder: "Dieser Befehl wird häufig benutzt". In diesem Kapitel werden wir uns um Instruktionen kümmern, die nicht so sehr im Mittelpunkt des Interesses stehen.

NOP (No Operation - Tue nichts), Opcode $EA, tut genau das, was sein Name aussagt, nämlich nichts! Wenn der 6502 auf einen NOP trifft, dann wird nur der Programmzähler um eins hochgezählt und ein bißchen Zeit dabei verbraucht. Wofür kann das gut sein? Es gibt zwei Situationen, in denen NOP verwendet wird: in einer sorgfältig ausgerechneten Warteschleife und, häufiger noch, als Füllstoff für Programmlücken, wie sie beim Testen von Programmen entstehen.

```
0800: 20 00 10 JSR $1000
0803: 20 00 20 JSR $2000
0806: 20 00 30 JSR $3000
```

Wenn Sie beim Testen dieses Programms entdecken, daß die zweite Unterroutine falsch ist, dann können Sie trotzdem die dritte ausprobieren, indem Sie den mittleren Unterprogrammaufruf durch drei NOP-Befehle überschreiben:

```
0800: 20 00 10 JSR $1000
0803: EA       NOP
0804: EA       NOP
0805: EA       NOP
0806: 20 00 30 JSR $3000
```

An dieser Stelle eine Warnung: In manchen Handbüchern (z.B. im APPLE II-Benutzer-Handbuch) steht, daß alle 105 undefinierten Opcodes als NOP benutzt werden können. Das ist falsch! Nur $EA ist ein NOP; die anderen Instruktionen können alle möglichen Nebeneffekte haben, die nirgendwo dokumentiert sind.

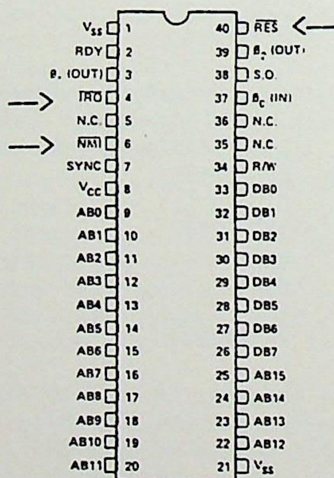
Die großen Verdienste, die NOP leistet, wenn es ums Zeittotschlagen geht, bringen uns zum Thema Ausführungszeiten. Meistens ist es nur wichtig, daß ein Programm schnell ist, oder wenigstens schnell genug für eine bestimmte Aufgabe. In Unterprogrammen zur Tonerzeugung ist es dagegen wichtig, genau zu wissen, wieviel Zeit eine Instruktion verbraucht. Die kleinste Zeiteinheit auf dem APPLE beträgt 0,977 Mikrosekunden, das ist die Dauer von einem Instruktionszyklus.

Alle Instruktionen brauchen zu ihrer Ausführung mindestens zwei Zyklen. DEX und SEC sind sehr schnell, sie kommen mit zwei Zyklen aus, LDA $45 braucht drei Zyklen, RTS und LDA ($45),X brauchen jeweils sechs. Im allgemeinen ist eine Instruktion um so schneller, je weniger Speicherzugriffe sie zu ihrer Ausführung benötigt.

RTI

RTI (Return from Interrupt - Rückkehr von einer Unterbrechung) ist eine Instruktion, die Sie vielleicht nie benutzen werden. Trotzdem sollten wir die Frage klären: "Was ist überhaupt ein Interrupt (Unterbrechung)?"

Drei der vierzig Beinchen des 6502-Mikroprozessors erlauben es, den Prozessor in seinem normalen FETCH/EXECUTE-Rhythmus zu stören: RESET, NMI (Non-Maskable Interrupt - Nicht-maskierbarer Interrupt) und IRQ (Interrupt Request - Interrupt-Anforderung). Alle drei veranlassen den Prozessor (sehr höflich oder auch sehr ruppig), seine bisherige Tätigkeit aufzugeben und etwas anderes zu tun.



RESET

Es gibt zwei Möglichkeiten, auf dem APPLE II einen RESET durchzuführen: durch Einschalten des Geräts oder durch Drücken der RESET-Taste (außer auf den ganz alten APPLES muß dazu immer auch die CONTROL-Taste betätigt werden). In beiden Fällen wird ein indirekter Sprung nach $FFFC (also nach der Adresse, die in $FFFC,$FFFD gespeichert ist) durchgeführt. Auf den meisten APPLES beginnt dieses Programm an Adresse $FA62.

Die RESET-Behandlung ist im Monitor-Listing (siehe die einschlägigen Handbücher) nachzulesen: Zunächst ertönt der bekannte Piepston, der Bildschirm wird auf Textdarstellung geschaltet, und dann stellt sich die große Frage: "Bin ich gerade eingeschaltet worden, oder war es die RESET-Taste?" Die Antwort hierauf steht in zwei Bytes im Speicher. Sehen diese nicht in Ordnung aus, dann nimmt der APPLE an, daß er gerade eingeschaltet worden ist. Er löscht dann den Bildschirm, schreibt "APPLE II" in die erste Zeile, sucht nach der Disk-Interface-Karte und versucht, eine Diskette zu "booten".

Wenn die beiden Testbytes in Ordnung scheinen, dann kehrt APPLE in die Umgebung zurück, die beim RESET gerade aktiv war. Wenn vorher ein BASIC-Programm lief, dann kehren Sie zu BASIC zurück; Ihr Programm ist unversehrt, läuft aber nicht.

Die RESET-Behandlung wird noch ausführlicher in den 'Reference'-Handbüchern erklärt.

IRQ

Wenn das entsprechende Signal am IRQ-Pin des Prozessors gegeben wird, dann unterbricht er die Bearbeitung des laufenden Programms, merkt sich aber erst die Stelle, so daß er später dort weitermachen kann. Der Programmzähler und das Statusregister werden auf den Stack geschrieben, dann wird ein indirekter Sprung nach $FFFE ausgeführt.

Obwohl der APPLE II (zumindest in der Grundausstattung) völlig ohne Interrupts auskommt, spielen sie jedoch in anderen Mikrocomputern eine wichtige Rolle. Ein Beispiel dafür bietet die Software-Uhr. In vielen Systemen ist mit dem IRQ-Pin ein Timer verbunden, der ihm regelmäßig Impulse (z.B. 10 pro Sekunde) liefert.

Die Interrupt-Routine stellt zunächst fest, daß es sich um eine externe Unterbrechung handelt und nicht um einen BRK-Befehl; anschließend zählt sie eine Speicherstelle hoch. Das könnte dann so aussehen (die Adressen $00 und $01 sollen den 16-Bit-Zähler enthalten):

```
            INC $00
            BNE SKIP
            INC $01
    SKIP:   RTI
```

Diese einfache Uhr zählt zwar nur, wieviele Zehntelsekunden vergangen sind, seit der Zähler zuletzt auf null gesetzt wurde, aber das ist auf jeden Fall besser, als gar keine Uhr zu haben. Die Ausführungszeiten von Benchmark-Programmen kann man damit ganz gut bestimmen.

Solche Interrupt-Routinen werden oft auch als BACKGROUND-(Hintergrund-)Programme bezeichnet. Der 6502 unterbricht die Arbeit am FOREGROUND-(Vordergrund-)Programm zehnmal pro Sekunde, kehrt aber anschließend immer wieder brav zurück. Da das Hintergrund-Programm wenig Zeit beansprucht, sieht es so aus, als ob das Hauptprogramm den 6502 für sich alleine hätte.

Interrupt-Programme müssen sehr sorgfältig mit den Registern umgehen. Können Sie sich das Vergnügen vorstellen, ein Hauptprogramm zu testen, wenn das X-Register zehnmal pro Sekunde auf mysteriöse Weise auf null gesetzt wird? Ein IRQ rettet nur den PC und das P-Register automatisch. Wenn das Interrupt-Programm auch die anderen Register benutzen will, dann müssen diese zunächst an einen sicheren Platz gerettet werden und vor der Rückkehr ins Vordergrund-Programm wiederhergestellt werden. Üblicherweise benutzt man den Stack für diese Zwischenablage.

Mit dem I-Bit im P-Register können Interrupts "maskiert" werden. Wenn das Bit gesetzt ist, wird der IRQ-Pin völlig ignoriert. Das ist sinnvoll, wenn Sie sich beispielsweise in einer genau berechneten Zeitschleife befinden, in der Sie nicht gestört werden wollen, oder aber, wenn Sie bereits einen Interrupt bearbeiten.

Die ganze Wahrheit über BRK

Haben Sie sich schon einmal gefragt, wofür das B-Flag eigentlich gut ist, oder warum man BRK manchmal auch einen "Software-Interrupt" nennt?

BRK löst beim 6502 genau die gleiche Reaktion aus wie ein IRQ-Signal. Es wird ein indirekter Sprung an Adresse $FFFE ausgeführt und nachgesehen, ob der Interrupt von der Hardware oder von der Software stammt. Wenn das B-Flag gesetzt ist, dann war ein BRK-Befehl der Auslöser. Wenn der Interrupt durch den IRQ-Pin veranlaßt wurde (unwahrscheinlich, denn die normale APPLE-Hardware tut sowas nicht), dann lädt die Interrupt-Routine den PC und das P-Register vom Stack und setzt das zuvor laufende Programm fort.

BRK ist nützlich beim Testen von Programmen. Ein BRK an einer strategisch wichtigen Stelle in Ihrem Programm hilft Ihnen herauszufinden, welches Unterprogramm funktioniert und welches noch nicht.

VISIBLE COMPUTER behandelt den BRK-Befehl ein wenig anders: Das laufende Programm wird einfach angehalten. Wenn der 6502 auf ein BRK stößt, während ein Programm mithilfe des GO-Kommandos ausgeführt wird, dann wird die Kontrolle an VC zurückgegeben. Der Bildschirm zeigt die Registerinhalte zum Zeitpunkt des BRK, der PC enthält die Adresse des BRK plus 2.

NMI

NMI ist ähnlich wie IRQ. Er benutzt jedoch einen anderen Vektor ($FFFA) und kann auch nicht ignoriert werden. Der NMI-Pin könnte beispielsweise mit einem Spannungssensor gekoppelt sein: Wenn die Betriebsspannung unter einen bestimmten Wert fällt, dann wird ein Interrupt ausgelöst. Die Interrupt-Routine könnte dann eine Notstromversorgung einschalten oder schnell alle wichtigen Files abschließen.

Negative Zahlen

Alle Zahlen, die wir bisher addiert, subtrahiert oder sonstwie behandelt haben, waren positiv. Doch wie halten wir es (nicht nur auf dem Bankkonto) mit den negativen Zahlen? Oder, anders gefragt, was zeigt der Akkumulator, wenn wir 6 von 3 subtrahieren? Auf jeden Fall kein Minuszeichen, soviel ist schon sicher.

Irgendwelche Vorschläge, wie man negative Zahlen darstellen könnte? Ich will Ihnen einen Hinweis geben: es hat etwas mit Bit 7 zu tun. Wie wäre es mit folgender Methode? Wir benutzen nur die niedrigsten sieben Bits für den absoluten Wert der Zahl und Bit 7 als Vorzeichen:

```
0011 1111 = + $3F
1011 1111 = - $3F
```

und

```
0100 1010 = + $4A
1100 1010 = - $4A
```

Nicht schlecht, oder? Fast so, wie wir Menschen es tun: Wenn kein Minuszeichen davorsteht, ist die Zahl positiv, ansonsten negativ.

Wenn wir das siebte Bit fürs Vorzeichen benutzen, dann können wir in einem Byte die Zahlen von -127 bis +127 darstellen, in zwei Bytes die Werte -32767 bis +32767 (erinnert das jemanden an die INTEGER-Variablen in BASIC?).

Doch Stop! Obwohl unsere Idee einer glasklaren Logik folgt, hat sie einen entscheidenden Nachteil: Sie ist nicht praktikabel!

3 + (-6) sollte -3 produzieren. Tut es das?

```
  0000 0011      (3)
+ 1000 0110     (-6)
  ---------
  1000 1001     (-9)
```

Offensichtlich nicht. Wie wär's mit 26 + (-14)?

```
  0001 1010      (1A)
+ 1000 1110     (-0E)
  ---------
  1010 1000     (-28)
```

Das ist nicht einmal "ungefähr richtig".

