Der ATARI® Assembler

Ihr Wegweiser zu aufregend neuen Erkenntnissen im Bereich der Personal-Computer! Lernen Sie, wie man den **ATARI ASSEMBLER MODUL** benutzt, und entdecken Sie die Vorteile des Programmierens in Assembler für den Atari 400 bzw. 800.

Oliff-61

Don Inman / Kurt Inman

Der Atari Assembler

Don Inman / Kurt Inman

Der Atari Assembler

IDEA

CIP-Kurztitelaufnahme der Deutschen Bibliothek
Inman, Don:
Der Atari-Assembler / Don Inman; Kurt Inman.
[Aus d. Amerikan. übertr. von Adrienne Deutsch]. –
Puchheim: IDEA, 1983.
Einheitssacht.: The atari assembler <dt.>
ISBN 3-88793-025-8
NE: Inman, Kurt:

ISBN 3-88793-025-8

Gesamtherstellung: Mayer-Druck, Augsburg
Printed in Germany

Aus dem Amerikanischen übertragen
von Adrienne Deutsch

Umschlagentwurf von Steve Oliff

Titel der Originalausgabe
THE ATARI ASSEMBLER

Inhaltsverzeichnis

Abbildungen

Vorwort

Mit dem enormen Anwendungszuwachs von Mikrocomputern, wie z. B. dem Atari 400 bzw. 800, nimmt auch die Zahl derer, die sich einen eigenen Rechner kaufen, ständig zu. Diese neuen Benutzer möchten natürlich die Fähigkeiten eines so vielseitigen Gerätes optimal nutzen. Nach dem Erlernen von BASIC ist ein Assembler der nächste logische Schritt.

Der Atari Assembler Modul eignet sich hierfür geradezu ideal. Er ist außerordentlich leistungsfähig und trotzdem leicht verständlich und einfach anzuwenden. Er besitzt alle Voraussetzungen, um einem das mühsame Programmieren in Maschinensprache zu ersparen.

Das vorliegende Buch wurde im wesentlichen unter Berücksichtigung folgender Gesichtspunkte geschrieben:

(1) einfache, detaillierte Richtlinien für die Benutzung des Atari Assembler Moduls zu liefern;

(2) grundlegende Informationen über das Programmieren in Assembler zu geben.

Das Buch ist für Anfänger in der Assembler-Programmierung, die einige BASIC-Kenntnisse mitbringen, gedacht. Die Benutzung des Assemblers wird schrittweise genau beschrieben. Eingabe und Ausführung von Programmen wird anhand von Bildschirmskizzen im Zwischen- und Endstadium gezeigt. Erklärungen zur Maschinensprache erfolgen in Wort und Bild. Nichts blieb unversucht, das Buch möglichst leicht lesbar und verständlich zu machen.

Wir halten es für richtig, das Programmieren in Assembler auf dem Umweg über BASIC zu lernen. Wir gehen davon aus, daß Sie Grundkenntnisse in BASIC besitzen, und werden versuchen, Sie nach und nach mit der Assembler-Sprache vertraut zu machen, wobei wir Ihr BASIC-Wissen verwenden wollen.

Damit Sie überprüfen können, was Sie gelernt haben, gibt es nach jedem Kapitel Übungsfragen und Antworten. Dieses Buch soll keine vollständige Darstellung des Atari Assembler Moduls oder des 6502-Befehlsvorrates geben, dennoch ist es ausführlich genug, um Ihnen deren Anwendung auf einem gehobenen Niveau zu ermöglichen. Das bis dahin Erlernte soll Sie ermutigen, Ihre Assembler-Kenntnisse weiter zu vertiefen.

Einleitung

Dieses Buch befaßt sich in erster Linie mit der Bedienung und Anwendung des Atari Assembler Moduls. Der Assembler Modul wird anstelle des BASIC Moduls in den linken Schlitz des Atari Rechners geschoben.

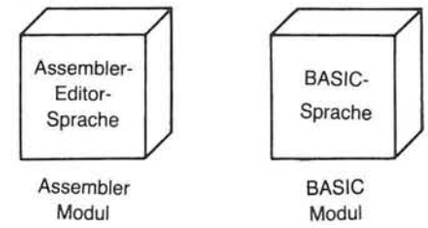


Abb. 1-1 Sprachmodule

Durch die Assembler Modul Programme wird ein Assembler-Programm geschrieben und editiert. Es wird in Maschinencode assembliert, und das Maschinenprogramm erzeugt schließlich bei seiner Ausführung die gewünschten Resultate. Abb. 1-2 zeigt die einzelnen Schritte dieser Prozedur.

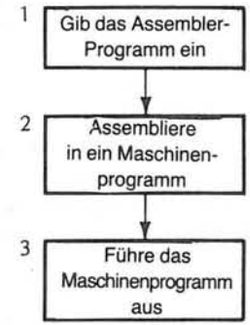


Abb. 1-2 Programmschritte beim Assembler

Die Besprechung des Assembler Moduls und seine Anwendung beginnt in Kapitel 4. Statt sich Hals über Kopf ins kalte Wasser der Assembler-Programmierung zu stürzen, sollten Sie dies Problem lieber langsam angehen. In diesem und den folgenden Kapiteln werden Sie mit dem Aufbau des Atari bekannt gemacht. Außerdem erfolgt eine kurze Wiederholung von BASIC, womit Sie ja, wie wir annehmen, einigermaßen vertraut sind. In Kapitel 2 und 3 (siehe Abb. 1-3) lernen Sie, wie man Programme in Maschinensprache innerhalb eines eigenen BASIC-Programms erstellen und ausführen kann. Wir haben diesen Weg gewählt, weil wir annehmen, daß die meisten Benutzer des Atari 400 bzw. 800 in Atari-Basic programmieren werden, das ihnen als Modul zur Verfügung steht. Auf diese Weise wird Ihnen das Erlernen einer Maschinensprache mittels einer Sprache, die Sie bereits kennen, ermöglicht.

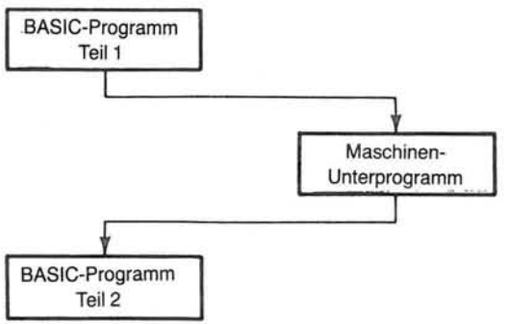


Abb. 1-3 Maschinen-Unterprogramm von BASIC

BASIC-Befehle müssen zunächst (mittels des BASIC Moduls) übersetzt oder interpretiert werden, ehe sie der Computer versteht. Sie können daher nicht so schnell ausgeführt werden, wie Befehle, die in Maschinensprache geschrieben sind. Zudem verbraucht ein BASIC-Programm mehr Speicherplatz als ein äquivalentes Programm in Maschinensprache.

Es ist nun möglich, den größten Teil eines Programms in BASIC, den Rest, bei dem es auf besonders schnelle Ausführung ankommt, in Maschinensprache zu schreiben. Diese in Maschinensprache formulierten Teile des Programms werden vom BASIC-Programm als Unterprogramme (subroutines) angesprochen. Diese Unterprogramme „versteht" der Rechner sofort, und die Zeit, die andernfalls für die Interpretation nötig ist, wird eingespart. Diese Methode wird in den Kapiteln 2 und 3 sorgfältig dargestellt und anhand von Musterbeispielen schrittweise erläutert.
Der Rechner führt zwar Maschinenbefehle schneller aus, als dies bei in BASIC geschriebenen Befehlen möglich ist, dafür braucht man aber länger, um ein Programm in Maschinensprache zu schreiben.
Die Maschinensprache hat einige Nachteile:
1. Jeder Befehl hat seinen eigenen *numerischen* Code, den der Rechner versteht. Entweder muß man diese Codes auswendig lernen (um Himmels Willen!) oder jedesmal, wenn man sie braucht, in einer Tabelle nachsehen:

Beispiele:

Hex Code	*Ausgeführter Befehl*
A9	Lade den Akkumulator mit der dem Hex-Code folgenden Zahl.
AD	Lade den Akkumulator mit der Zahl, die unter der dem Hex-Code folgenden Adresse abgespeichert ist.
8D	Speichere den Wert aus dem Akkumulator unter der dem Hex-Code folgenden Adresse ab.

2. Jeder Befehlscode (Op Code) in Maschinensprache muß in der richtigen Reihenfolge gespeichert werden, damit das Programm richtig arbeitet.

3. Sollte der Programmierer Verzweigungen machen, um die sequentielle Abarbeitung zu ändern, so muß er ganz genau berechnen, wie viele Speicherplätze er überspringen muß, damit er wieder beim richtigen Befehl ankommt, und dies entsprechend programmieren.

4. Das Programmieren in Maschinensprache erfordert ungeheure Kleinarbeit, und die Gefahr, dabei Fehler zu machen, ist erheblich.

Durch die Assembler-Sprache fallen viele Nachteile des Programmierens in Maschinensprache weg. Man vergleiche die folgende Auflistung mit den Nachteilen der Maschinensprache.

1. Jeder Befehl in Assemblersprache hat seinen eigenen Buchstabencode, der eine Abkürzung der auszuführenden Operation ist.

Beispiele:

Zahlencode	*Operand*	*Ausgeführter Befehl*
LDA	#14	Lade den Akkumulator mit der Zahl 14.
LDA	$1100	Lade den Akkumulator mit dem Wert, der unter der Adresse 1100 abgespeichert ist.
STA	$1105	Speichere den Wert aus dem Akkumulator unter der Adresse 1105 ab.

Mit dieser Codierung kann der Programmierer viel leichter arbeiten als mit dem Zahlencode der Maschinensprache.

2. Die Assemblersprache hat, ähnlich wie BASIC, Zeilennummern, und das Programm wird vom Assembler automatisch in der richtigen Reihenfolge abgespeichert.

3. Mittels Marken (Kombinationen von Worten, Buchstaben und/oder Zahlen) werden Verzweigungen zu markierten Befehlen, und die komplizierten Berechnungen fallen weg (der Assembler erledigt das für Sie).

4. Der Assembler erstellt für Sie aus dem Buchstabencode das Programm in Maschinensprache, wodurch die zeitraubende Kleinarbeit vollkommen entfällt. Die Gefahr, Programmierfehler zu machen, ist viel geringer als bei Programmen, die man selbst in Maschinensprache schreiben würde.

Zunächst befassen wir uns damit, wie der Rechner in seiner eigenen Sprache arbeitet, um uns danach in Kapitel 4 dem Atari Assembler Modul zuzuwenden.

RECHNERAUFBAU

Wir nehmen an, daß Sie mit der Atari-BASIC-Sprache vertraut sind und nun die Fähigkeiten des Rechners in seiner eigenen Sprache erforschen möchten.
Das Programmieren in Maschinensprache erfordert, daß Sie die wichtigsten Bausteine des Atari Computers kennenlernen. Hierbei handelt es sich vor allem um die Zentraleinheit (Central Processing Unit = CPU) und den Speicher.

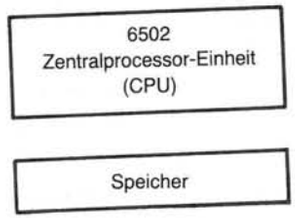


Abb. 1-4 Bausteine des Atari

Sie müssen sich klar machen, wie man den Speicher des Rechners zum Ablegen und Wiederauffinden von Informationen benutzt. Zu diesem Zweck werden Sie die Binärcodes kennenlernen, mit denen man dem Rechner Befehle erteilt. Diese Codes nennt man die Befehlsliste der Zentraleinheit. Dies klingt sehr kompliziert, aber wir gehen ganz langsam, Schritt für Schritt, vor und werden laufend wiederholen.

Die Atari Computer 400 bzw. 800 benutzen eine Zentraleinheit mit der Bezeichnung 6502. Der 6502 gehört zu den Mikroprozessoren mit der Produktbezeichnung MCS650X, die von MOS Technology, Inc. entwickelt wurden. Er kann eine Befehlsliste (siehe Anhang A), die durch den Maschinencode gegeben ist, ausführen. Die Befehle sind 8-stellige Binärzahlen (8 Bits).

Binärziffern können nur die Werte 1 (Eins) oder 0 (Null) annehmen. Ein Befehl in Maschinensprache, den der 6502 erkennen kann, besteht aus 8 dieser Binärziffern.

Beispiele:

Binärcode	*Befehl*
10101001	Lade den Akkumulator mit der Zahl, die diesem Befehl folgt.
10101101	Lade den Akkumulator mit dem Wert, der unter der Adresse, die nach diesem Befehl kommt, gespeichert ist.
10001101	Speichere den Wert aus dem Akkumulator unter der Adresse, die diesem Befehl folgt, ab.

Sie sehen, daß jeder Binärcode genau 8 Binärstellen (8 Bits) lang ist. Einen Block von 8 Bits nennt man ein *Byte*. Bei dem im oben angegebenen Befehl erwähnten Akkumulator handelt es sich um einen besonderen Speicherplatz, das sogenannte *Register*. Es kann genau ein Byte enthalten.
Der Akkumulator (oft mit Register A bezeichnet) ist wahrscheinlich das am häufigsten verwendete Register (ein besonderer Zwischenspeicher). Alle Operationen zwischen verschiedenen Speicherzellen laufen über den Akkumulator oder eines der anderen Register. Werden Daten aus einer Speicherzelle in eine andere gebracht, dient der Akkumulator als Zwischenspeicher. Die Bearbeitung von Daten geschieht im Akkumulator, der daher bei vielen Assembler- oder Maschinen-Befehlen verwendet wird. Der Befehlsfolge entsprechend, „akkumuliert" er die Ergebnisse aufeinanderfolgender Operationen.
Am besten stellt man sich den Mikroprozessor 6502 als eine Kombination verschiedener Funktionselemente vor. Abb. 1-5 zeigt die Register (spezielle Speicherzellen innerhalb der Zentraleinheit), die der Rechner bei der Ausführung vieler verschiedener Schritte immer wieder benutzt. Einige dieser Funktionselemente dienen ganz besonderen Zwecken, während andere dem Programmierer für allgemeinere Vorhaben zur Verfügung stehen.

Abb. 1-5 Funktionselemente des Mikroprozessors 6502

Der Datentransfer zwischen dem Speicher und den internen Registern der Zentraleinheit geschieht über 8 in zwei Richtungen verlaufenden Datenübertragungsleitungen, dem sogenannten Daten-Bus. Jedes Bit eines Daten-Bytes läuft über eine eigene Leitung. Die Übertragung eines ganzen Bytes (8 Bit) geschieht gleichzeitig. Obwohl also jedes Bit seinen eigenen Weg geht, wird das Byte vom Daten-Bus als Einheit übertragen. Man nennt daher die Struktur des Mikroprozessors 6502 eine Byte-orientierte Struktur. Die 8 Übertragungsleitungen werden als Bus bezeichnet.

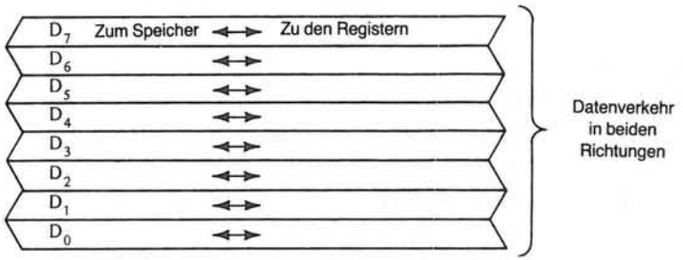
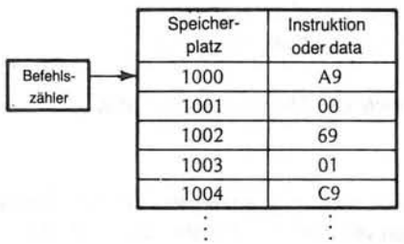


Abb. 1-6 Datenbus

Die Register X und Y werden auch als Zwischenspeicher verwendet. Sie bieten darüber hinaus die Möglichkeit, ihren Inhalt durch einen entsprechenden Befehl um eins zu erhöhen bzw. zu erniedrigen. Daher können die Register X und Y als Zähler (oder Zeiger) verwendet werden, um Daten in aufeinanderfolgende Speicherzellen zu bringen oder sie von solchen Zellen zu holen (laden). Eben diese Verwendungsmöglichkeit, nämlich auf aufeinanderfolgende Speicherzellen zu zeigen (indizieren), hat zu der Bezeichnung Index-Register geführt. Sie können auch als Zähler verwendet werden, um so Bedingungen für das Ende einer Schleife (einer wiederholt durchlaufenen Folge gleicher Befehle) zu liefern. Die Befehlsliste des 6502 enthält verschiedene Spezialbefehle, um die Index-Register mit zuvor festgelegten Werten zu laden, damit die Ausführung von Schleifen in ähnlicher Weise, wie dies in BASIC bei den FOR-NEXT-Schleifen geschieht, erleichtert wird.

Der 16-Bit Befehlszähler fungiert als Befehlsadresszeiger, um den Befehlsablauf in der gewünschten Reihenfolge zu sichern. Die Befehle eines Programms werden in aufeinanderfolgenden Speicherzellen abgelegt. Letztere enthalten Befehlscodes in Maschinensprache und/oder Adressen-Bytes und Zahlen, die bearbeitet werden sollen. Um die gewünschte Reihenfolge der Schritte eines Programms zu steuern, wird der Befehls-(Adress-) Zähler als Zeiger verwendet, um diejenige Speicherzelle zu bezeichnen, in der der jeweils nächste Befehl für den Mikroprozessor steht. Der Befehlszähler wird, nachdem ein Befehl „geholt" worden ist, um eins erhöht, und zeigt somit die Adresse des nächsten Befehls an.



Speicher-platz	Instruktion oder data
1000	A9
1001	00
1002	69
1003	01
1004	C9
⋮	⋮

Abb. 1-7 Befehlszähler

Der Stapelspeicher (stack) ist ein spezieller Teil des Gesamtspeichers, der Daten in der Reihenfolge abspeichert, wie sie ankommen. Sie werden gewissermaßen „gestapelt", was zur Folge hat, daß das letzte abgespeicherte Datum auch als erstes wieder

Abb. 1-8 Last-In, First-Out Stapel (zuletzt gestapelt, zuerst weggeholt)

den Speicher verläßt (Last-In, First-Out). Das 16-Bit Stapelregister führt Buch über den verwendeten Stapelspeicherplatz; es enthält die Adresse des Stapelkopfes (des zuletzt abgespeicherten Datums). Das Stapelregister und der Befehlszähler sind groß genug (16 Bits), um eine Adresse maximaler Länge (von 0 bis 65535) speichern zu können.