Statt Ihnen jetzt eine ganze Latte logischer, aber nicht funktionierender Lösungen vorzuführen, will ich Ihnen doch lieber gleich den richtigen Weg zeigen, das Zweierkomplement.

Auch hier wird Bit 7 benutzt, um eine negative Zahl zu bezeichnen, aber die restlichen 7 Bits werden anders dargestellt. Das Zweierkomplement wird gebildet, indem man alle Bits umdreht und am Schluß eins addiert. -$19 wird durch das Zweierkomplement von +$19 dargestellt:

```
 $19 = 0001 1001
-$19 = 1110 0110 + 1 = 1110 0111

 $64 = 0110 1000
-$64 = 1001 0111 + 1 = 1001 1000
```

Das ist nicht ganz so einfach wie unsere erste Methode, aber mit dem Zweierkomplement funktioniert es dafür. Addieren wir 3 und (-6):

```
-6 = Zweierkomplement von +6
   = Zweierkomplement von 0000 0110
   = 1111 1001 + 1
   = 1111 1010
```

Also:

```
  0000 0011      (3)
+ 1111 1010     (-6)
  ---------
  1111 1101      (?)
```

Da das Ergebnis negativ ist (Bit 7 ist gesetzt), müssen wir wieder das Zweierkomplement bilden, um zu sehen, ob die Addition korrekt durchgeführt worden ist.

```
Zweierkomplement von 1111 1101 = 0000 0010 + 1
                               = 0000 0011
                               = 3
```

Es hat funktioniert, unser Ergebnis ist -3!

Der Zweck dieses Buches ist es, Sie in die Assemblerprogrammierung einzuführen, und nicht, Sie auf die Promotion zum Dr.bin. vorzubereiten. (Deutlich hörbare Seufzer der Entäuschung.) Wir werden uns deshalb den ausführlichen Beweis schenken, warum das Verfahren immer funktioniert.

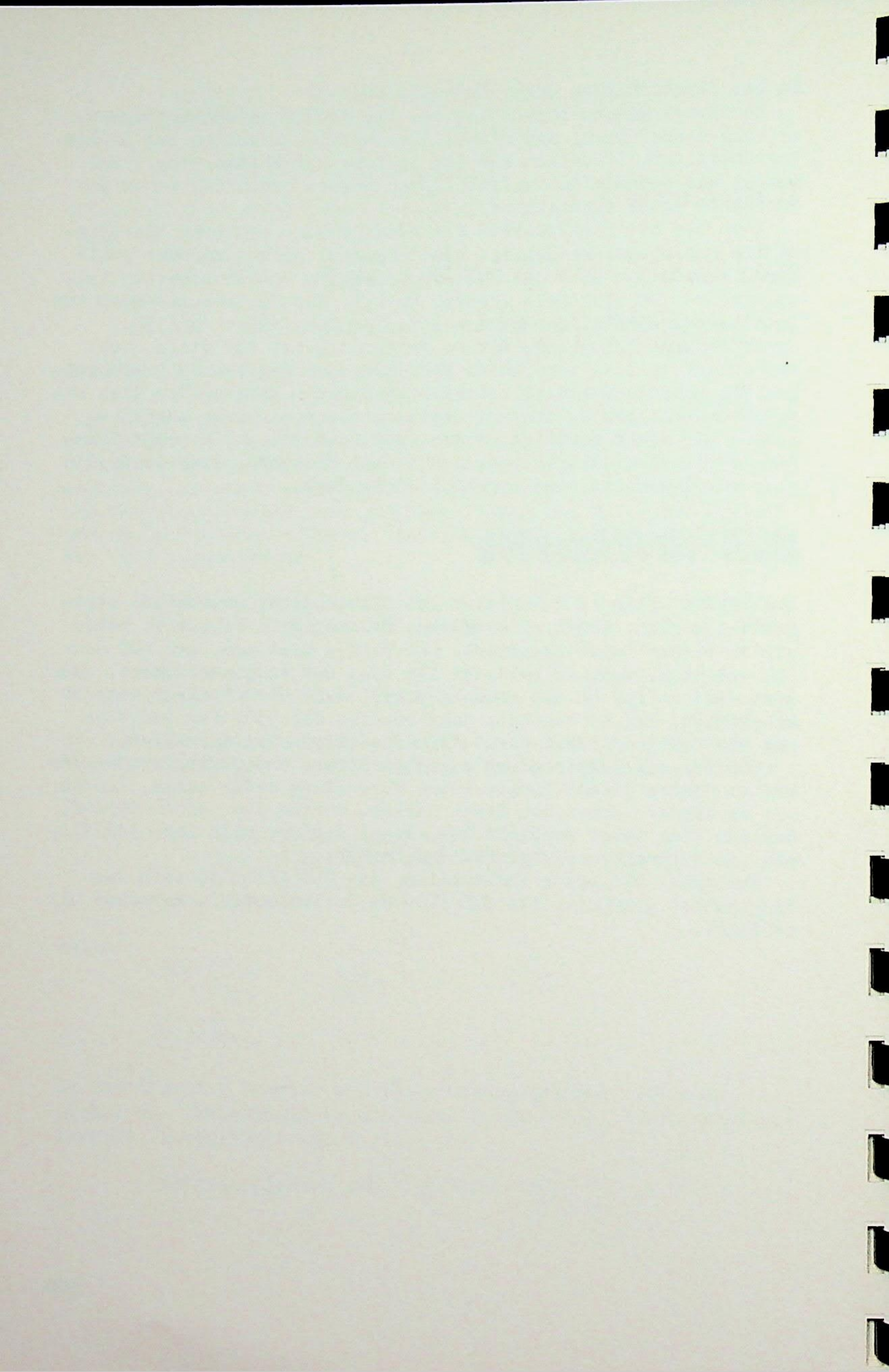
Üben Sie mit PRACTICEPROG die Additon von negativen Ein-Byte-Zahlen und positiven Zahlen. Wenn Sie faul sind (und/oder pfiffig), können Sie sich die Zweierkomplemente vom eingebauten Kalkulator des VC ermitteln lassen. Um z.B. das Zweierkomplement von $3411 auszurechnen, subtrahieren Sie einfach +$3411 von $0.
Abschließende Bemerkung: Binäre Arithmetik ist für viele nicht auf Anhieb voll zu verstehen. Wenn die vorangegangenen Ausführungen Sie mehr verwirrt als erleuchtet haben - trösten Sie sich mit dem Gedanken, daß in den meisten Assemblerprogrammen negative Zahlen gar nicht benötigt werden. Und wenn Sie es in sechs Monaten oder in fünf Jahren wirklich einmal brauchen, dann lesen Sie sich den Abschnitt noch einmal in Ruhe durch.

BCD-Arithmetik

Das Dezimal-Flag im P-Register ist bisher nicht sonderlich strapaziert worden. Außer einer kurzen Erwähnung in Kapitel 7 haben wir es bisher total ignoriert. Das D-Flag bestimmt, wie ADC und SBC arbeiten. Wenn es gelöscht ist (das war es bisher immer, oder zumindest sollte es das gewesen sein), dann wird Binärarithmetik ausgeführt. Ist es gesetzt, dann benutzt der 6502 für Addition und Subtraktion "BCD-Zahlen" (Binär codierte Dezimalzahlen).

BCD ist ein Zwischending zwischen Binär- und Dezimalsystem. Da wir im Rahmen dieses Kurses keine Verwendung dafür haben, lassen wir es einfach unter den Tisch fallen. Achten Sie jedoch darauf, daß das Flag immer gelöscht ist, sonst liefern alle Ihre Additionen und Subtraktionen falsche Ergebnisse.

Übrigens: Die erste Instruktion, die der APPLE II nach dem Einschalten ausführt, ist CLD (Lösche Dezimalmodus). Belassen wir es dabei.



Kapitel 15

Unser erstes Programm

Bisher haben wir die Beispielprogramme benutzt, ohne lange danach zu fragen, wie sie entstanden sind. Sie tippten einfach BLOAD, und schon waren sie da. Da Sie am Ende dieses Kurses in der Lage sein sollen, selbstständig ein Assemblerprogramm zu schreiben, werden wir das jetzt zusammen üben. Von der ersten vagen Idee zum fertigen 6502-Programm.

Hmmmmmmmm. Was könnten wir als Idee benutzen... Nein, das haben wir schon mal gemacht. Zu kompliziert. Wie wär's mit.... Nein, viel zu einfach. Ich hab's: Musik machen auf der APPLE-Tastatur. Arbeitstitel: "ASCII ORGAN" ("die ASCII-Orgel").

Je höher der ASCII-Wert der gedrückten Taste, desto tiefer der Ton. Die Dauer des Tons soll konstant sein, das Programm wird mit ESC beendet.

Zunächst werden wir versuchen, das Problem in einer Mischung aus natürlicher Sprache und Assemblersprache zu formulieren. Sie können dazu auch BASIC verwenden, falls Ihnen das lieber ist.

```
START:  GOSUB GETKEY
        IF KEY ='ESC' THEN END
        GOSUB BEEP
        GOTO START
```

Als nächstes übersetzen wir dieses (Fast-)Programm in die Mnemonics der 6502-Assemblersprache. Wir benutzen zunächst Marken statt konkreter Adressen, denn wir wissen noch nicht, wo das Programm später ablaufen soll (vielleicht an $300 oder an $A00).

```
START:   JSR  GETKEY
         CMP  #$9B
         BNE  SKIP
         RTS
SKIP     JSR  BEEP
         JMP  START
GETKEY:  LDA  $CØØØ
         BPL  GETKEY
         BIT  $CØ1Ø
         RTS
BEEP:    LDY  #$8Ø
BEEP1    TAX
BEEP2    DEX
         BNE  BEEP2
         BIT  $CØ3Ø
         DEY
         BNE  BEEP1
         RTS
```

Diese Form des Programms heißt Assemblersprache (fälschlicherweise oft mit "Assembler" abgekürzt), der Prozessor kann es noch nicht verstehen. Bevor wir unser Orgel-Programm starten können, müssen wir es in 6502-Maschinensprache (auch Objekt-Code genannt) "assemblieren":

```
8ØØ-20 0E 08  START:   JSR  GETKEY
8Ø3-C9 9B              CMP  #$9B
8Ø5-DØ Ø1              BNE  SKIP
8Ø7-6Ø                 RTS
8Ø8-2Ø 11 Ø8  SKIP     JSR  BEEP
8ØB-4C ØØ Ø8           JMP  START
80E-AD ØØ CØ  GETKEY:  LDA  $CØØØ
811-1Ø FB              BPL  GETKEY
813-2C 1Ø CØ           BIT  $CØ1Ø
816-6Ø                 RTS
817-AØ 8Ø     BEEP:    LDY  #$BØ
819-AA        BEEP1    TAX
81A-CA        BEEP2    DEX
81B-DØ FD              BNE  BEEP2
81D-2C 3Ø CØ           BIT  $CØ3Ø
820-88                 DEY
821-DØ F6              BNE  BEEP1
823-6Ø                 RTS
```

Jede Zeile "per Hand" zu übersetzen, ist eine mühsame Aufgabe. Nachdem Sie eine halbe Stunde lang Handbücher gewälzt haben, relative Sprünge und Adressen ausgerechnet haben, kommen Ihnen sicher Zweifel, ob es eine gute Idee war, sich mit der Maschinensprache einzulassen.

Wenn das Programm fertig übersetzt ist, muß es mithilfe des EDIT-Befehls in den Rechner eingegeben werden. Überprüfen Sie,

daß Sie beim Eintippen keinen Fehler gemacht haben, indem Sie das disassemblierte Programm mit dem Original vergleichen. Dieses Verfahren hilft Ihnen jedoch nichts, wenn Sie einen logischen Fehler im Progamm haben. Wenn Sie dann größere Veränderungen zu machen haben, dann können Sie mit dem Assemblieren gleich noch einmal von vorne beginnen.