Einzelne Bits des 8-Bit Status-Registers der Zentraleinheit werden verwendet, um Sonderwirkungen von Befehlen auf den „Status" des Rechners zu verfolgen. Das Vorhandensein bzw. Nichtvorhandensein einer solchen Sonderwirkung zeigt sich daran, ob ein gewisses Bit auf eins oder Null gesetzt ist. Diese einzelnen Bits nennt man auch Zustandsbits (flag). Die vom Mikroprozessor verwendeten Zustandsbits sind Carry, Zero, Interrupt, Decimal, Break, Overflow und Negative. Wir werden sie genauer untersuchen, wenn wir sie zum Verständnis von Befehlen benötigen. In Kapitel 2 werden Sie als erstem dem Zustandsbit Carry (Carry flag) begegnen.

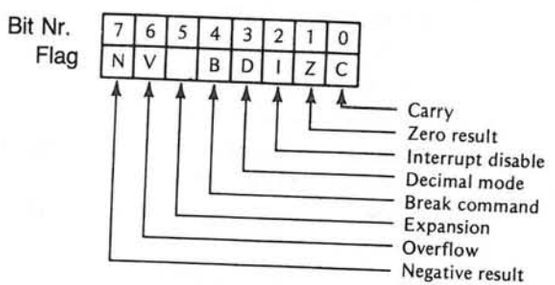


Abb. 1-9 Status-Register

EIN BASIC-ÜBERBLICK

Im 2. Kapitel werden wir die Verwendung von Programmen in Maschinensprache einführen, die ihrerseits durch BASIC-Programme eingegeben werden. Die Programme in Maschinensprache fungieren daher als Unterprogramme (subroutines) des BASIC-Programms. Das Atari-BASIC besitzt eine Funktion, die den Übergang vom BASIC-Programm zu einem derartigen Maschinensprachen-Programm gestattet, die sog. USR-Funktion.

Beispiel:

```
120 X = USR (1000)
```

Gehe zum Maschinen-Unterprogramm, das bei der Adresse 1000 anfängt.

Der Hauptteil dieses Kapitels dient dem Überblick über einige BASIC-Anweisungen, Befehle, Funktionen und Schlüsselwörter, die der Atari 400 bzw. 800 verwendet. Besondere Aufmerksamkeit verdienen dabei die USR-Funktion sowie die Schlüsselworte POKE, PEEK und ADR. Reichen Ihre Kenntnisse darüber Ihrer Meinung nach aus, dann gehen Sie an das 2. Kapitel.
Das Atari-BASIC verwendet Anweisungen und Schlüsselworte, die sich aufgrund der graphischen und akustischen Möglichkeiten des Rechners von anderen BASIC-Versionen unterscheiden. Diese werden hier kurz beschrieben. Eine ausführliche Beschreibung finden Sie im Atari-BASIC-Handbuch.

Graphische Schlüsselworte

COLOR Dient in erster Linie der Wahl des entsprechenden Farbregisters, hängt aber in der Anwendung vom jeweiligen Graphikmodus ab.

DRAWTO Zieht eine Linie von einem durch PLOT gegebenen Punkt zu einem anderen gegebenen Punkt.

FILL Kein eigentliches Schlüsselwort, sondern eine Ein/Ausgabe-Operation, die einen Teil des Bildschirms zwischen gegebenen (PLOT) Punkten und Geraden mit einer angebenen Farbe ausfüllt.

GET Dient der Eingabe des Code-bytes für das Zeichen, welches auf dem Bildschirm an der Schreibstelle erscheint (cursor).

GRAPHICS Steuert den graphischen Modus.

LOCATE Speichert die Farbnummer, die einen besonderen Punkt auf dem Bildschirm in der angebenen Variablen steuert.

PLOT Bringt einen Punkt in Abhängigkeit von den angegebenen X, Y-Koordinaten auf den Schirm.

POSITION Bringt den cursor auf eine bestimmte Bildschirmstelle.

PUT Dient der Ausgabe von Daten auf dem Bildschirm.

SETCOLOR Lädt ein angebenes Register mit spezifizierten Farb- und Helligkeitsdaten.

Schlüsselworte für Akustik und Spiel

PADDLE Gibt den momentanen Zustand des spezifizierten Controllers (eine Zahl zwischen 1 und 228, die von der Position des Drehreglers - Paddle abhängt) an den Rechner.

PTRIG Gibt eine Zahl an den Rechner, die die Stellung des Druckknopfes auf einem Controller beschreibt (0 wenn der Knopf gedrückt ist, andernfalls 1).

SOUND Spielt eine Note in einer von 4 spezifizierten Klangfarben, wobei Tonhöhe, Verzerrung und Lautstärke angegeben wird.

STICK Gleiche Funktion wie PADDLE, jedoch für Steuerknüppel-Controller.

STRIG Gleiche Funktion wie PTRIG, jedoch für Steuerknüppel-Controller.

Spezielle Schlüsselworte

Vier Schlüsselworte sind für uns besonders wichtig: ADR, PEEK, POKE und USR. Diese ermöglichen nämlich den Übergang von BASIC-Programmen zu Programmen, die in Maschinensprache geschrieben sind, und umgekehrt.

ADR Diese Funktion liefert die Adresse der Speicherzelle, die eine spezifizierte Zeichenkette enthält. Die Kenntnis dieser Adresse ermöglicht es dem Programmierer, Daten an USR-Unterprogramme weiterzugeben. Zwei Schritte sind nötig, um die Adresse einer Matrix von Zahlenwerten zu erhalten. Man benutzt zunächst die DIM-Anweisung für eine Zeichenkette der Länge 1. In einem anderen Zweig des BASIC-Programms fragt man nach der Adresse der String-Variablen.

Beispiel:

```
200 DIM A$(1)
```

Zeichenkette A$

```
210 PRINT ADR(A$)
```

Die Adresse der Matrix ist um 1 größer als diejenige der Zeichenkette A$.
Bei Verwendung der ADR-Funktion ist allerdings Vorsicht geboten. Wenn Ihr Programm (das aus der Datenmatrix besteht) nicht richtig adressiert ist, kann der Rechner ohne Wiederkehr ins Nirwana (die falsche Stelle) entschwinden. Wenn das passiert, kann man ihn abstellen und nach ungefähr 5 Sekunden wieder in Betrieb nehmen. Inzwischen ist dann allerdings auch Ihr Programm über alle Berge. Also nochmal von vorn!

PEEK Diese Funktion erlaubt dem Benutzer, den Inhalt einer spezifizierten Speicherzelle nachzusehen. Er kann die so „sichtbare" Information in seinem BASIC-Programm verwenden.

Beispiele:

```
200 PRINT "LINKER RAND BEI"; PEEK (82)
```

Diese Speicherzelle enthält stets die Position, an der beim Beschriften jeder Zeile das erste Zeichen erscheint (gewöhnlich 2)

```
210 PRINT "RECHTER RAND BEI"; PEEK (83)
```

Diese Zelle enthält die Position des letzten Zeichens einer Schreibzeile (gewöhnlich 39).

Man kann sich jede Speicherzelle des Atari (Adresse dezimal gegeben) anschauen, ohne ihren Inhalt zu zerstören. Dies gilt sowohl für ROM-(Read Only Memory) als auch für RAM-Zellen (Random Access Memory). RAM ist ein pro-

grammierbarer Speicher, aus dem man lesen und in den man hineinschreiben kann. ROM ist ein Festwertspeicher; aus ihm kann nur gelesen werden. In Kapitel 2 werden wir diesen Befehl häufig verwenden, um von uns in Maschinensprache abgespeicherte Programme anzuschauen.

POKE Diese Funktion ist gewissermaßen das Gegenteil von PEEK. Man kann mit ihr Daten in RAM-Zellen bringen oder ändern. Wir werden sie ebenfalls in Kapitel 2 verwenden, um in Programme, die in Maschinensprache geschrieben sind, einzugreifen.

```
150 POKE 82,8   ← Ändere die Position des ersten Zeichens
                  pro Zeile in 8 um.
160 POKE 83,30  ← Ändere den rechten Zeilenrand in 30 um.
```

Dies hätte folgenden Effekt.

```
01234567890123456789012345678901234567890123456789
                DRUCK (ZEICHEN)
                NUR VON 8–30
```
← Schirmbildstellen (0–39)

POKE kann sowohl im direkten wie im indirekten Modus verwendet werden. Sie können, um es nochmals zu sagen, in den Festwertspeicher (ROM) keine Daten hineinschreiben. Aus ihm kann nur gelesen werden! Da eine POKE-Anweisung gespeicherte Daten ändert, muß bei ihrer Anwendung äußerste Sorgfalt walten. Werden fehlerhafte Daten gespeichert oder Daten falsch gespeichert, so kann dies verheerende Folgen haben. Wollen Sie POKE benutzen, dann nehmen Sie zur Sicherheit die Speicherbelegung (Atari memory map) in Anhang G des Atari BASIC Manuals zur Hand.

Ein Tip, um POKE-Fehler zu vermeiden: Schauen Sie erst nach, was in der Speicherzelle steht, in die Sie hineinschreiben wollen, und notieren Sie den Inhalt. Sollte dann Ihre POKE-Anweisung nicht das gewünschte Ergebnis zeitigen, können Sie den ursprünglichen Zustand wieder herstellen.

USR Diese Funktion ruft („calls") ein in 6502 Maschinensprache geschriebenes Unterprogramm auf und fährt mit dem nach der Ausführung erzielten Ergebnis fort. Sie wird in Kapitel 2 verwendet, um die Programme in Maschinensprache, die mit Hilfe der POKE-Anweisung gespeichert wurden, auszuführen.

Das Format ist:

```
USR(X,Y,Z,. . . .)
```

Willkürliche Eingabedaten (Variable, die vom Unterprogramm übernommen werden).