Es muss einen besseren Weg geben

Wäre es nicht schön, wenn man diesen ganzen Prozess des Übersetzens einem Programm überlassen könnte?

Glücklicherweise gibt es solche Programme, man nennt sie "Assembler". Ein Assembler übersetzt ein Programm aus der Assemblersprache (oft auch Source-Code, Quellprogamm, genannt) in 6502-Maschinensprache (Object Code - Objektprogramm). Zu dem Assembler gehört in der Regel ein Editor, mit dem das Source-Programm in den Rechner eingegeben werden kann.

```
1000 * ASCII ORGAN
1010 *
1020 START  JSR GETKEY
1030        CMP #$9B
1040        BNE SKIP
1050        RTS
1060 SKIP   JSR BEEP
1070        JMP START
1080 *
1090 *
1100 GETKEY LDA $C000
1110        BPL GETKEY
1120        BIT $C010
1130        RTS
1140 *
1150 *
1160 BEEP   LDY #$80
1170 BEEP1  TAX
1180 BEEP2  DEX
1190        BNE BEEP2
1200        BIT $C030
1210        DEY
1220        BNE BEEP1
1230        RTS
```

Dieses Source-Programm für unsere ASCII-Orgel unterscheidet sich nur wenig von unserer handgeschriebenen Version. Der Editor, mit dem es erstellt wurde, benutzt Zeilennummern ähnlich wie APPLE-SOFT-BASIC. Die Sternchen kennenzeichnen Kommentare im Programm, sie entsprechen den REM-Befehlen in BASIC. Eine Zeile, die mit einem Sternchen beginnt, wird vom Assembler ignoriert.

Nachdem das Source-Programm eingegeben wurde, kann der Assembler es in Objekt-Code übersetzen. Das dauert bei so einem kurzen Programm wie ASCII-ORGAN nicht länger als ein paar Sekunden. Anschließend kann man das Maschinenprogramm auf Diskette abspeichern oder es laufen lassen (oder beides).

```
                1000 * ASCII ORGAN
                1010 *
0800- 20 0E 08  1020 START   JSR GETKEY
0803- C9 9B     1030         CMP #$9B
0805- D0 01     1040         BNE SKIP
0807- 60        1050         RTS
0808- 20 17 08  1060 SKIP    JSR BEEP
080B- 4C 00 08  1070         JMP START
                1080 *
                1090 *
080E- AD 00 C0  1100 GETKEY  LDA $C000
0811- 10 FB     1110         BPL GETKEY
0813- 2C 10 C0  1120         BIT $C010
0816- 60        1130         RTS
                1140 *
                1150 *
0817- A0 80     1160 BEEP    LDY #$80
0819- AA        1170 BEEP1   TAX
081A- CA        1180 BEEP2   DEX
081B- D0 FD     1190         BNE BEEP2
081D- 2C 30 C0  1200         BIT $C030
0820- 88        1210         DEY
0821- D0 F6     1220         BNE BEEP1
0823- 60        1230         RTS
```

Nachdem wir den Entstehungsprozess von ASCII-ORGAN verfolgt haben, wollen wir jetzt sehen, wie es im einzelnen funktioniert. GETKEY kennen wir noch aus Kapitel 7. Es liest die Tastatur und liefert den ASCII-Wert der gedrückten Taste im Akku ab. Wenn die ESCAPE-Taste gedrückt worden ist, dann wird das Programm mit RTS beendet. Bei jeder anderen Taste wird die BEEP-Routine aufgerufen, und anschließend beginnt der ganze Vorgang wieder von vorne.

In BEEP wird zunächst $80 ins Y-Register geladen. Y bildet den Zähler für die äußere Schleife und bestimmt damit, wie lange der Ton dauert. Die Höhe des Tons, die Frequenz, hängt vom Inhalt des Akkumulators ab (hoher ASCII-Wert - tiefer Ton). Der Inhalt des Akkus wandert ins X-Register, das wir als Zähler für die innere (Warte-)Schleife benutzen. Wenn es auf null heruntergezählt worden ist, produzieren wir einen einzelnen "Klick" auf dem Lautsprecher, dekrementieren anschließend das Y-Register und wiederholen die ganze Prozedur. Auf diese Weise produzieren wir 128 "Klicks", wobei die Länge der Pausen dazwischen über die Tastatur gesteuert wird.

Das ganze noch mal? Gut, diesmal mit einem konkreten Beipiel: JSR nach GETKEY und warten. Nach Tausenden oder Millionen von

Warteschleifen entschließt sich der Benutzer endlich, die X-Taste zu drücken. Dadurch wird der BPL-Test in der GETKEY-Routine negativ, und, nachdem der Keyboard-Strobe gelöscht worden ist, geht's mit RTS zurück ins Hauptprogramm.

Der Wert im Akku, $D8, wird mit dem ASCII-Wert für ESC, $9B, verglichen. Da die beiden nicht gleich sind, wird das gefährliche RTS umschifft, wir landen bei SKIP und rufen als nächstes BEEP auf. Da der Wert der gelesenen Taste in der inneren Schleife heruntergezählt wird, produzieren die Tasten mit den höchsten ASCII-Werten die tiefsten Töne; - es dauert einfach länger, bis der nächste Klick vom Lautsprecher kommt.

Das Programm hat einen kleinen Fehler (ich habe nie behauptet, daß es perfekt wäre): Da die innere Schleife bei hohen Tönen kürzer ist als bei tiefen, ist auch die Gesamtdauer des Tones kürzer als bei einem tiefen Ton.

Machen Sie mit dem Simulator ein paaar Runden durch ORGANPROG. Wie bei allen Programmen mit Tastaturabfrage müssen Sie darauf achten, die Taste genau dann zu betätigen, wenn $C000 ausgelesen wird.

Selbst mit größter Geduld werden Sie dem Programm nichts entlocken können, das auch nur entfernt nach Musik klingt. Der Simulator ist viel zu langsam, als daß er mehrere hundert oder gar tausend Klicks pro Sekunde erzeugen könnte. Starten Sie jetzt das Programm mit GO.

Der erste Einsender, der uns eine Kassette mit einer auf der ASCII-Orgel gespielten "Mondscheinsonate" schickt, gewinnt eine Baggerfahrt durch die Eifel! Mit Licht!!

BUBBLE SORT

Früher oder später werden Sie Daten sortieren wollen. Ein beliebtes Sortierverfahren ist BUBBLE SORT. Es gibt viele bessere Verfahren (offen gesagt, es gibt keines, das schlechter wäre als BUBBLE SORT), aber für kurze Listen lohnt kein extravagantes Verfahren.

Wir beginnen am Ende der unsortierten Liste und vergleichen jedes Element mit seinem unmittelbaren Vorgänger. Wenn die beiden noch nicht in der richtigen Reihenfolge stehen, dann tauschen sie einfach ihre Plätze, und wir vergleichen das nächste Paar. Nach dem ersten Gang durch die Liste sind wir sicher, daß das kleinste Element ganz oben gelandet ist; im zweiten Durchgang brauchen wir es daher nicht mehr zu überprüfen. Nach dem nächsten Durchgang sind die beiden ersten Elemente an ihrem richtigen Platz, usw.

Nach n-1 Durchgängen haben wir eine Liste mit n Elementen vollständig sortiert. Probieren wir es aus mit n = 4. Das Sternchen kennzeichnet das unterste Element des zu vergleichenden Paares.

1) 34
19
77
22 *

Da 22 kleiner als 77 ist, tauschen die beiden ihre Plätze. Der Zeiger wandert eins nach oben.

2) 34
19
22 *
77

Diesmal wird nicht getauscht. Zeiger eins hoch.

3) 34
19 *
22
77

19 und 34 werden vertauscht. Damit ist der erste Durchgang beendet, der kleinste Wert befindet sich bereits oben. Der zweite Durchgang beginnt wieder unten, ist diesmal aber kürzer, da das oberste Element nicht mehr getestet werden muß.

4) 19
34
22
77 *

Kein Tausch, eins hoch.

5) 19
34
22 *
77

22 und 34 tauschen die Plätze, der zweite Durchgang ist beendet. Die beiden obersten Werte sind jetzt korrekt.

6) 19
22
34
77 *

Der letzte Durchgang benötigt nur noch einen Vergleich: Da 77 nicht kleiner als 34 ist, ist kein Tausch nötig. Die Liste ist fertig sortiert!

Das ist BUBBLE SORT. Durch eine kleine Veränderung in der Abfrage können wir erreichen, daß die Liste in umgekehrter Reihenfolge sortiert wird.

Das Finanzamt verlangt von der BINÄR GmbH eine sortierte Liste mit dem Alter aller 256 Mitarbeiter. Der erste Entwurf in BASIC könnte so aussehen:

```
FOR COUNT = 0 TO 254
    (GEHE VON UNTEN NACH OBEN, VERGLEICHE ALLE
     BENACHBARTEN PAARE UND TAUSCHE, FALLS NOTWENDIG)
NEXT COUNT
```

Die innere Schleife etwas ausgearbeitet sieht folgendermaßen aus:

```
FOR POINTER = 0 TO (254-COUNT)
    IF ARRAY(POINTER) ² ARRAY(POINTER+1) THEN TAUSCH
NEXT POINTER
```

Da wir den Anfang der Liste nicht noch einmal zu testen brauchen, geht die Schleife nur bis (254-COUNT). Das Vertauschen zweier Elemente sieht so aus:

```
                  TMP = ARRAY(POINTER)
       ARRAY(POINTER) = ARRAY (POINTER+1)
     ARRAY(POINTER+1) = TMP
```

Und hier ist das komplette Programm:

```
    FOR COUNT = 0 TO 254
       FOR POINTER = 0 TO (254-COUNT)
          IF ARRAY(POINTER) ² ARRAY(POINTER+1) THEN GOSUB TAUSCH
       NEXT POINTER
    NEXT COUNT

TAUSCH:           TMP = ARRAY(POINTER)
       ARRAY(POINTER) = ARRAY (POINTER+1)
     ARRAY(POINTER+1) = TMP
```

Das Programm ist erstaunlich kurz und grauenhaft langsam. Für eine Liste mit 256 Elementen sind rund 32000 Vergleiche und etwa halb so viele Tauschoperationen notwendig (abhängig davon, wie gut die Liste vorher schon sortiert war). Das folgende Programm ist etwas schneller:

```
                    1000 * BUBBLE SORT
                    1010 *
                    1020 *
                    1030 * SORTS DATA FROM $A00 - $AFF
                    1040 * SMALLEST VALUES TO TOP
                    1050 *
                    1060 *
                    1070 COUNTR .EQ $FA
                    1080 TABLE  .EQ $A00
                    1090 *
                    1100 *
0800- A9 FF         1110 START  LDA #$FF
0802- 85 FA         1120        STA COUNTR     COUNT = 255
0804- A0 00         1130 LOOP   LDY #$00       Y IS PONTER
0806- B9 00 0A      1140 LOOP2  LDA TABLE,Y
0809- C8            1150        INY
080A- D9 00 0A      1160        CMP TABLE,Y
080D- B0 0D         1170        BCS NOSWAP
                    1180 *
                    1190 * SWAP TABLE,Y AND TABLE,Y-1
                    1200 *
080F- AA            1210        TAX            SAVE IT
0810- B9 00 0A      1220        LDA TABLE,Y
0813- 88            1230        DEY
0814- 99 00 0A      1240        STA TABLE,Y
0817- C8            1250        INY
0818- 8A            1260        TXA            RESTORE IT
0819- 99 00 0A      1270        STA TABLE,Y
081C- C4 FA         1280 NOSWAP CPY COUNTR
081E- D0 E6         1290        BNE LOOP2
0820- C6 FA         1300        DEC COUNTR
0822- D0 E0         1310        BNE LOOP
0824- 60            1320        RTS
```

Schauen wir uns das Schritt für Schritt an:

1110 - 1120 COUNTR wird mit $FE (254) initialisiert und nach jedem kompletten Durchgang durch die Liste dekrementiert. Wir sind fertig, wenn COUNTR = 0.

1130 Beginn der äußeren Schleife. Bringt uns für jeden Durchgang an das Ende der Liste.

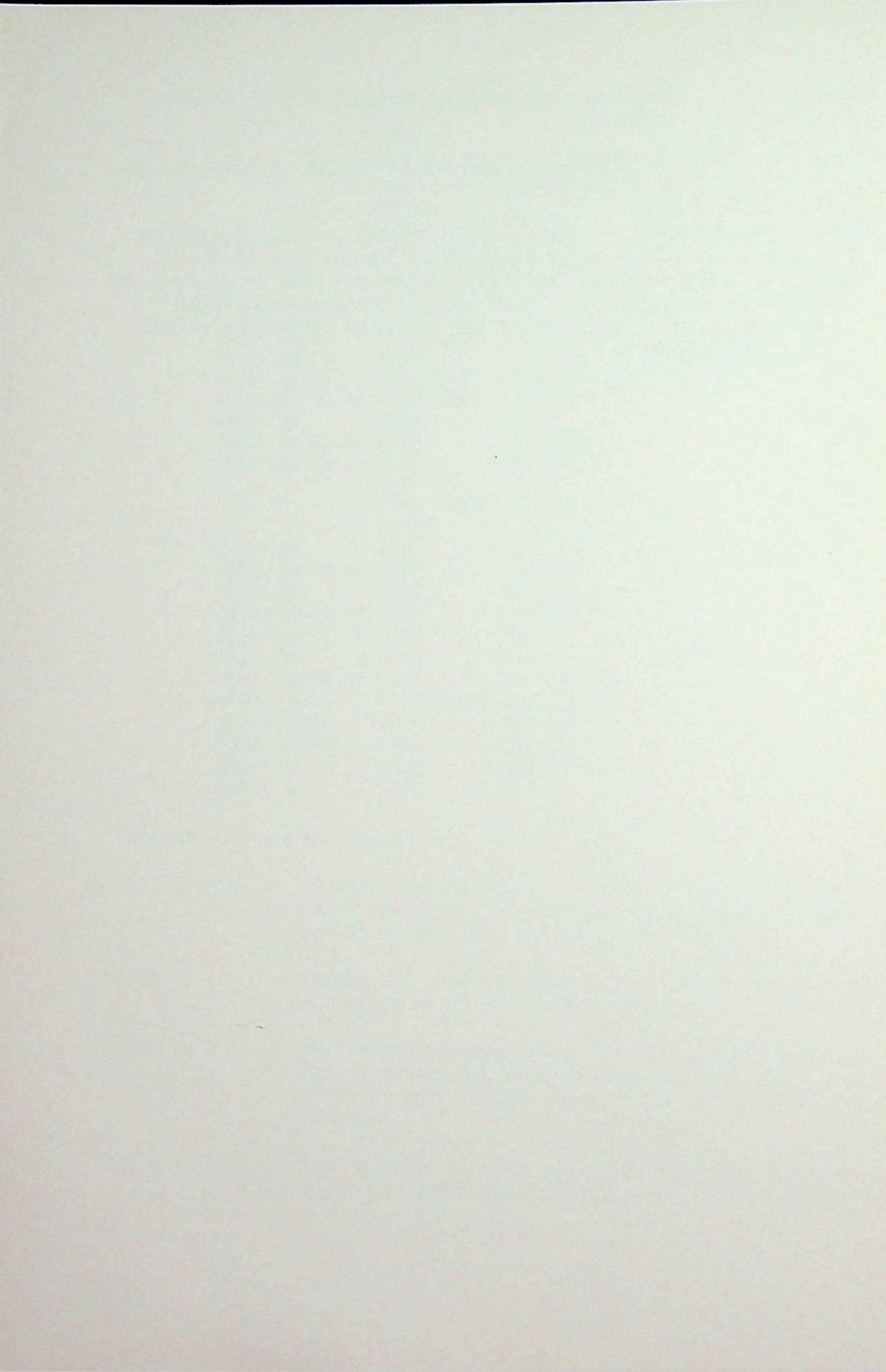
1140 Beginn der inneren Schleife. Y dient als Pointer. Wir arbeiten uns Paar für Paar hoch, bis wir den (bereits sortierten) oberen Teil der Tabelle erreichen.

1140 - 1270 Hier wird verglichen und gegebenenfalls getauscht. Jeder Durchgang durch diesen Teil inkrementiert Y.

1280	Haben wir den oberen, sortierten Teil bereits erreicht?
1300	Ein Durchgang ist beendet. Falls es der 255. war, sind wir fertig.

Laden Sie SORTPROG. An Adresse $0900 enthält es die komplette, unsortierte Liste der BINÄR-Mitarbeiter. Lassen Sie es mit GO ablaufen, und Sie werden feststellen, daß es 32000 Vergleiche und 16000 Tauschoperationen in einer knappen Sekunde bewältigt!

Wie alt ist das jüngste Mitglied der BINÄR-Belegschaft? Und wie alt das älteste?



Kapitel 16

Und wie geht's weiter?

Kaufen Sie sich einen Assembler. Sofort. Da Sie jetzt wissen, was ein Assembler leistet, sollten Sie nie wieder Ihre Zeit mit dem Nachschlagen von Opcodes und dem Berechnen von relativen Adressen verschwenden.

Besonders preiswert ist der ASSYST-Assembler. Er ist umsonst; Sie finden ihn auf der Rückseite der VC-Diskette, seine Benutzung wird im Anhang ausführlich erklärt.

Wenn Ihre Programme schließlich länger als ein- bis zweihundert Zeilen werden, dann ist es Zeit für einen ausgewachsenen Assembler. MERLIN von 'Roger Wagner Publishing' ist ein Macro-Assembler, der auch professionellen Ansprüchen gerecht wird. (Ein Macro ist eine Zusammenfassung von mehreren Assembler-Instruktionen unter einem gemeinsamen Namen; das Schreiben von Assemblerprogrammen kann dadurch stark verkürzt werden. Beim Übersetzungsvorgang setzt der Assembler dann die einzelnen Instruktionen wieder ein).

Andere gute Assembler sind ORCA/M von 'Hayden', der S-C MACRO ASSEMBLER von 'S-C Software Corporation', LISA von 'Lazerware' und der EDITOR/ASSEMBLER aus dem APPLE DOS TOOL KIT.

Wenn Sie einen APPLE //e, //c oder einen II Plus mit 16K-Karte besitzen, dann können Sie den MINI-ASSEMBLER von APPLE benutzen. Er wird zusammen mit INTEGER-BASIC geladen, wenn Sie Ihre DOS 3.3-Master-Diskette booten. Der MINI-ASSEMBLER ist ein Mittelding zwischen Assembler und Handübersetzung. Für Programme bis zu 10 Zeilen ist er kaum zu schlagen, - sobald es aber mehr wird, wird es sehr schnell chaotisch.

Welchen Assembler Sie auch immer wählen, erwarten Sie zunächst keine Wunderdinge von ihm. Sie müssen sich mit den Editor-Befehlen vertraut machen, die sich u.U. völlig von den Befehlen Ihres Textverarbeitungsprogramms unterscheiden. Dann geht es an die Pseudo-Ops, das sind Mnemonics, die nicht in 6502-Befehle übersetzt werden, sondern Anweisungen an den Assembler darstellen. Es gibt Pseudo-Ops, die angeben, für welche Adresse das Programm übersetzt werden soll, ob ein Listing erzeugt werden soll, ob eine Symboltabelle gewünscht wird, etc.

Die effiziente Benutzung eines ausgereiften Assemblers zu erlernen, kostet sicher noch einmal den gleichen Aufwand wie das Lernen der Assembersprache selbst.

"Ich habe mit meinem Assembler ein Programm geschrieben, und jetzt funktioniert es nicht ....."

Programme funktionieren nie gleich auf Anhieb, und Assemblerprogramme schon gar nicht! Zum Austesten eines Maschinenprogrammes können Sie VISIBLE COMPUTER oder auch den altbewährten APPLE-MONITOR benutzen.

Der APPLE-Monitor

Der MONITOR ist vielleicht nicht so ausführlich und auch nicht so fehlertolerant wie VC, aber er hat alles, was Sie zum Austesten eines Programmes benötigen. In den einschlägigen APPLE-Handbüchern ist er zudem ausführlich beschrieben.

Lassen Sie uns eine kleine Beispielsitzung mit dem MONITOR beschreiben. Booten Sie eine normale DOS 3.3-Diskette und geben Sie nach dem BASIC-Prompt das Kommando "CALL -151" ein. Der MONITOR meldet sich mit einem Sternchen. DOS ist immer noch aktiv. Sie können zwar keine BASIC-Kommandos (RUN, LIST, etc.) mehr benutzen, denn sie ergeben jetzt keinen Sinn mehr; dafür können jetzt die DOS-Komandos CATALOG, BLOAD, etc. benutzt werden.

Laden Sie ein Testprogramm von der Diskette oder tippen Sie eins direkt ein; mit dem Kommando "L" können Sie es auf dem Bildschirm auflisten (disassemblieren). Während Sie beim L-Kommando des VC nur fünf Zeilen auf dem Bildschirm sehen, sehen Sie jetzt zwanzig, und zwar blitzschnell. (Ein gutes Beispiel für den Geschwindigkeitsunterschied zwischen BASIC und Maschinensprache.)

Starten Sie Ihr Programm , indem Sie die Anfangsadresse gefolgt von einem "G" eintippen. Im Gegensatz zu VC werden keine höflichen Meldungen ausgegeben, wenn Ihr Programm auf einen undefinierten Opcode läuft oder sich an verbotenen Adressen zu schaffen macht. Sobald es sich bis zum letzten RTS durchgeschlagen hat, wird die Kontrolle wieder an den MONITOR übergeben.

Wie kann man ein Programm in handliche Stücke zerlegen, die einzeln ausgeführt werden? Zum Glück gibt es das BRK-Kommando. In Kapitel 15 haben wir gelernt, das BRK ($00) einen Sprung nach $FA62 verursacht, eine ROM-Routine, die zunächst nachsieht, ob ein Hardware-Interrupt vorliegt, anschließend den Inhalt der Register ausgibt und den MONITOR aufruft.

Pflastern Sie die strategischen Stellen Ihres Programms großzügig mit $00, Sie können dann den Ablauf des Programms kontrollieren und nachsehen, welchen Inhalt die Register zu einem bestimmten Zeitpunkt haben.

Darüberhinaus hat der MONITOR Befehle für einfache Hex-Arithmetik, für das Verschieben von Speicherbereichen, sowie verschie-

dene andere Tricks. Schauen Sie in Ihrem 'Reference'-Handbuch nach.

BASIC und Maschinensprache

In Kapitel 1 haben wir erklärt, daß der clevere APPLE-Programmierer nur dann Assembler benutzt, wenn er es nicht mehr vermeiden kann. Selbst dann versucht er es zunächst mit einer Mischung aus BASIC für das Grundgerüst und Maschinensprache für die zeitkritischen Aufgaben. Beim Entwurf solcher "Hybridprogramme" kann ein grundlegendes Verständnis von BASIC nicht schaden.

Was ist BASIC?

APPLESOFT ist ein (ziemlich langes) 6502-Maschinenprogramm im ROM-Bereich $D000 - $F800; darüberhinaus belegt es rund die Hälfte der Zero Page für seine eigenen Variablen (das APPLESOFT-Programmierhandbuch beschreibt diese im Detail).

Bei der Benutzung von APPLESOFT arbeiten Sie viel mehr mit Konzepten und Ideen, anstatt mit Adressen und Registern. Sie können z.B. eingeben: X = Y + Z, ohne sich darum kümmern zu müssen, wo der Inhalt von X im Speicher abgelegt ist. Sie können eine Formel wie: R = SIN(2*THETA) aus einem Mathe-Buch abschreiben, ohne zu wissen, wie der Sinus intern berechnet wird. BASIC schlägt die Brücke zwischen diesen (fast umgangssprachlichen) Befehlen und der Acht-Bit-Welt des 6502 mit seinen 56 Instruktionen.

Sie haben in diesem Buch gelernt, daß der 6502 weder Schach spielen noch die Einkommenssteuer ausrechnen kann. Man kann 6502-Programme schreiben, die so etwas können, der 6502 selbst kann jedoch nur Addieren, Subtrahieren, Shiften, Verzweigen, etc.