Eine ganze Zahl oder ein arithmetischer Ausdruck, dessen Ergebnis eine ganze Zahl ist. Es handelt sich um die (dezimale) Adresse, unter der das Programm in Maschinensprache beginnt

Definitionen

Zum Überblick haben wir die wichtigsten Definitionen aus dem Atari BASIC-Handbuch zusammengefaßt.

ARRAY Ein Feld zum Abspeichern von Daten zwecks späterer Verwendung.

ARRAY VARIABLE Name eines Feldes mit einem oder mehreren Elementen.

BASIC Eine einem englischen Satz ähnliche Anweisung, die dem Computer sagt, was er tun soll.

BREAK KEY Damit wird der Programmablauf unterbrochen.

CONCATENATION Aneinanderreihung (Verkettung) von zwei oder mehr Strings (Zeichenketten).

CONSTANT Zahl oder Zeichenkette ohne Variablenname.

DEFERRED PROGRAMS Programme, die zwecks späterer Verwendung mit Zeilennummern abgespeichert werden.

DIRECT PROGRAMS Programme, die sofort, d.h. unmittelbar nach Eingabe der Programmzeile, ausgeführt werden.

EXPRESSION Ausdruck, der aus einer beliebigen Kombination zulässiger Variabler, Konstanter, Operatoren und Funktionen bestehen kann, die gemeinsam verwendet werden.

FUNCTION Eine dem Rechner fest eingegebene Funktion (Standardfunktion), die dem Programmierer zur Verwendung in seinem Programm zur Verfügung steht.

KEYWORD Wortsymbole der Programmiersprache BASIC, die nicht anderweitig verwendet werden dürfen (auch Schlüsselworte).

LOGICAL LINE Jede numerierte Zeile eines BASIC-Programms; sie besteht aus ein bis drei Bildschirmzeilen, die mit RETURN beendet werden.

NESTED LOOP Geschachtelte Schleifen.

OPERATOR Symbole zur Ausführung von arithmetischen, vergleichenden oder logischen Operationen.

RETURN KEY Damit werden die einzelnen Programmzeilen eingegeben.

STRINGS Eine beliebige Zeichenkette, die durch Hochkommas (aus nur einem Strich bestehende Anführungszeichen) zusammengefaßt ist.

VARIABLES Name für eine Zahl oder Zeichenkette.

Eine vollständige Beschreibung des Atari-BASIC finden Sie im Atari BASIC Handbuch. Nach diesem kurzen Überblick lassen Sie uns Kapitel 2 angehen.

BASIC-Unterprogramme in Maschinensprache

Mit Hilfe der USR-Funktion können kleine Programme in Maschinensprache von einem BASIC-Programm aus aufgerufen und ausgeführt werden. Es ist auf diese Weise möglich, einen Teil des BASIC-Programms in Maschinensprache zu schreiben, d.h. in ein in Maschinensprache geschriebenes Unterprogramm zu „springen" und nach dessen Ausführung an der geeigneten Stelle des BASIC-Programms fortzufahren. Abb. 2-1 zeigt ein entsprechendes Flußdiagramm.

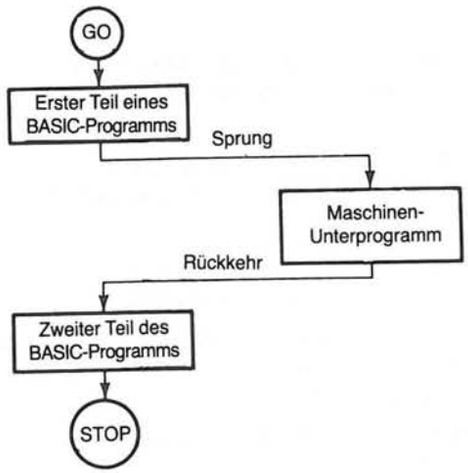


Abb. 2-1 Flußdiagramm mit Unterprogramm

Der Rechner kann nur Befehle im Dualzahlcode mit fester Stellenzahl verstehen. Diese Binärcodes nennt man Maschinensprache.

DUALZAHLMUSTER

Der Rechner interpretiert die „Bits" als 0 oder 1. Diese 0-1-Muster haben für den Rechner jeweils eine ganz bestimmte Bedeutung. Wir müssen sie daher erlernen, wenn wir direkt mit ihm verkehren wollen.

Sie sehen hier ein Beispiel für ein 8-Bit-Muster (welches der Rechner von Aufbau und Stellenzahl her verstehen kann):

0	1	0	1	1	0	1	1
AUS	EIN	AUS	EIN	EIN	AUS	EIN	EIN

Der Rechner erkennt in diesem Muster einen eindeutigen Zahlencode und reagiert auf diesen entweder mit einer entsprechenden Aktion oder verwendet ihn als speziellen Datenteil.
Ursprünglich wurde die Zentraleinheit des Computers von MOS Technology, Inc. vertrieben. Heute wird sie auch von zwei anderen Firmen (Synertek und Rockwell) gehandelt. Sie trägt die Bezeichnung Mikroprozessor 6502. Die Ausführung eines Befehls oder die Verarbeitung eines Zahlenwertes geschehen grundsätzlich über die Zentraleinheit, daher der Name.
Wie viele andere Mikroprozessoren versteht auch der 6502 (mithin auch der Atari) nur Befehle, die in Einheiten von 8 Binärstellen, sogenannten Bytes, codiert sind. Auf dem Weg zur Programmierung in Maschinensprache stellt daher das Erlernen des Umgangs mit binär codierter Information die größte Hürde dar.

0 1 1 1 1 1 0 1	← Block aus 8 Bits oder 1 Byte
1	← ein Bit
1 0	← zwei Bits (heute nicht viel wert)
1 1 0 1	← vier Bits (auch „nybble" genannt)
0 1 1 1 1 1 0 1	← acht Bits (allgemein Byte genannt)

Der Atari arbeitet mit Worten, die 8 Binärstellen lang sind, d.h. er versteht Worte von der Größe eines Byte. Sämtliche Befehle und Zahlenwerte müssen der Zentraleinheit in dieser Form übermittelt werden. Nachfolgend ist eine typische Anweisung dargestellt. Nach dem innerhalb eines Programms in Maschinensprache gegebenen Befehl lädt der Rechner den Akkumulator mit einem Datenbyte.

Lade Akkumulator aus Register L	
BUCHSTABENCODE (Abkürzung)	*BINÄRCODE*
LDA	10101001

Lassen Sie sich nicht von Fachausdrücken beeindrucken. Wie wir später sehen werden, ist der Akkumulator nichts anderes als eine besonderen Zwecken dienende Speicherzelle. Wir verwenden ihn an dieser Stelle lediglich zur Illustration eines Befehlsformats.
Der zum Atari gehörende BASIC Modul interpretiert automatisch BASIC-Anweisungen und Dezimalzahlen für den Rechner. Programmiert man daher in BASIC, so gibt man Zahlenwerte dezimal ein und erhält auch die Ergebnisse als Dezimalzahlen, obwohl der Rechner intern mit Binärzahlen arbeitet.

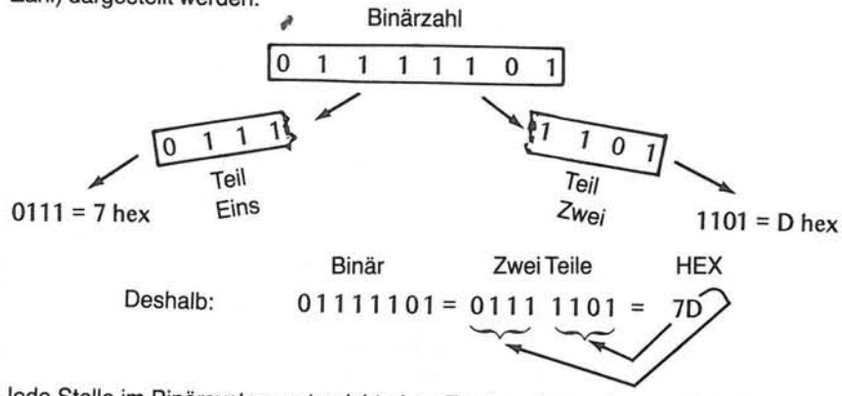
DIE HEXADEZIMALE NOTATION

Meist werden die Codes der Maschinensprache auch noch in einem 3. Zahlensystem angegeben, d.h. *hexadezimal* notiert. Das Hexadezimalsystem hat die Basis 16; es wird als „Kurzschrift" bei der Darstellung von Binärzahlen benutzt. Abb. 2-2 zeigt in Tabellenform äquivalente dezimal-, binär- und hexadezimal geschriebene Zahlen.

Dezimal	*Binär*	*Hexadezimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Abb. 2-2 Äquivalente dezimale, binäre und hexadezimale Zahlen

Wir werden diese Kurzschrift häufig einfach als Hex bezeichnen. Vier Binärstellen können durch eine Hexstelle dargestellt werden. Demnach kann unser 8-Bit-Befehl, durch Aufteilung des Bytes in zwei Hälften, durch eine 2-stellige hexadezimale Zahl (Hex-Zahl) dargestellt werden.



Jede Stelle im Binärsystem entspricht einer Zweierpotenz, ebenso wie jede Stelle im Zehnersystem einer Zehnerpotenz. Die Zwei heißt Basis des Binärsystems und die Zehn entsprechend Basis des Zehnersystems. Um die Bedeutung der einzelnen Stellen besser zu verstehen, schauen wir uns die Binärzahlen von 0000 bis 1111 einmal genauer an.

Binärstellen				*Dezimal Äquivalent*
2^3	2^2	2^1	2^0	
0	0	0	1	0+0+0+1 = 1
0	0	1	0	0+0+2+0 = 2
0	1	0	0	0+4+0+0 = 4
1	0	0	0	8+0+0+0 = 8

Durch geeignete Besetzung dieser Stellen mit 0 oder 1 können wir jeden Dezimalwert von 0 bis 16 oder jeden Hex-Wert von 0 bis F darstellen.