Ein BASIC-Programm ist eine lange Liste, die von einem Maschinenprogramm namens APPLESOFT verwaltet wird. Die Liste beginnt bei $801 und wächst im Speicher weiter nach oben (eigentlich gibt es noch eine zweite Liste, die von oben nach unten wächst; - wenn sich die beiden treffen, ist Feierabend).

Der 6502 führt dieses BASIC-Programm nicht direkt aus. Wenn Sie den Rechner bei der Ausführung eines BASIC-Programm stoppen, dann werden Sie im Programmzähler niemals einen Wert finden, der kleiner als $D000 ist. Stattdessen führt der 6502 im ROM ein ausgeklügeltes Programm aus, das ständig in dieser BASIC-Programm genannten Datenliste nachsieht, was als nächstes zu tun ist. Wenn es mit dieser Liste nicht klar kommt oder etwas Unvernünftiges ausführen soll (z.B. eine Division durch null), dann unterbricht es die Ausführung und gibt eine Meldung auf dem Bildschirm aus.

Das APPLESOFT-Handbuch enthält Informationen, wie APPLESOFT seine Daten organisiert, vor allem, wie die Variablen angelegt

werden. Darüberhinaus finden wir dort wenig über die interne Arbeitsweise des Interpreters, - APPLE betrachtet das als Firmengeheimnis.

Es gibt jedoch andere Bücher (z.B. "APPLESOFT-Trickkiste") und vor allem viele Zeitschriftenartikel über APPLESOFT. BASIC-Anhänger haben sich die Mühe gemacht, den Interpreter Stück für Stück zu zerlegen, immer auf der Suche nach Unterprogrammen, die man auch von Assemblerprogrammen direkt aufrufen kann. Warum soll man sich ein neues Sinus-Unterprogramm oder eine Grafik-Routine schreiben, wenn es das alles schon gibt?

Mit dem BASIC-Kommando CALL können Sie ein Maschinenprogramm aufrufen. Es benötigt die Startadresse als Dezimalzahl, also z.B. CALL 640, wenn das Unterprogramm bei $280 beginnt.

Solange Ihre Maschinenprogramme klein genug bleiben, können Sie in den ersten $D0 Bytes der Seite 3 untergebracht werden. Der Befehl CALL 768 taucht in so vielen BASIC-Programmen auf, daß man ihn schon fast für einen festen Bestandteil der Sprache halten könnte.

Seite 2 wird von der GETLN-Routine des MONITOR als Eingabepuffer benutzt (diese wird ihrerseits von APPLESOFT bei der Bearbeitung von INPUT aufgerufen). Solange Sie keine extrem langen Zeilen eingeben wollen, können Sie den hinteren Teil des Puffers für Ihre Programme benutzen.

Auf der Zero Page ist für Programme kein Platz; trotzdem haben DOS, APPLESOFT und MONITOR noch ein paar Lücken gelassen, die Sie für Ihre Variablen und Zeiger benutzen können.

Wenn Sie mehr Platz brauchen, dann müssen Sie Ihr BASIC-Programm verschieben. Es beginnt normalerweise an Adresse $801, doch das kann man ändern. Ein Zeiger an Adresse $67,$68 deutet auf den Anfang des Programms; wenn Sie den Inhalt verändern, wird das nächste Programm an die neue Adresse geladen. Das folgende kurze BASIC-Programm setzt den Programmanfang auf $4001. Dadurch werden 6K für Maschinenprogramme frei, und die Hi-Res-Seite 1 liegt gut geschützt unterhalb des BASIC-Bereiches. 22K bleiben für das BASIC-Programm übrig, das sollte in den meisten Fällen reichen.

```
110 POKE 104,64
120 POKE 16384,0
130 PRINT CHR$(4);"RUN PROGRAM"
```

Aus irgendeinem (guten) Grund erwartet APPLESOFT eine Null vor der ersten Speicheradresse.

Wenn Ihr Maschinenprogramm nur wenige Parameter benötigt, dann POKEn Sie diese am besten in die entsprechenden Speicherstellen. Bei mehr oder längeren Parametern kann das Maschinenprogramm direkt auf die APPLESOFT-Variablen zugreifen. Das ist allerdings schwieriger und verlangt eine genaue Kenntnis der Struktur der BASIC-Variablen.

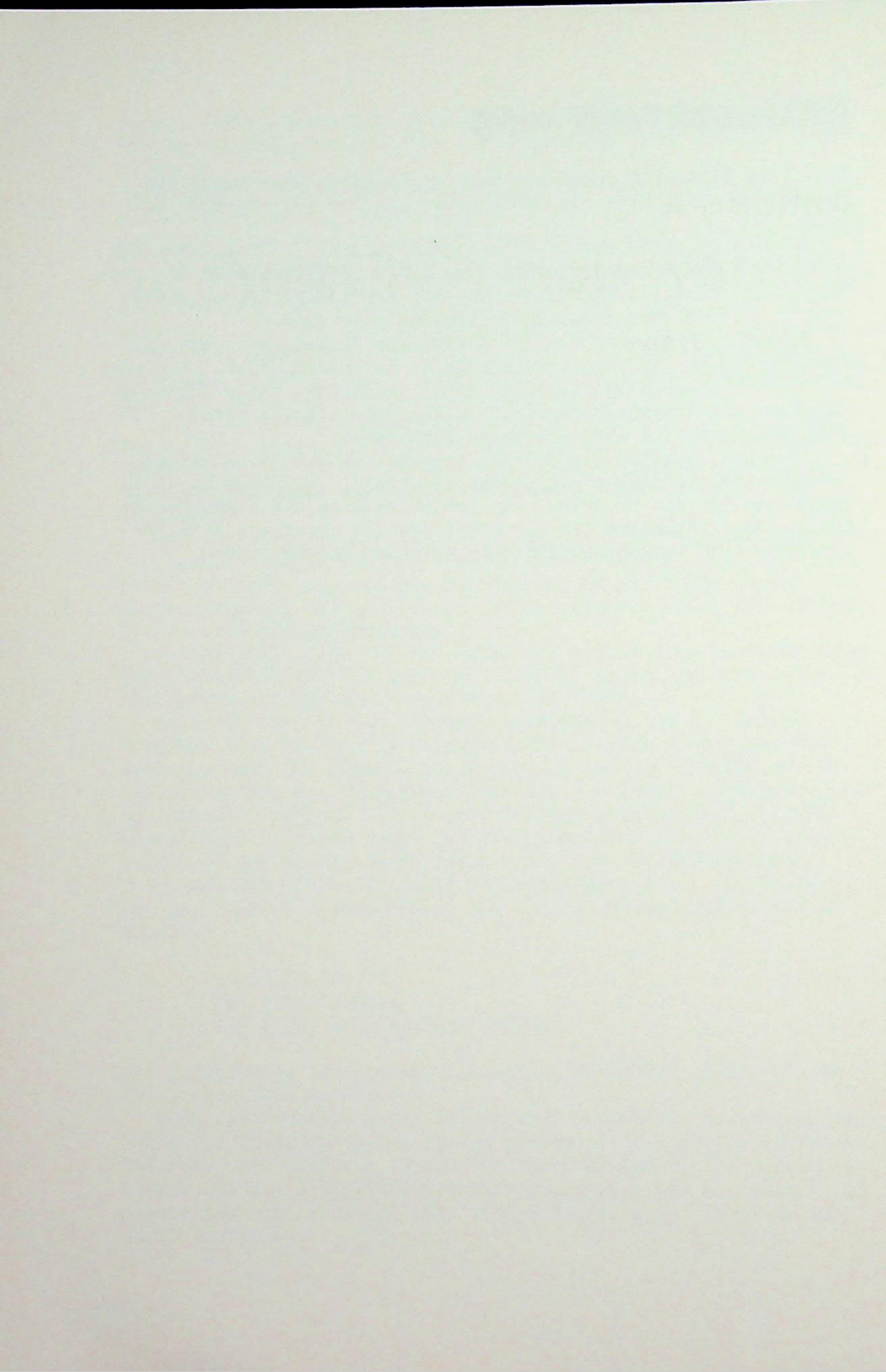
Schlussbemerkung

Natürlich haben Sie in diesem Kurs nicht alles über Maschinensprache gelernt. Hier sind drei Tips, wie Sie weitermachen sollten:

Erstens, lesen Sie gute Bücher. Beurteilen Sie Computerbücher nicht nach dem Umschlag, sondern nach dem Inhalt. Lassen sie sich Bücher von Freunden empfehlen, die mit der Assemblerprogrammierung vielleicht schon weiter fortgeschritten sind.

Zweitens, studieren Sie die Assemblerprogramme von anderen. Das MONITOR-Listing in Ihrem 'APPLE II Reference Manual' ist eine Fundgrube für gute Ideen. Jeden Monat erscheinen viele Computermagazine mit Assemblerprogrammen aller Schwierigkeitsgrade. Suchen Sie sich eins aus und wühlen sich durch.

Drittens, starten Sie eigene Projekte. Fangen Sie klein an (obwohl auch harmlose Problemchen manchmal ihre Tücken haben): Schreiben Sie ein Programm, das den Bildschirm mit einem Muster füllt, oder verändern Sie ein existierendes Programm. Arbeiten Sie sich dann langsam an die schwierigeren Sachen heran.



Anhang A

Hinter den Kulissen von VC

Speicherbelegung durch VC

```
------------------------------------------------------------
$BFFF:
     :  DOS
$9AA6:
     :  VISIBLE COMPUTER (BASIC-Programm)
$4000:
     :  HIRES Page 1  (wird von VC benutzt)
$2000:
     :  VC-Tabellen
     :
$1A00:  VC Maschinenspracheroutinen
     :
     :  DOS TOOL KIT HIRES-Zeichengenerator
$0EFF:
     :  JSR-Behandlung
$0E00:
     :  VC-Stack
$0D00:
     :  1K Benutzerspeicher
$0800:
     :  Text-Seite 1
$0400:
     :  Seite 3; $3D0-$3FF reserviert für DOS
$0300:
     :  GETLN-Puffer. Wird von VC nicht benutzt
$0200:
     :  6502-Stack
$0100:
     :  6502 ZERO PAGE (VC benutzt $06-$09 und $FE,$FF)
$0000:
------------------------------------------------------------
```

VISIBLE COMPUTER besteht aus einem BASIC-Programm, das den Hauptteil der Arbeit erledigt, und aus Unterprogrammen in Maschinensprache, die das Ergebnis der SHIFT- und LOGICAL-Operationen ausrechnen. BASIC ist für diese Bit-Fummelei nur schlecht geeignet.

Um sich vor dem Benutzer zu schützen, verwendet VC eine eigene Zero Page und einen eigenen Stack. Sie können das verfolgen, wenn Sie die Bereiche $0000-$01FF und $0C00-$0DFF vergleichen.

Im MASTER-Modus haben Sie zwar Zugriff auf die echte Zero-Page, nicht jedoch auf den echten Stack. Dieser ist nur Programmen zugänglich, die mit GO gestartet werden; der Simulator benutzt dagegen immer den fiktiven Stack an $D00.

Wenn Sie Monitor-Unterprogramme im NOSHARE-Modus ausführen lassen, kann es leicht zu Pannen kommen, wenn die erwarteten Parameter nicht an den entsprechenden Stellen in der fiktiven Zero Page stehen. Wenn Sie dagegen auf die echte Zero Page zugreifen, müssen Sie darauf achten, keine Speicherstellen zu verändern, die für BASIC oder DOS wichtig sind. So ist das Leben eines Assembler-Programmierers!

Übrigens ...

VC ist ein Hilfsmittel zum Erlernen der Maschinensprache und nützlich beim Testen von Assemblerprogrammen. VC ist keine exakte Darstellung der inneren Vorgänge des 6502-Mikroprozessors. VC führt alle 151 Opcodes korrekt aus; die Resultate werden aber oft auf anderen Wegen erreicht. Die Darstellung der "Mikroschritte" bei der Befehlsausführung dient nur dem konzeptionellen Verständnis - sie sind nicht notwendigerweise mit den realen Vorgängen identisch.

Anhang B

ASCII-Zeichensatz

		Most Significant Bits							
HEX		0	1	2	3	4	5	6	7
	BINARY	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

Anmerkungen

Die Codes $00-$1F produzieren keine abdruckbaren Zeichen sondern "kontrollieren" ein angeschlossenes Gerät. Code $46 veranlasst einen Drucker, ein "F" auszugeben, Code $0C dagegen bewirkt einen Seitenvorschub.

Auf der Apple-Tastatur können die Codes $01-$1A mit Hilfe der "Ctrl"-Taste und den Buchstaben A-Z erzeugt werden. Wichtige Controll-Zeichen haben ihre eigene Taste: RETURN produziert den gleichen Code ($0D) wie Ctrl-M.

Einige interessante Codes:

Code		Bedeutung
$07	BEL	BELL; erzeugt einen Ton
$08	BS	BACKSPACE; bewegt den Cursor nach links
$0A	LF	LINE FEED; Cursor eine Zeile nach unten
$0C	FF	FORM FEED; Seitenvorschub
$0D	CR	CARRIAGE RETURN; Cursor an den Anfang der Zeile
$1B	ESC	ESCAPE

- Die Linkspfeil-Taste erzeugt ASCII-Code $08, die Rechts-Taste $15. Auf dem Apple //e erzeugt der Pfeil nach oben den Wert $0B und der Pfeil nach unten den Wert $0A.

- Die IIplus-Tastatur kann keine Kleinbuchstaben generieren.

- Die Appletastatur generiert Zeichen, bei denen Bit 7 gesetzt ist.

- Die Codes für Kleinbuchstaben sind die gleichen wie die für Großbuchstaben, außer daß Bit 5 gesetzt ist.

- Die niedrigsten 4 Bits in den ASCII-Codes der Ziffern entsprechen dem Wert der Ziffer. Mit dem Befehl "AND #$0F" kann man so das ASCII-Zeichen in die entsprechende Binärzahl umwandeln.

Anhang C
VC Monitor-Befehle

Der Monitor-Modus wird durch das "#"-Zeichen in der letzten Zeile angezeigt. VC erwartet dann eines der folgenden Kommandos.
Monitor-Befehle haben die allgemeine Form:

Kommando (Argument1) (Argument2)

Kommandos und Argumente müssen durch mindestens ein Leerzeichen voneinander getrennt sein, dürfen jedoch selbst keine Leerzeichen enthalten!
Im folgenden bedeutet:

adresse	Eine 16-Bit Zahl
wert	Ein 8-Bit oder 16-Bit Wert
register	Eines der Register auf dem Bildschirm: DL, DB, IR, A, S, P, X, Y, PC, AD, MEMA, MEMD
filename	Ein Dateiname (ohne Leerzeichen!)

Schrägstriche trennen äquivalente Parameter, Klammern umschließen optionale Parameter.

BASE

Syntax: BASE register/ALL/MEM/MON HEX/BIN/DEC

Zweck: Legt fest, in welcher Basis die Zahlen auf dem Bildschirm dargestellt oder vom Monitor interpretiert werden. Anstelle von "register" kann man mit Hilfe von ALL alle Register zusammen ansprechen. MEM ändert die Speicherdarstellung, MON bestimmt die Basis für Monitoreingaben.

Bsp: BASE PC BIN

BLOAD

Syntax: BLOAD filename (P3/P2)

Zweck: Lädt Programme und Daten, die zuvor mit BSAVE gespeichert wurden. Die Standard-Ladeadresse ist $800, das File muß den DOS-Konventionen entsprechen.
Nach jedem BLOAD überprüft VC, ob es nicht selbst überschrieben wurde. In diesem Fall versucht es von der Diskette neu zu booten.
Infolge des verwendeten Kopierschutzes ist es nicht möglich, im MASTER-Modus von der VC-Diskette oder im NON-MASTER-Modus von einer normalen DOS 3.3-Diskette zu lesen.

Bsp: BLOAD MAGNUMOPUS P3

BSAVE

Syntax: BSAVE filename (P3/P2)

Zweck: Speichert den angegebenen Speicherbereich auf die Diskette in Laufwerk 1 unter "filename". P3 (Page 3) speichert $D0 Bytes von $300-$3CF, P2 speichert den Bereich $200-$2FF. Wenn kein Argument angegeben wird, dann wird der Bereich $800-$BFF abgespeichert.

Bsp: BSAVE MAGNUMOPUS

CALC

Syntax: CALC

Zweck: Mit diesem Kommando wird der eingebaute Kalkulator aufgerufen. Die Operatoren +, -, * und / stehen zur Verfügung. Wie bei den Monitorkommandos müssen die einzelnen Operanden und Operatoren durch Leerzeichen voneinader getrennt sein.
Mit Ctrl-H, -B oder -D (Hex, Bin, Dez) kann die Zahlenbasis des Kalkulators verändert werden. Um die Zahl $3CF dezimal darzustellen, schalten Sie zunächst auf hexadezimal (Ctrl-H), geben dann 3CF ein und schalten anschließend mit Ctrl-D auf dezimal um.
ESC beendet den Kalkulator, jedes andere Zeichen löscht die Eingabezeile.
Ein Ergebnis größer als 65535 oder kleiner als -32767 veranlasst eine Fehlermeldung. Negative Zahlen werden als

Zweier-Komplement dargestellt. Binäre Zahlen werden bei der Ausgabe mit Leerzeichen formatiert - bei der Eingabe dürfen sie jedoch keine Leerzeichen enthalten!

Bsp: CALC

CASE

Syntax: CASE UPPER/LOWER

Zweck: Schaltet Groß- oder Kleinbuchstaben ein. Kleinbuchstaben sind Standard.

Bsp: CASE UPPER

EDIT

Syntax: EDIT adresse

Zweck: Ermöglicht das Verändern der Speicherinhalte. Nach dem Aufruf erscheint die gewählte Adresse und der aktuelle Inhalt.

Wie bei CALC hat jetzt die zuerst betätigte Taste ein besondere Bedeutung: RETURN und die Rechtspfeil-Taste rufen die nächsthöhere Adresse auf, der Linkspfeil geht zurück zur vorangegangenen Adresse. ESC führt zurück zum Monitor, jede andere Taste ruft die Eingaberoutine auf. Der eingegebene Wert ersetzt den vorherigen Inhalt der angewählten Adresse.

Bsp: EDIT 900

ERASE

Syntax: ERASE

Zweck: Löscht den Bildschirm.

Bsp: ERASE

GO

Syntax: GO

Zweck: Die Ausführung eines Unterprogramms wird direkt an den 6502 übergeben. Voraussetzung: VC befindet sich im

MASTER-Modus und die erste Instruktion der Routine ist ein JSR.

Falls VC durch das laufende Programm nicht zerstört worden ist, kehrt die Kontrolle an VC zurück, der Bildschirm zeigt den Zustand der Register nach Ausführung des Programms.

Bsp: GO

L

Syntax: (adresse) L

Zweck: Beginnend bei "adresse", werden 5 Instruktionen disassembliert. Wenn keine Adresse angegeben wird, dann wird bei der zuletzt disassemblierten Adresse fortgesetzt.

Bsp: A00 L

LC/RC

Syntax: LC/RC adresse

Zweck: Nach dem Öffnen von WINDOW kann die Startadresse für die rechte oder linke Spalte spezifiziert werden.

Bsp: RC 900

LOAD MEMORY

Syntax: adresse wert

Zweck: Eine Abkürzung des EDIT-Kommandos. Der alte Inhalt von "adresse" wird durch "wert" ersetzt.

Bsp: A00 FF

LOAD REGISTER

Syntax: register wert

Zweck: Der Inhalt eines Registers kann verändert werden. Bei einem 16-Bit-Register kann "wert" 0-65535 umfassen, für 8-Bit-Register sind nur die Werte 0-255 erlaubt.

Bsp: PC 300

MASTER

Syntax: MASTER ON/OFF

Zweck: Der MASTER-Modus ist nur für fortgeschrittene VC-Benutzer. Er ermöglicht den Zugriff auf alle Adressen und die Ausführung der Kommandos GO, ZP und BSAVE. Im MASTER-Modus können normale DOS 3.3-Disketten gelesen und beschrieben werden, der Zugriff auf die VC-Diskette ist jedoch nicht mehr möglich.

Bsp: MASTER OFF

POP

Syntax: POP

Zweck: Ein RTS wird simuliert: Der Program-Counter wird mit den beiden obersten Bytes vom Stack geladen.
POP ist nützlich, um aus langweiligen Warteschleifen heraus zu kommen, oder um festzustellen, auf welchem Weg man in ein Unterprogramm gekommen ist.
POP wird ignoriert, wenn S größer als $FD ist.

Bsp: POP

PRINTER

Syntax: PRINTER ON/OFF

Zweck: Die disassemblierten Zeilen werden auch auf dem Drucker ausgegeben. Wenn der Drucker nicht eingeschaltet ist, sich nicht in Slot 1 befindet oder Sie ein Nicht-Standard Interface benutzen, kann sich VC "aufhängen". In diesem Fall hilft nur RESET.

Bsp: PRINTER ON

RESTART

Syntax: RESTART

Zweck: VISIBLE COMPUTER wird auf die Anfangswerte zurückgesetzt (Warm Start). Der Benutzerbereich wird davon nicht beeinflußt.

Bsp: RESTART

RESTORE

Syntax: RESTORE

Zweck: Gegenstück zu ERASE. Der Bildschirm wird neu gezeichnet.

Bsp: RESTORE

STEP

Syntax: STEP 0/1/2/3

Zweck: Kontrolliert die Geschwindigkeit des 6502-Simulators. Die einzelnen Stufen werden in Anhang D beschrieben.

Bsp: STEP 3

WINDOW

Syntax: WINDOW OPEN/CLOSE/MEM

Zweck: Mit diesem Kommando wird der Inhalt des mittleren Bildschirm-Drittels bestimmt. Voreingestellt ist CLOSE, nur der Prozessor wird dargestellt. Mit MEM werden 16 Speicherstellen (in zwei Spalten, siehe LC/RC) dargestellt, OPEN löscht die Speicherdarstellung.

Bsp: WINDOW MEM

ZP

Syntax: ZP SHARE/NOSHARE

Zweck: Legt die Benutzung der Zero Page fest und ist nur im MASTER-Modus zulässig. Im SHARE-Modus wird die echte Zero Page mit DOS, BASIC und VC geteilt, im NOSHARE-Modus wird eine fiktive Zero Page an $D00 benutzt.

Bsp: ZP SHARE

Anhang D
6502-Simulator

Der Simulator ist der Teil von VISBLE COMPUTER, der die 6502-Maschinensprache ausführt. Die 151 definierten Opcodes werden, zerlegt in entsprechende Mikroschritte, auf dem Bildschirm abgearbeitet. Undefinierte Opcodes werden ignoriert.

Wenn der Simulator aktiv ist, wird in der ersten Zeile des "Message"-Fensters (links oben) entweder FETCH angezeigt (wenn der FETCH-Zyklus läuft) oder die Instruktion, die gerade ausgeführt wird.

Mikroschritte

In der zweiten Zeile wird der Mikroschritt dargestellt, der gerade ausgeführt wird. VC unterscheidet bis zu 9 Mikroschritte, die zur Ausführung der Instruktion notwendig sind:

CALC ADDRESS	AD wird mit Hilfe von X oder Y modifiziert
COMPUTE	Eine arithmetische, logische oder SHIFT-Operation wird durchgeführt
COND FLAGS	Die Flags werden konditioniert
DEC	Ein Register wird dekrementiert
INC	Ein Register wird inkrementiert
READ	Der Inhalt von Adresse AD wird in DATA LATCH gebracht
T:	Der Inhalt eines Registers wird in ein anderes Register transferiert
TEST FLAG	Der Zustand eines Flags wird festgestellt

WRITE	Der Inhalt von DATA LATCH wird an die Speicherstelle AD geschrieben

Kontrolle des Simulators

Die Arbeitsweise des Simulators kann durch das Kommando STEP variiert werden:

STEP 3 Der ausführlichste und langsamste Modus. Nach jedem Mikroschritt hält der Simulator an und wartet, bis die Leertaste betätigt wird. Nach Abarbeitung der Instruktion Rückkehr zum Monitor.