Beispiele:

0101 = $2^2 + 2^0$ = 4+1 = 5 dezimal und auch 5 hex
1001 = $2^3 + 2^0$ = 8+1 = 9 dezimal und auch 9 hex
1100 = $2^3 + 2^2$ = 8+4 = 12 dezimal entspricht C hex
1011 = $2^3 + 2^1 + 2^0$ = 8+2+1 = 11 dezimal entspricht B hex

Wir wollen nun genauer untersuchen, wie wir eine beliebige 8-Bit-Binärzahl durch zwei Hexstellen darstellen können. Wir haben bereits gesehen, daß die größte Hexstelle (F) der 4-Bit-Binärzahl 1111 entspricht. Die nächst höhere Binärzahl ist 10000. Die 1 steht an der Stelle der Potenz $2^4 = 16$. Wir haben demnach nichts außer einer 16. Diese können wir durch den Hex-Wert 10 ausdrücken, d.h. 16 ohne Einerstellenwert. Es gibt eine direkte Beziehung zwischen den oberen 4 Bits einer 8-Bit Binärzahl und der Ziffer der 16-Stelle einer Hexzahl.

Binärstellen				Hex Wert
2^7	2^6	2^5	2^4	16^1
0	0	0	1	1
0	0	1	0	2
0	1	0	0	4
1	0	0	0	8

$2^4 = 16$
$2^5 = 2*16 = 32$
$2^6 = 4*16 = 64$
$2^7 = 8*16 = 128$

Sehen Sie sich nun die Binärstellenwerte der vollständigen 8-Bit-Zahl an.

Binärstellen								Dezimal Äquivalent	Hex Äquivalent
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
0	0	0	0	0	0	0	1	0+0+0+0+0+0+0+1 = 1	1
0	0	0	0	0	0	1	0	0+0+0+0+0+0+2+0 = 2	2
0	0	0	0	0	1	0	0	0+0+0+0+0+4+0+0 = 4	4
0	0	0	0	1	0	0	0	0+0+0+0+8+0+0+0 = 8	8
0	0	0	1	0	0	0	0	0+0+0+16+0+0+0+0 = 16	10
0	0	1	0	0	0	0	0	0+0+32+0+0+0+0+0 = 32	20
0	1	0	0	0	0	0	0	0+64+0+0+0+0+0+0 = 64	40
1	0	0	0	0	0	0	0	128+0+0+0+0+0+0+0 = 128	80

Abb. 2-3 Acht-Bit Konvertierung

Verwendet man alle 8 Bits, so kann man damit jede Dezimalzahl von 0 bis 255 oder jeden Hex-Wert von 0 bis FF darstellen. Unterteilen wir eine 8-stellige Binärzahl in zwei Teile von 4 Bit Länge, so kann jede Hälfte durch eine Hexziffer dargestellt werden.

Beispiele:

Binär	01101101	64+32+8+4+1 = 109 dezimal
Binär geteilt	0110 1101	
Hex	6 D	6*16+13 = 109 dezimal
Binär	11000101	128+64+4+1 = 197 dezimal
Binär geteilt	1100 0101	
Hex	C 5	12*16+5 = 197 dezimal
Binär	10101100	128+32+8+4 = 172 dezimal
Binär geteilt	1010 1100	
Hex	A C	10*16+12 = 172 dezimal

UMWANDLUNG (KONVERTIERUNG) VON HEXADEZIMALZAHLEN IN DEZIMALZAHLEN

Einstellige Hex-Codes lassen sich sehr leicht in Dezimalzahlen umwandeln. Sollten Sie die entsprechenden Werte noch nicht im Kopf haben, schauen Sie nochmals die folgende Tabelle an.

Hexadezimal	Dezimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Abb. 2-4 Konvertierung von Hexadezimalzahlen in Dezimalzahlen

Ein zweistelliger Hex-Code läßt sich ebenfalls leicht in eine Dezimalzahl umwandeln, wenn man sich die beiden folgenden Tatsachen merkt:

1. Die zu jedem einstelligen Hex-Code gehörende Dezimalzahl (laut obiger Tabelle).
2. Den jedem einstelligen Hex-Code zugeordneten Stellenwert.

Die Stellenwerte sind im Dezimalsystem entsprechende Potenzen von 10. Die letzte Stelle einer ganzen Zahl im Dezimalsystem ist ein Vielfaches der Potenz 10^0, die sogenannte Einerstelle ($10^0 = 1$). Die nächste Stelle ist ein Vielfaches der Potenz 10^1, die sogenannte Zehnerstelle ($10^1 = 10$).

Beispiele:

$$89 = (8 \times 10^1) + (9 \times 10^0)$$

oder

$$(8 \times 10) + (9 \times 1) = 80 + 9$$

Ziffer Stellenwert Ziffer Stellenwert

$$23 = (2 \times 10^1) + (3 \times 10^0)$$

oder

$$(2 \times 10) + (3 \times 1) = 20 + 3$$

Die hexadezimalen Stellenwerte sind Potenzen von 16 (hexadezimal = 6 und 10). Die letzte Stelle einer ganzen, hexadezimalen Zahl ist ein Vielfaches von 16^0, die sogenannte Einerstelle ($16^0 = 1$). Die folgende Stelle ist entsprechend ein Vielfaches von 16^1 und wird Sechzehnerstelle genannt ($16^1 = 16$).

Beispiele:

23 hex = $(2 \times 16^1) + (3 \times 16^0)$

oder

$(2 \times 16) + (3 \times 1) = 32 + 3$ dezimal

$= 35$ dezimal

Ziffer — Stellenwert — Ziffer — Stellenwert

89 hex = $(8 \times 16^1) + (9 \times 16^0)$

oder

$(8 \times 16) + (9 \times 1) = 128 + 9$ dezimal

$= 137$ dezimal

A9 hex = $(10 \times 16^1) + (9 \times 16^0)$

oder

$(10 \times 16) + (9 \times 1) = 160 + 9$ dezimal

$= 169$ dezimal

Versuchen Sie nun einmal, die zu den folgenden Hex-Codes gehörenden Dezimalzahlen einzutragen.

Hex Code	*Dezimal Äquivalent*
68	
18	
A9	
3F	
69	
41	
8D	
00	
18	
60	

Abb. 2-5 Übung zur Hex-Code Konvertierung

Wenn Sie die in Abb. 2-5 angegebenen Hex-Codes richtig konvertiert haben, können Sie sich an Ihr erstes Programm in Maschinensprache wagen. Die Codes sind Maschinensprachenbefehle und Daten. Denken Sie jedoch daran, daß Sie diese von einem BASIC-Programm aus eingeben, weshalb Sie für die Eingabe die Dezimalform nehmen müssen. Prüfen Sie nun Ihr Ergebnis anhand von Abb. 2-7.
Bevor wir uns dem Unterprogramm in Maschinensprache zuwenden, schauen wir uns zunächst das für die Eingabe und Ausführung des Unterprogramms benutzte BASIC-Programm an.

BASIC PROGRAMM – ADDIERE ZWEI ZAHLEN

```
100  REM INITIALIZE STORAGE ADDRESS
110  CLR: DIM E$(1), E(10)

120  REM POKE IN SUBROUTINE
130  FOR Y = 1 TO 10
140       INPUT N
150       POKE ADR(E$)+Y,N
160  NEXT Y

170  REM CALL SUBROUTINE
180  X=USR(ADR(E$)+1)

190  REM PRINT RESULTS
200  GR.0:PRINT "THE SUM IS ";
210  PRINT PEEK(6144)
220  END
```

Man beachte Zeile 110. Nach der DIM-Anweisung für E$ muß ein Feld (in diesem Programm E) für die im Maschinenprogramm verwendeten Codes dimensioniert werden. Damit werden die notwendigen Speicherplätze reserviert, um die in der FOR-NEXT-Schleife in den Zeilen 130-160 durch die POKE-Anweisung eingegebenen Codes aufzunehmen.
Abb. 2-6 zeigt ein Flußdiagramm für das BASIC-Programm und dessen Unterprogramm in Maschinensprache.

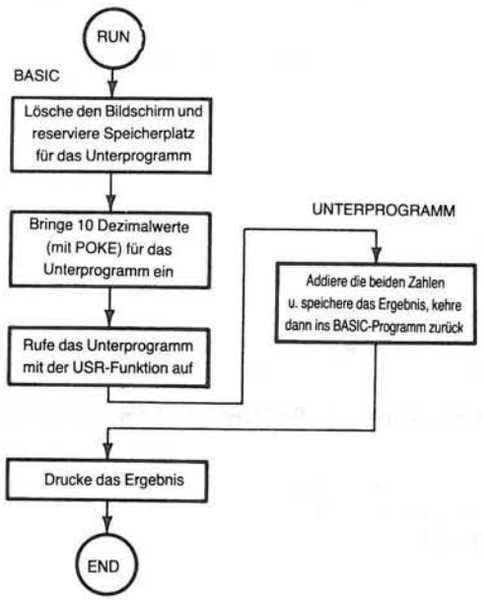


Abb. 2-6 Flußdiagramm des Additions-Unterprogramms

Geben Sie RUN ein, so wird das Maschinenprogramm in den Speicher geschrieben. Geben Sie jedesmal mit der Eingabe-Schleife die Dezimalzahl ein, die dem auf das Fragezeichen (input prompt) folgenden Hex-Code entspricht. Bei den Eingaben handelt es sich um die Werte, die Sie in Abb. 2-5 eingetragen haben.