STEP 2 Keine automatischen Pausen zwischen den Mikroschritten, sondern nur nach Drücken der Leertaste. Anschließend Rückkehr zum Monitor.

STEP 1 Wie Modus 2, jedoch wird nach Beendigung der Instruktion automatisch die nächste Instruktion bearbeitet. Mit ESC geht's zurück zum Monitor.

STEP 0 Ähnlich wie Modus 1, jedoch wird der Bildschirm während der Ausführung nicht aktualisiert. Mit ESC wird der Monitor aufgerufen, die Register zeigen dann den aktuellen Wert an.
Dies ist der schnellste Simulator-Modus, er benötigt etwa 2 Sekunden pro Instruktion.

In den Modi 1-3 kann die Geschwindigkeit der Ausführung durch die Tasten 1 (schnell) bis 9 (langsam) verändert werden.

Der Simulator kann jederzeit durch Betätigen der Leertaste angehalten werden. Mit Hilfe von C kann dann der Kalkulator aufgerufen werden. Nach Verlassen des Kalkulators kehrt man in den Haltezustand zurück; nach erneutem Drücken der Leertaste setzt der Simulator seine Arbeit fort.

Anhang E

Fehlermeldungen

BAD FNAME	Der Filename, der als Argument für BLOAD oder BSAVE angegeben wurde, enthält unzulässige Zeichen.
BAD OPCODE	Der Simulator hat einen undefinierten Opcode angetroffen.
BASE	VC kann die eingegebene Zahl nicht interpretieren. Überprüfen Sie bitte, ob die eingegebenen Ziffern in der gerade eingestellten Zahlenbasis zulässig sind.
COMMAND	Der Kommandointerpreter kann Ihre Eingabe nicht verstehen.
DISK FULL	Kein Platz mehr auf der Diskette
DIV BY 0	Der Kalkulator sollte durch Null dividieren
FILE LOCK	Sie versuchten ein File zu speichern, das bereits als "gelocktes" File existiert.
I/O	Umfassst alle Arten von Disk-Fehlern: Diskette nicht richtig eingelegt, Tür nicht geschlossen, Diskette nicht initialisiert, etc. Diese Fehlermeldung erhalten Sie auch dann, wenn Sie versuchen, im MASTER-Modus die VC-Diskette zu lesen oder im NON-MASTER-Modus auf eine normale DOS 3.3-Diskette zugreifen wollen.
MISMATCH	Sie versuchen ein File zu lesen, das nicht vom Typ "B" (Binär) ist, oder ein File zu schreiben, das bereits mit dem gleichen Namen, aber einem anderen Typ, existiert.

NO FILE	File wurde nicht gefunden
NOT JSR	Ein GO-Kommando wurde gegeben, aber die erste Instruktion ist kein JSR.
NOT MASTER	Sie haben ein Kommando gegeben, das nur im MASTER-Modus erlaubt ist.
RANGE	Sie haben einen Wert eingegeben, der für die gewählte Adresse zu groß ist.
W.PROTECT	Sie versuchen auf eine schreibgeschützte Diskette zu schreiben. (Legen Sie ihre eigenen Programme immer auf einer normalen DOS 3.3-Diskette ab und nie auf der VC-Diskette)
ER XXXX-XX	Ein Fehler in VISBLE COMPUTER selbst ist aufgetreten. Entweder haben Sie MASTER-Modus einen Teil des Programmes zerstört oder Sie haben einen Fehler im Programm entdeckt. Im zweiten Fall schreiben Sie uns bitte!

Vermeiden Sie die Eingabe von Ctrl-C! Es passiert nichts Aufregendes, es kann aber sein, das VC danach nicht mehr richtig funktioniert.
RESET ist nur in äußersten Notfällen zu benutzen, bspw. wenn sich der Rechner wegen eines nicht angeschlossenen Druckers "aufgehängt" hat.

Anhang F

ASSYST: Das Assemblersystem

von John T. Cox

Handbuch von John T. Cox und Charles Palmquist

(C) Copyright 1983 Thunder Software

Einführung

ASSYST ist ein interaktiver Editor/Assembler für die APPLE II-Familie. Er ist für Hobby-Programmierer und als Lehrmittel konzipiert, nicht als Werkzeug für professionelle Anwender. Dem Anfänger erleichtert er den Einstieg in die 6502-Assemblerprogrammierung, indem er auf komplizierte Extras verzichtet.

Das Handbuch besteht zum größeren Teil aus der Beschreibung einer typischen Arbeitssitzung mit ASSYST. Im anschließenden Nachschlageteil werden die einzelnen Kommandos noch einmal detailliert erklärt, außerdem werden die Fehlermeldungen des Assemblers aufgelistet und weitere Informationen über ASSYST gegeben. Eine Liste aller Files, die sich auf der Diskette befinden, schließt das Handbuch ab.

Der beste Weg, sich mit ASSYST vertraut zu machen, ist: Legen Sie das Handbuch neben die Tastatur und probieren Sie die einzelnen Beispiele der Reihe nach aus!

Es geht los

Im folgenden wird vorausgesetzt, daß Sie sich mit Ihrem APPLE und dem Betriebssystem DOS 3.3 auskennen. (Um das Programm mit allen Rechnern der APPLE II-Familie kompatibel zu machen, werden im

folgenden nur Großbuchstaben verwendet. Wenn Sie einen APPLE //e oder //c haben, dann rasten Sie am besten die Feststelltaste ein.) Die Programmdiskette ist nicht kopiergeschützt; bevor Sie anfangen, machen Sie sich bitte mithilfe von COPYA oder einem anderen Kopierprogramm eine Kopie.

Booten Sie Ihre Diskette und schauen Sie sich zunächst einmal das Hauptmenu an:

```
(E)DIT          CREATE OR CHANGE A FILE
(A)SSEMBLE      ASSEMBLE A PROGRAM FILE
(C)ATALOG       LIST DISK CATALOG
(R)ENAME        RENAME A DISK FILE
(D)ELETE        DELETE A DISK FILE
(Q)UIT          EXIT PROGRAM
(X)ECUTE        RUN A BINARY PROGRAM
(L)IST          LIST AN ASSEMBLED FILE
```

Jedes Kommando wird mit seinem ersten Buchstaben abgekürzt. Geben Sie ein "C" ein, um sich das Inhaltsverzeichnis der Diskette anzusehen. Sie enthält mehrere Files, die durch ein "B" als Binärfiles gekennzeichnet sind; es sind dies die Programme, die ASSYST selbst bilden. Files, die durch ein "T" als Textfiles ausgewiesen sind, sind Beispielprogramme, an denen im folgenden die Arbeitsweise von ASSYST demonstriert werden soll.

Benutzen Sie zunächst das Kommando (R)ENAME und ändern Sie den Namen des Files TEST FILE in TFILE. Vergewissern Sie sich an Hand von (C)ATALOG, daß alles geklappt hat.

Der Editor

Bei der Assemblerprogrammierung werden Sie die meiste Zeit mit dem Editor verbringen, um Textfiles damit zu erzeugen oder zu verändern. Der ASSYST-Editor wird mit (E)DIT aufgerufen. Auf die Frage nach dem Eingabefile (WHAT FILE DO YOU WISH TO EDIT?) antworten Sie mit TFILE. ASSYST erwartet einen zweiten Namen als Ausgabefile (AND WHAT FILE FOR THE OUTPUT?), Sie antworten darauf mit APLHA BEEP. Wenn Sie zweimal den gleichen Namen verwenden, wird ein neues File unter diesem Namen eröffnet; es ist daher Vorsicht geboten: Ein bereits existierendes File mit diesem Namen geht dann verloren.

ASSYST durchsucht jetzt die Diskette nach TFILE und lädt die ersten 100 Zeilen in den Speicher. Auf dem Bildschirm sehen Sie zunächst nur eine Titelzeile, die Sie an den Editor erinnert, und einen einsamen Doppelpunkt. Bevor Sie jetzt aus TFILE Ihr erstes Assemblerprogramm machen, probieren Sie das Kommando .H aus (der Punkt vor dem "H" ist wichtig: der Editor erkennt daran, daß es sich um ein Kommando handelt und nicht um eine Eingabe).

.H zeigt Ihnen eine Übersicht aller Editor-Kommandos. Wie Sie sehen, beginnen alle Kommandos mit einem Punkt. Die meistgebrauchten sind sicherlich .U(Gehe um Zeilen nach oben) und .D(Gehe Zeilen nach unten). Probieren Sie .D5 (ohne Leerzeichen vor der Zahl), dann .U2 und .D18. Da der Bildschirm nur 16 Zeilen auf einmal zeigt, verlieren Sie dabei den Anfang des Files aus den Augen. Die oberste Zeilennummer ist 0 (unmittelbar vor der ersten Zeile), die unterste ist 100. Wenn Sie die Zeile 100 überschreiten, dann speichert der Editor die ersten hundert Zeilen (zusammen mit Ihren Änderungen) auf Diskette und liest die nächsten 100 Zeilen ein.

Beachten Sie: Sie können zu dem ersten Teil, der bereits abgespeichert wurde, nur dadurch zurückkehren, daß Sie den Editor verlassen und wieder von vorne beginnen.

Zeilennummern

Die Zeilennummern des ASSYST-Editors helfen Ihnen, sich innerhalb der Files zu bewegen und diese abzuändern. Im Gegensatz zu APPLESOFT sind sie jedoch nicht Bestandteil des Programmes, sondern werden vom Editor hinzugefügt. Die erste Zeile hat die Nummer 1, die zweite die Nummer 2, usw. Wenn Sie Zeilen einfügen oder löschen, dann wird alles automatisch neu durchnummeriert.

Die aktuelle Zeile

Die letzte Zeile auf dem Bildschirm ist die sogenannte "aktuelle Zeile". Mithilfe der Kommandos .U und .D suchen Sie sich Ihre "aktuelle Zeile" aus und verändern sie anschließend mit dem Kommando .M:

MODIFY

Probieren Sie es aus. Machen Sie die Zeile 33 zur aktuellen Zeile und geben Sie .M ein. Die folgende Zeile kann jetzt modifiziert werden:

```
33*;THS IS A COMMMANT
```

Bewegen Sie den Cursor mit der Rechtspfeiltaste zum "S" in "THS". Mit CTRL-I wird der INSERT-Modus eingeschaltet: alle nachfolgenden Zeichen werden an der Cursorposition eingefügt, und der Rest der Zeile wird nach rechts verschoben. Nachdem Sie aus THS ein THIS gemacht haben, können Sie den INSERT-Modus mit ESC wieder abschalten. Gehen Sie jetzt zum zweiten "M" in COMMMANT. Mit CTRL-D löschen Sie das Zeichen links vom Cursor. Anschließend ersetzen Sie noch das "A" durch ein "E", und Zeile 33 ist korrigiert:

33 ;THIS IS A COMMENT

Mit der RETURN-Taste verlassen Sie den MODIFY-Modus (Sie brauchen dazu nicht bis zum Ende der Zeile zu gehen, wie in APPLESOFT).

Benutzen Sie jetzt die Kommandos .U, .D und .M und ändern Sie die Zeile 11 von:

11 ORC $0800

in:

11 ORG $0800

Mit dem Kommando .Ekönnen Sie ein oder mehrere Zeilen komplett löschen. .E6 beispielsweise löscht 6 Zeilen rückwärts(!), beginnend mit der aktuellen Zeile. Wenn Ihre aktuelle Zeile vorher 19 war, dann werden die Zeilen 14-19 gelöscht, und die neue aktuelle Zeile ist danach 14 (die alte Zeile 20). Gehen Sie zur Zeile 23:

23 ;ERASE ME

und probieren Sie es aus: .E1

INPUT

Ab und zu ist es notwendig, Zeilen komplett neu einzugeben (insbesondere, wenn man mit einem leeren File neu anfängt). Gehen Sie zur Zeile 27 und probieren Sie .I. Sie befinden sich jetzt im INPUT-Modus (nicht zu verwechseln mit dem INSERT-Modus) und können neue Zeilen einfügen:

; THIS IS AN UNNECESSARY COMMENT
; AND SO IS THIS

Im INPUT-Modus funktionieren auch alle Tricks des MODIFY-Modus: CTRL-I, CTRL-D, usw. Wenn Sie Ihre Zeilen eingefügt haben, so kommen Sie mit jedem Editor-Kommando (.U .D, etc.) wieder zurück in den Kommando-Modus.