Hex Code	*Dezimal Äquivalent*
68	104
18	24
A9	169
3F	63 ←
69	105
41	65 ←
8D	141
00	0
18	24
60	96

Fügen Sie diese zwei Dezimalzahlen hinzu

Abb. 2-7 Dezimal-Äquivalente von Hex-Codes

Nach der Eingabe aller 10 Codes wird das Maschinenprogramm ausgeführt, und nach dessen Abarbeitung das Ergebnis ausgegeben. Geben Sie das Programm ein, und geben Sie RUN. So sollte es auf dem Bildschirm nach der neunten Eingabe, unmittelbar vor Ihrem RETURN-Befehl, aussehen.

```
?104
?24
?169
?63
?105
?65
?141
?0
?24
?96          ← Die 10. Eingabe
```

Geben Sie zum zehnten Mal RETURN, wird der Bildschirm gelöscht und Ihr Ergebnis erscheint.

```
THE SUM IS 128
READY
■
```


Herzlichen Glückwunsch! Sie haben soeben Ihr erstes Programm in Maschinensprache bewältigt. Bei der Bearbeitung von Zeile 180 durch Ihr BASIC-Programm sorgt die USR-Funktion für die Ausführung des Programms in Maschinen-Code.
In Abb. 2 - 8 finden sie jeden Hex-Code, den entsprechenden Buchstabencode der Assemblersprache sowie eine Angabe darüber, was er bewirkt.

Hex-Code	*Buchstaben-code*	*Erklärung*
68	PLA	Nimm ein Byte aus dem Stapelspeicher und bringe es in den Akkumulator
18	CLC	Lösche das CARRY Flag-Bit (Zustandsbit)
A9	LDA	Lade den Akku mit dem unmittelbar folgenden Wert
3F		Geladener Wert (3F hex = 63 dezimal)
69	ADC	Addiere den nachfolgenden Hex-Wert zu dem im Akku befindlichen Wert
41		Addierter Wert (41 hex = 65 dezimal)
8D	STA	Speichere den Wert aus dem Akku unter der nachfolgenden Adresse
00		Niederwertiges Byte im Speicher
18		Höherwertiges Byte im Speicher
60	RTS	Kehre vom Unterprogramm zurück

Abb. 2-8 Maschinen-Unterprogramm

Beachten Sie, daß einige Codes aus einzelnen Bytes (68, 18, 60), einige aus zwei Bytes (A9 und 3F, 69 und 41), und einer aus drei Bytes bestehen (8D und 00 und 18). Daten und Speicheradressen vervollständigen den entsprechenden Befehl zur Bildung einer Multi-Byte-Anweisung.

ARBEITSWEISE DES PROGRAMMS IN MASCHINENSPRACHE

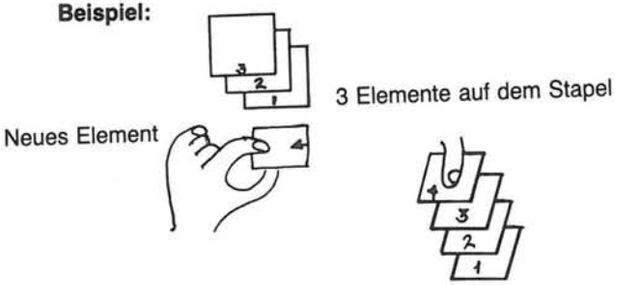
Bislang wissen wir kaum etwas über die einzelnen Codes in Maschinensprache. Sie erscheinen uns derzeit noch als Buch mit sieben Siegeln; für den Rechner jedoch hat jeder Code eine genaue Bedeutung. Wir wollen sie in folgender Gruppierung nacheinander besprechen.

1. 68
2. 18

3. A9 3F
4. 69 41
5. 8D 00 18
6. 60

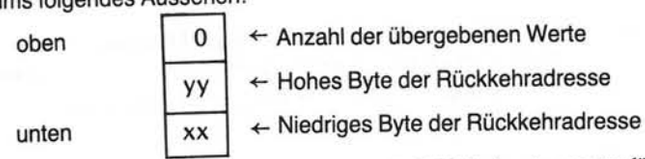
1. Der erste Befehl (68) in Maschinensprache bewirkt den Transport eines Datenbytes aus dem Stapelspeicher in den Akkumulator.
Der Stapelspeicher ist ein besonderer Teil des gesamten Speichers, in dem Zahlen und Adressen in ganz spezieller Form für den späteren Gebrauch abgelegt werden. Jeder neue Datenwert gelangt auf den obersten Platz des Stapels. Alle anderen Werte werden um eins „nach unten" gerückt, wenn ein neuer Wert auf den ersten Platz kommt.

Beispiel:

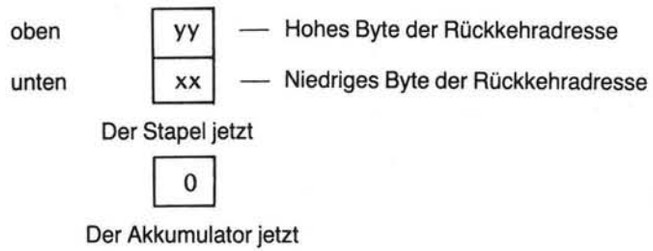


Wenn eine USR-Funktion von einem BASIC-Programm aus ein Programm in Maschinensprache „aufruft", wird die BASIC-Adresse, zu der das Maschinenprogramm nach getaner Arbeit zurückkehren soll, in zwei getrennten Bytes gestapelt – der niedrige Adressenteil zuerst, dann der höhere.

Danach bzw. „darauf" stapelt der Rechner alle Variablen, die an das Unterprogramm in Maschinensprache übergeben werden. Der letzte Wert, der dabei gestapelt wird, ist die Anzahl der übergebenen Variablen. In unserem Beispiel fand keine Übergabe irgendwelcher Variabler statt. Daher hatte der Inhalt des Stapels beim Aufruf des Unterprogramms folgendes Aussehen:



Um nach Erledigung des Unterprogramms die Rückkehradresse verfügbar zu haben, mußten wir die 0 vom Stapel herunternehmen. Dies geschah mit dem Befehl (68). Dieser Befehl brachte die 0 in den Akkumulator. Damit steht uns jetzt die Rückkehradresse zuoberst zur Verfügung.



Die im Akkumulator befindliche Zahl (0) wird nicht weiter verwendet, sie mußte jedoch vom Stapel genommen werden, wollte man an die Rückkehradresse gelangen.

2. Der zweite Befehl (18) löscht das Carry-Bit im Status Register (setzt es auf 0). Der Addierbefehl (vierter Befehl) addiert automatisch das Carry-Bit beim Summieren zweier Zahlen. Da Sie dies aber nicht wollen, wird mittels dieses Befehls dafür gesorgt, daß das Bit 0 ist. Später werden Sie gelegentlich einige Mehrfachbyte-Additionen mit größeren Zahlen ausführen. Dabei muß man, wie wir sehen werden, das Carry-Bit verwenden, wenn man alle außer dem letzten signifikanten Byte addieren will.

3. Der dritte Befehl besteht aus zwei Hex-Codes (oder zwei Bytes):
A9 3F
A9 ist die hexadezimale Darstellung des Befehls „Lade den Akkumulator mit dem unmittelbar folgenden Wert". 3F ist die Zahl, die geladen wird.

4. Der vierte Befehl besteht aus zwei Hex-Codes (zwei Bytes):
69 41
69 ist die hexadezimale Darstellung des Befehls „Addiere die nachfolgende Zahl zu der im Akkumulator befindlichen Zahl". Der Akku enthält hinterher das Ergebnis. Der Hex-Wert 41 ist die addierte Zahl.

5. Der fünfte Befehl besteht aus drei Hex-Codes (drei Bytes):
8D 00 18
8D ist die hexadezimale Darstellung der Anweisung „Speichere den im Akkumulator befindlichen Wert unter der nachfolgenden Adresse ab." Die verwendete Adresse ist 1800 Hex (6144 dezimal). Man beachte hier, daß die Adresse in zwei Bytes abgespeichert werden muß, da sie für die Darstellung in einem Byte zu groß ist. Daher werden zwei Hex-Codes verwendet. Man merke sich auch die Reihenfolge, in der das höhere und das niedrigere Adressbyte stehen (18 bzw. 0). Das niedrige Adressbyte kommt zuerst, dann das höhere. Diese Vereinbarung gilt bei Befehlen mit einem Zwei-Byte-Adressteil.
6. Der letzte Befehl besteht aus einem einzigen Hex-Code (ein Byte):
60
60 ist die hexadezimale Darstellung des Befehls „Kehre aus dem Unterprogramm zurück". Dieser Befehl läßt den Rechner dorthin zurückkehren, woher er gekommen ist.

Der Rechner holt sich die beiden obersten Bytes vom Stapel, die die Stelle enthalten, zu der er im BASIC-Programm gehen muß, um die nächste BASIC-Anweisung entgegenzunehmen.

Ein in Maschinensprache geschriebenes Programm wird, ebenso wie ein BASIC-Programm nach Zeilennummern, sequentiell in der Reihenfolge der Adressen abgearbeitet. Es gibt Ausnahmen, wie die BASIC-Anweisungen BASIC GOTO und GOSUB; Programme in Maschinensprache können, wie wir später sehen werden, entsprechende Manöver mittels JUMP- und BRANCH-Anweisungen ausführen.
Wenn auch dieses Buch vornehmlich auf das Programmieren in Assembler (die gebräuchliche Abkürzung für Assembler-Programmiersprache) und den Atari Assembler Modul ausgerichtet ist, sollte man sich deutlich vor Augen halten, daß der Rechner selbst nur mit der Maschinensprache arbeitet. Die Assemblersprache ist, wie wir später sehen werden, im Grunde nichts weiter als eine Methode, mit der man sich die Maschinencodes besser merken kann und welche deren Eingabe wesentlich erleichtert.
Im ersten Programm haben wir sechs Maschinenbefehle verwendet. Da solche Anweisungen in Maschinensprache codierte Zahlen sind, kann man sie sich nur schwer merken. In der Assemblersprache wird nun für jede Anweisung ein Buchstabencode verwendet, der eine Abkürzung der Befehlsbedeutung darstellt.

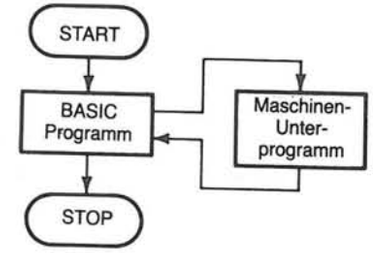
BISHER VERWENDETE BEFEHLE

Maschinen-sprache	*Assembler*	*Funktion*
68	PLA	Nimm ein Byte aus dem Stapelspeicher und bringe es in den Akkumulator
18	CLC	Lösche das Carry Flag-Bit
A9	LDA	Lade den Akku mit dem unmittelbar folgenden Hex-Wert
69	ADC	Addiere den nachfolgenden Hex-Wert (mit carry) zu dem im Akku befindlichen Wert
8D	STA	Speichere den Wert aus dem Akku unter der nachfolgenden Adresse
60	RTS	Kehre vom Unterprogramm zurück

ZUSAMMENFASSUNG

In diesem Kapitel haben Sie gelernt:

- Wie man von einem BASIC-Programm aus mittels der URS-Funktion ein Unterprogramm in Maschinensprache aufruft;



- Daß die Befehle in Maschinensprache Binärzahlen sind, die in hexadezimaler Form eingegeben werden können;
- Daß einige Anweisungen ein Byte, einige zwei Bytes und andere schließlich drei Bytes benötigen;
- Wie der Stapelspeicher zum Zwischenspeichern von Zahlen und Adressen verwendet wird;
- Wie man Zahlen aus dem Stapelspeicher holt;
- Daß jedes Unterprogramm einer Rückkehranweisung vom Unterprogramm (Return from Subroutine) bedarf, um die Ablaufsteuerung wieder dem Hauptprogramm zu übergeben, und schließlich
- Die folgenden Anweisungen in Maschinensprache:

68	PLA	Nimm ein Byte aus dem Stapelspeicher und bringe es in den Akkumulator
A9	LDA	Lade den Akku mit dem unmittelbar folgenden Hex-Wert
69	ADC	Addiere den nachfolgenden Hex-Wert (mit carry) zu dem im Akku befindlichen Wert
8D	STA	Speichere den Wert aus dem Akku unter der nachfolgenden Adresse
60	RTS	Kehre vom Unterprogramm zurück
18	CLC	Lösche das Carry-Flag-Bit

ÜBUNGEN

1. Hexadezimalcodes sind Abkürzungen, um _____ Codes aufzuschreiben, die der Rechner versteht.
   (dezimale, binäre)
2. Welche Zahlendarstellung wird in BASIC-Programmen verwendet?
   _____
   (binär, hex, dezimal).

3. Geben Sie die zu den folgenden Binärzahlen gehörenden Hexadezimal- bzw. Dezimalzahlen an.
   a) 1101 = _____ hex = _____ dezimal
   b) 0110 = _____ hex = _____ dezimal
   c) 10101001 = _____ hex = _____ dezimal
4. Verwenden Sie innerhalb eines BASIC-Programms ein Unterprogramm in Maschinensprache, dann geben Sie die Befehle in _____ Codierung, (hex, dezimal)
   und der BASIC-Interpretierer konvertiert sie in _____ Form. (hex, dezimal)
5. Wie lautet die aus drei Buchstaben bestehende Abkürzung für die Funktion, mit der man ein Maschinensprachenprogramm in Atari-BASIC aufruft? _____
6. Wenn Sie von einem BASIC-Programm aus ein Unterprogramm aufrufen wollen und zu diesem Zweck die Variable E$ folgendermaßen dimensioniert haben: 1(100 DIM E$ (1)), wie muß dann die BASIC-Zeile aussehen, die dieses Unterprogramm aufruft?
   500 _____
7. Wie heißt die letzte Anweisung, die ein in Maschinensprache geschriebenes Programm ausführen muß, um zum BASIC-Programm zurückzukehren?
   _____
8. Wollen Sie die Dezimalzahl 72, die sich im Akkumulator befindet, unter der hexadezimalen Adresse 1925 abspeichern, dann besteht der Befehl zum Abspeichern dieses Wertes in Maschinensprache aus einem 3-Byte-Code. Ergänzen Sie die Leerstellen.
   8 D
   _____
   _____
9. Welcher Hex-Wert wird in Aufgabe 8 unter 1925 abgelegt?
   _____

ANTWORTEN

1. Binär
2. Dezimal
3. (a) 1101 = D hex = 13 dezimal
   (b) 0110 = 6 hex = 6 dezimal
   (c) 10101001 = A 9 hex = 169 dezimal
4. Hex
   Dezimal

5. USR
6. 500 X = USR (ADR(E$)+1)
   ↖ Für X kann auch eine andere Variable stehen
7. 60 oder RTS oder Rückkehr vom Unterprogramm
8. 8 D
   25 (Vergessen Sie nicht die umgekehrte Reihenfolge)
   19
9. 48 (4*16+8=72)

Speicherverwendung

Werden Programme in Maschinensprache und BASIC gemeinsam benutzt, so müssen für jede Sprache getrennte Speicher verwendet werden. Der Atari BASIC-Interpretierer (im BASIC ROM Modul) besorgt von sich aus die nötigen Speicherplätze für den BASIC-Teil Ihres Programms. Sie müssen sicherstellen, daß der in Maschinensprache geschriebene Teil Ihres Programms an anderer Stelle des Speichers abgelegt wird.

ATARI SPEICHERBELEGUNG

Ein Teil der 65536 Speicherplätze des Rechners wird vom Atari-Betriebssystem (8K des Festwertspeichers ROM) und den ROM-Modulen (wie der BASIC- oder Assembler Modul), die man in die Console des Rechners einschieben kann, verwendet. Es ist nicht möglich, Programme in Maschinensprache mit POKE dorthin zu bringen. Das Betriebssystem und die ROM-Module verwenden auch Teile des programmierbaren Speichers RAM. In diese darf man keinerlei Daten mittels POKE hineinschreiben, da man dabei sein eigenes Programm zerstören oder anderweitig Unheil anrichten könnte.
Abb. 3-1 zeigt die Belegung des Atari-Speichers. Bei näherer Betrachtung sehen Sie FREE RAM, die frei programmierbaren Speicherteile. Diese können Sie risikolos für Ihre Programme verwenden. Wie jedoch bereits gesagt, müssen Sie die BASIC- und die Maschinensprachen-Programmteile im Speicher trennen.

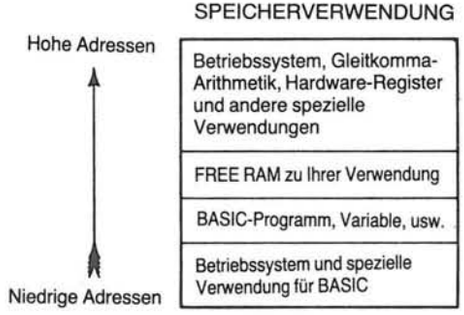
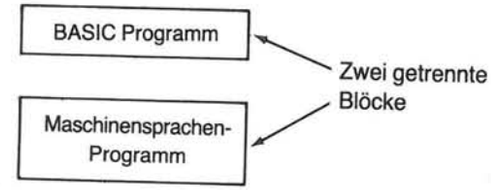


Abb. 3-1 Speicherbelegung



Die von uns bislang verwendete Technik war die DIMensionierung einer String-Variablen E$ auf (1). Dadurch wird die Speicherzelle reserviert, die unmittelbar auf die vom BASIC-Programm belegten Speicherzellen folgt. Die Adresse dieser Zelle wird sich natürlich in Abhängigkeit von der jeweiligen Programmlänge von BASIC-Programm zu BASIC-Programm ändern, aber der für den in Maschinensprache geschriebenen Programmteil verwendete Speicherplatz kommt immer nach dem für BASIC verbrauchten Platz. Wir wollen dies anhand von drei kurzen BASIC-Programmen verschiedener Länge zeigen. Sie enden alle mit der Ausgabe der Adresse der für E$ reservierten Speicherzelle.

DEMO PROGRAM #1

```
10    CLR: DIM E$(1)
20    PRINT ADR(E$)
```

Geben Sie das Demo-Programm # 1 ein und lassen Sie es laufen (RUN). Sie sehen dann, daß für E$ der Speicherplatz mit der Adresse 11182 reserviert ist.

```
RUN
11182

READY
■
```

DEMO PROGRAM #2

```
10    CLR: DIM E$(1)
20    FOR X = 1 TO 10   ← Eine sehr kleine Zeitverzögerung
30    NEXT X
40    PRINT ADR(E$)
```

Geben Sie das Demo-Programm # 2 ein und lassen Sie es laufen. Sie werden sehen, daß der für E$ reservierte Speicherplatz nun die Adresse 11221 besitzt. Da das Demo-Programm # 2 länger ist als das Demo-Programm # 1, hat der für E$ reservierte Speicherplatz eine höhere Adresse.

```
RUN
11221

READY
■
```

DEMO PROGRAM #3

```
10    CLR: DIM E$(1)
20    FOR X = 1 TO 10
30        PRINT X;
40    NEXT X
50    PRINT
60    PRINT ADR(E$)
```

Geben Sie das Demo-Programm # 3 ein und lassen Sie es laufen. Sie werden sehen, daß die für E$ reservierte Adresse noch höher ist, da das Demo-Programm # 3 länger ist als die beiden anderen.

```
.
.
.
RUN
12345678910
11235

READY
■
```

Wir haben die für E$ reservierte Speicherzelle verwendet, um für unser in Maschinensprache geschriebenes Programm einen geeigneten RAM-Speicherplatz zu finden, der nicht mit dem für unsere BASIC-Programme reservierten kollidiert. Wir haben es in den Speicherteil gebracht, der unmittelbar nach der für E$ reservierten Speicherzelle beginnt. Wir werden diese Methode auch fürderhin anwenden.