SAVE

Wenn Sie alle Änderungen ausgeführt haben, dann verlassen Sie den Editor mit .S. Das File wird unter dem Namen, den Sie zu Beginn als OUTPUT-File angegeben haben (ALPHA BEEP), auf der Diskette abgelegt.

QUIT

Sollten Sie einmal mehr verpfuscht als korrigiert haben, dann können Sie den Editor auch mit .Q verlassen. In diesem Fall wird

kein OUTPUT-File erzeugt, und Ihre Änderungen bleiben ohne Konsequenzen.

Der Assembler

Mit dem Kommando (A)SSEMBLE wird der Assembler geladen (aus Platzgründen ist das ein separates Programm). Auf die Frage WHICH FILE DO YOU WISH TO ASSEMBLE? antworten Sie mit ALPHA BEEP. Während der Assembler Ihr Programm übersezt, wird die gerade begearbeitete Zeile auf dem Bildschirm angezeigt. ASSYST ist ein sogenannter Two-Pass-Assembler (Doppelschrittassembler), d.h. es sind zwei Durchgänge nötig, um den Programmtext korrekt zu übersetzen.

Der zweite Durchgang ist viel schneller als der erste, und wenn Sie keine Fehler gemacht haben (sowas soll schon vorgekommen sein), dann wird das Resultat als BIN.ALPHA BEEP auf die Diskette geschrieben.

SAVE THE LISTING FILE? (Y/N)

Antworten Sie mit Y (Ja). Das Listing-File enthält den ursprünglichen Programmtext zusammen mit der Übersetzung (in hexadezimaler Form) und den Adressen. Es wird als ASM.ALPHA BEEP gespeichert und kann später ausgedruckt werden. Das ist vor allem bei der Fehlersuche nützlich.

Auch die Symboltabelle sollten Sie abspeichern. Sie wird an das Listing angehängt und hilft später bei der Interpretation desselben.

(A)SSEMBLER
(E)DITOR/MAIN MENU
(L)ISTER
(Q)UIT

Bevor Sie Ihr Programm starten, können Sie sich mithilfe des LISTERS das Ergebnis des Assemblers ansehen. Er gibt ASM.ALPHA BEEP entweder auf dem Bildschirm oder auf einem Drucker aus.

Kehren Sie mit (E)DITOR/MAIN MENU ins Hauptmenu zurück und wählen Sie (X)ECUTE. Als Filename geben Sie BIN.APLHA BEEP ein, und, man sehe und höre und staune: Auf dem Bildschirm erscheinen die 26 Buchstaben des Alphabets, begleitet von einem ansteigenden Ton.

Kommandoübersicht Editor

.D Bewegt den Cursor um Zeilen nach unten.

.E Löscht die letzten Zeilen.

.H Gibt eine Kurzübersicht über die Editorbefehle auf dem Bildschirm aus.

.I Versetzt den Editor in den INPUT-Modus. Neue Zeilen können eingegeben werden, die MODIFY-Kommandos CTRL-I und CTRL-D sind zugelassen. Wird durch jedes andere Editor-Kommando beendet.

.M MODIFY-Modus. Erlaubt das Verändern der gerade aktuellen Zeile. CTRL-I schaltet den INSERT-Modus ein, neue Zeichen können an der Cursor-Position eingefügt werden. ESC beendet INSERT. CTRL-D löscht das Zeichen links vom Cursor.

.Q Verläßt den Editor, ohne die Änderungen abzuspeichern.

.S Der geänderte Text wird im OUTPUT-File abgelegt, und ASSYST kehrt zum Hauptmenu zurück.

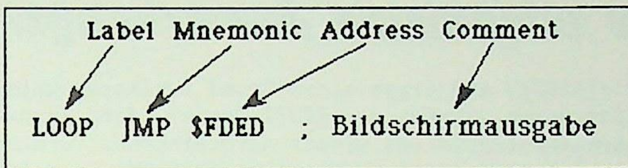
.U Bewegt den Cursor um Zeilen nach oben.

Anmerkungen:

- Ein neues File wird erzeugt, indem Sie am Anfang für INPUT- und OUTPUT-File den gleichen Namen angeben.
- Vergessen Sie nicht, den INSERT-Modus mit einem ESC abzuschalten, bevor Sie den MODIFY-Modus verlassen.
- Eine Zeile kann max. 68 Zeichen enthalten.

Kommandoübersicht Assembler

Jede Zeile eines Assemblerprogramms besteht in der Regel aus vier Teilen: einer Marke (LOOP), dem Befehl (JMP), einer Adresse ($FDED) und dem Kommentarfeld (;Bildschirmausgabe). Diese vier Teile sind durch mindestens ein Leerzeichen voneinander getrennt. Eine Marke (max. 6 Buchstaben) §&muß immer&§, ein Kommentar §&kann&§, und eine Zeile ohne Marke §&darf nicht&§ in der ersten Spalte anfangen.



ASSYST versteht folgende Symbole:

; Kennzeichnet einen Kommentar, der vom Assembler ignoriert wird.

$ Kennzeichnet eine Hexadezimalzahl.

% Kennzeichnet eine Binärzahl (max. 8 Bits).

Bezeichnet den Direkt-Adressierungsmodus.

' Zwei Apostrophs umschließen den String im ASC-Pseudo-Op.

+ Während der Assemblierung wird eine (dezimale) Konstante zu dem Wert einer Marke hinzuaddiert.

- Bezeichnet die Subtraktion einer Konstanten von einer Marke.

Pseudo-Ops

ORG Bestimmt die Anfangsadresse des Maschinenprogramms. Die ORG-Anweisung darf nur einmal auftauchen und muß die erste Anweisung im Programm sein.

ORG $800

ASC Konvertiert die folgende Zeichenkette in ASCII-Codes.

MESSGE ASC 'GEBEN SIE EINE ZAHL EIN'

EQU Weist einer Marke einen Wert zu.

SPEAKR EQU $C030

= Entspricht EQU.

SPEAKR = $C030

HEX Belegt einen Speicherplatz (ein oder zwei Byte) mit einem Hexadezimalwert.

ZEIGER HEX $FFFE

Einige allgemeine Anmerkungen

- Wenn Sie ASSYST aus irgendeinem Grund verlassen haben (mit QUIT oder durch Drücken der RESET-Taste), dann können Sie mithilfe des Ampersands (&) wieder zurückkehren. Voraussetzung dafür ist, daß die Speicherbereiche $300-$324 und $1000-$61A8 nicht verändert worden sind. Dies kann nützlich sein, wenn Sie Ihre Programme mithilfe des APPLE-MONITORS testen wollen.

- ASSYST legt den assemblierten Code an Adresse $800-$FFF ab. Diese 2K sollten für Programme von 600 bis 1200 Zeilen ausreichen. (Für längere Programme ist ASSYST sowieso ungeeignet)

- ASSYST kann maximal 250 Marken verarbeiten. (Wenn Sie weniger benutzen, dann beschleunigt das den Übersetzungsvorgang.)

- Dezimale Konstanten können Sie nur mit einem Trick verwenden:

```
     Nicht erlaubt ist:               LDA 7
     Erlaubt ist dagegen:       DUMMY EQU $0000
                                      LDA UMMY + 27
```

- ASSYST ist in APPLESOFT-BASIC geschrieben und mit dem MICROSOFT-Compiler übersetzt worden.

Fehlermeldungen

HEXDEC ERROR	Eine Hexadezimal ist zu groß, um in eine Dezimalzahl verwandelt werden zu können, oder sie enthält ein unerlaubtes Zeichen.
INVALID HEX DIGIT	Eine Hexadezimalzahl enthält ein unerlaubtes Zeichen.
IRRESOLVABLE LABEL REFERENCE	Eine undefinierte Marke wird benutzt.
IRRESOLVABLE ADRESSING MODE	Die angegebene Adressierungsart ist nicht erlaubt.
INVALID USE OF ASCII TYPE	Der Apostroph ist nur in Verbindung mit ASC erlaubt.
NO CLOSING QUOTES	Der zweite Apostroph fehlt in der ASC-Anweisung.

INVALID USE OF %	Binärzahlen dürfen nur aus Nullen und Einsen bestehen.
INVALID STATEMENT	Der Assembler kann die Zeile nicht interpretieren (wahrscheinlich eine Kommentarzeile, bei der das Semikolon vergessen wurde).
NO ORIGIN SET	Die ORG-Anweisung fehlt.
INVALID ADDRESS	Der Adressteil ist nicht zu interpretieren.
RELATIVE BRANCH OUT OF RANGE	Ein BRANCH-Befehl reicht nicht weiter als 127 Bytes vorwärts oder 128 Bytes rückwärts.
UNKNOWN OPCODE	Instruktion ist unbekannt.
APPLESOFT ERROR	Ein Fehler im ASSYST-Programm sollte eigentlich nicht vorkommen. Schreiben Sie uns.

Weitere Bücher aus dem Pandabooks-Verlag:

Mikrocomputer Grafik
von Roy E. Myers
ISBN 3-89058-000-9 DM 49,--

Apple II Assembler Programmierung
von Roger Wagner
ISBN 3-89058-003-3 DM 48,--

Apple II Raster Grafik
von Jeffrey Stanton
ISBN 3-89058-006-8 DM 49,--

Applesoft Trickkiste
Call A.P.P.L.E. in Depth
ISBN 3-89058-033-5 DM 44,--

Apple Pascal Grafik
von Tom Swan
ISBN 3-89058-009-2 DM 49,--

Apple II Schaltpläne
von Winston D. Gayler
ISBN 3-89058-012-2 DM 64,--

Apple ProDOS Handbuch
von Don Worth und Peter Lechner
ISBN 3-89058-036-X DM 46,--

Apple Pascal Trickkiste
Call A.P.P.L.E. in Depth
ISBN 3-89058-030-0 DM 48,--

BASIC-Programme für Funkamateure
von Wayne Overbeck und James A. Steffen
ISBN 3-89058-027-0 DM 48,--

Sie haben einen Apple...

wir haben die Software und die Hardware... wir haben die Bücher und die Zeitschriften*...

pandasoft

ALLES FÜR DEN APPLE II, II+, IIE

HARDWARE · SOFTWARE · BÜCHER · ZEITSCHRIFTEN

*Fordern Sie unseren Gratiskatalog an!

UNSERE ADRESSE:

pandasoft

UHLANDSTR. 195 · D-1000 BERLIN 12

TEL.:(030) 310 423

Ich besitze einen Apple. Bitte schicken Sie mir Ihren kostenlosen Katalog.

Name: ....

Adresse: ....

MERLIN

von Glen Bredon

Ein professioneller Macro–Assembler für
die Apple II–Familie!
Neben allen Standard–Features bietet MERLIN u.a.:

- komfortabler Editor mit globalen
  Such– und Ersetzfunktionen
- liest und schreibt Text– unf Binärfiles
- unterstützt 6502 und 65C02 Opcodes
- beinhaltet eine Bibliothek mit vielen
  nützlichen Unterprogrammen
- enthält einen Dissassembler
- kompatibel mit vielen 80–Zeichen–Karten
  und natürlich mit Apple //e und //c!
- deutsches Handbuch

Ausführliche Informationen von:

Pandabooks
Bismarckstr. 67
D–1000 Berlin 12

Ein Simulationsprogramm, das Sie in das Innere des 6502–Mikroprozessors führt. Sie sehen auf dem Bildschirm, wie die einzelnen Instruktionen in Zeitlupe ausgeführt werden, wie sich die Register und Flags verändern. Ein unverzichtbares Hilfsmittel beim Erlernen der Assemblerprogrammierung, danach ein wertvolles Werkzeug beim Testen Ihrer eigenen Programme.
Komplett mit Editor/Assembler und einem Lehrbuch zur 6502–Maschinensprache.

Für Apple *II*, *II* plus, *IIe*, *IIc*.

Pandabooks Berlin **ISBN: 3–89058–019–X**