WIE BASIC DAS MASCHINENPROGRAMM AUFRUFT

Man wird sich natürlich fragen, woher das BASIC-Programm weiß, wo es das Maschinen-Programm findet. Lassen Sie uns noch einmal zu dem BASIC-Programm in Kapitel 2 zurückkehren, welches zwei Zahlen addierte.

Erster Teil

```
100   REM INITIALIZE STORAGE ADDRESS
110   CLR: DIM E$(1), E(10)
```

Reserviere einen Speicherplatz für E$

Reserviere 10 Plätze für das Unterprogramm in Maschinensprache

Zweiter Teil

```
120   REM POKE IN SUBROUTINE
130   FOR Y = 1 TO 10
140       INPUT N
150       POKE ADR(E$)+Y,N
160   NEXT Y
```

Bringe (mit POKE) 10 Hex-Codes in die aufeinanderfolgenden Plätze (E$+1 bis E$+10)

Dritter Teil

```
170   REM CALL SUBROUTINE
180   X = USR(ADR(E$)+1)
```

Springe zur Adresse (E$) + 1, unter der das Unterprogramm beginnt

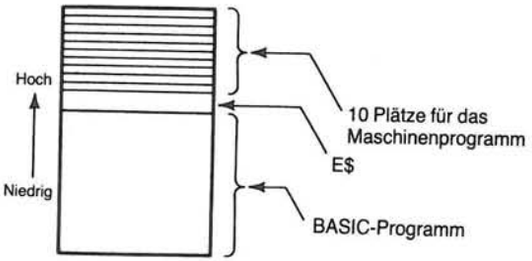


Abb. 3-2 Speicherplatz für das Maschinenprogramm

Im vierten Teil des BASIC-Programms wird das Ergebnis des Maschinenprogramms gedruckt.

Vierter Teil

```
190   REM PRINT RESULTS
200   GR.O: PRINT "THE SUM IS";
210   PRINT PEEK (6144)
220   END
```

← Drucke die Marke

Drucke das Ergebnis, das das Maschinenprogramm in der Speicherzelle 6144 (1800 hex) abgelegt hat

Der Rechner hat für E$ einen Platz in einem Teil des Speichers reserviert, der mit Ihrem BASIC-Programm nichts zu tun hat. In den Zeilen 130-160 bringt das Programm mittels POKE die Codes der Maschinensprache in den Speicher, unmittelbar oberhalb des für E$ reservierten Platzes. In Zeile 180 ruft das BASIC-Programm das Unterprogramm mit Hilfe der USR-Funktion auf. Diese Funktion gibt die Adresse an, unter der das Unterprogramm abgelegt ist. Sie arbeitet daher wie die BASIC-Anweisung GOSUB.

```
X = USR(ADR(E$)+1)
```

Aufruf des Unterprogramms

Das Unterprogramm fängt unter dieser Adresse an

Die Hex-Codes der Maschinensprache sind im Speicher folgendermaßen abgelegt:

Speicher-Adresse	*Gespeicherter Hex-Code*
E$+1	68
E$+2	18
E$+3	A9
E$+4	3F
E$+5	69
E$+6	41
E$+7	8D
E$+8	00
E$+9	18
E$+10	60

Abb. 3-3 Hex-Code Speicherung

Der Atari kann mit der USR-Funktion auch einen oder mehrere Werte an ein Unterprogramm übergeben.

Beispiel:

```
100 A=5
110 X = USR(ADR(E$)+1,A)
```

Übergebe den Wert von A an das Maschinenprogramm

Der Wert von A wird entsprechend der in Kapitel 2 gegebenen Beschreibung gestapelt, und zwar als eine zwei Byte umfassende Hexadezimalzahl.

Schauen wir uns den Vorgang nochmals an:

1. Die Adresse, zu der Ihr BASIC-Programm zurückkehren soll, wird gestapelt.

yy
xx

← Hohes Byte der Rückkehradresse

← Niedriges Byte der Rückkehradresse

2. Der Wert von A wird danach gestapelt. Ist A = 5, dann

00
05
yy
xx

← Höherwertiges Byte von A

← Niederwertiges Byte von A

← Hohes Byte der Rückkehradresse

← Niedriges Byte der Rückkehradresse

3. Die Anzahl der übergebenen Werte wird danach gestapelt.

01
00
05
yy
xx

← Anzahl der übergebenen Werte

← Höherwertiges Byte von A

← Niederwertiges Byte von A

← Hohes Byte der Rückkehradresse

← Niedriges Byte der Rückkehradresse

Das Maschinenprogramm kann sich dann die Werte vom Stapel herunterholen und verwenden.

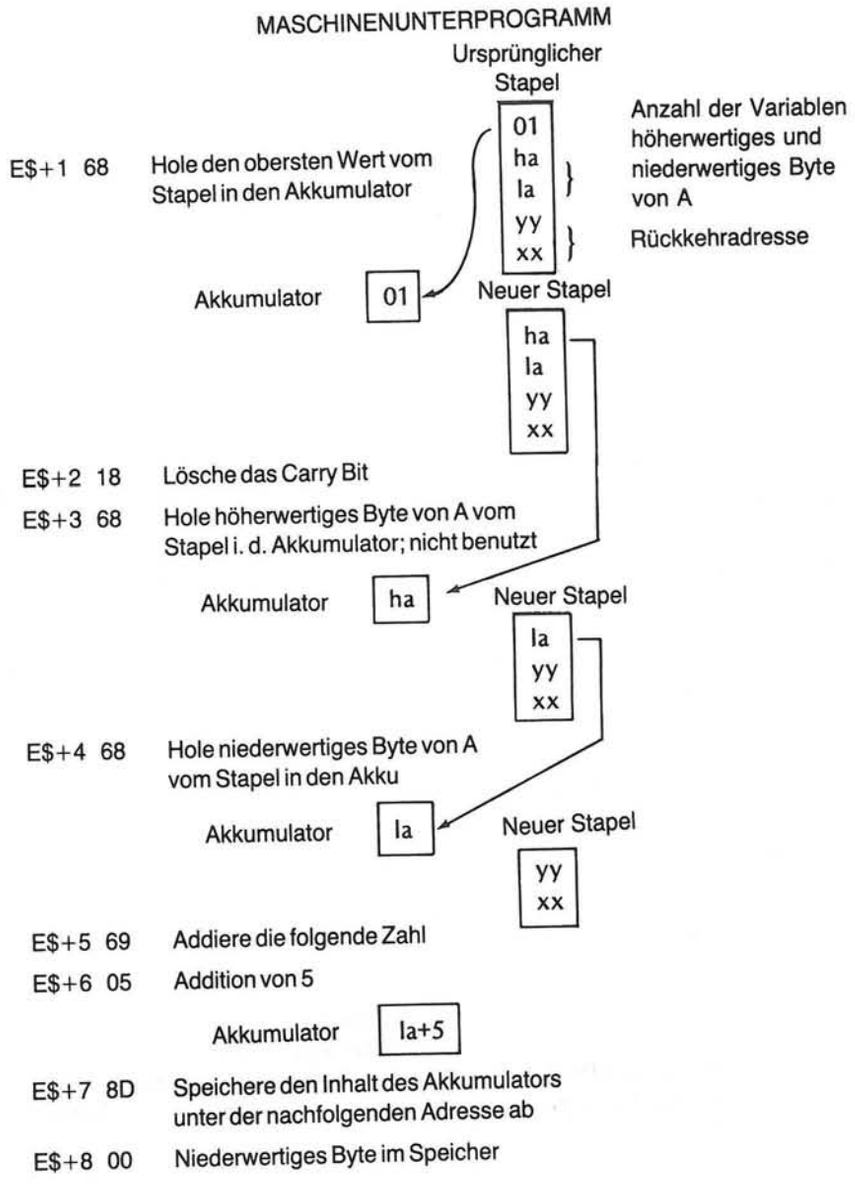
DIE ÜBERGABE VON VARIABLEN AN DAS UNTERPROGRAMM IN MASCHINENSPRACHE

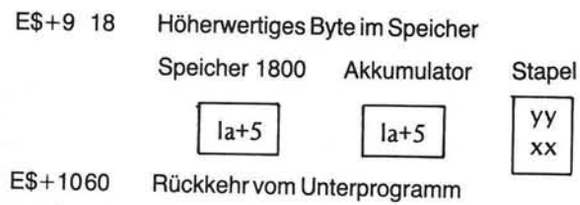
Mit dem folgenden Additionsprogramm sollen die entsprechenden Methoden erläutert werden. Es ermöglicht Ihnen, einen Wert einzugeben, der an das Unterprogramm weitergereicht wird. Das Unterprogramm addiert den Wert 5 zu dem von Ihnen eingegebenen Wert. Das Ergebnis wird so abgelegt, daß es vom BASIC-Programm wieder gefunden werden kann.

BASIC PROGRAMM – EINE VARIABLE UEBERGEBEN

```
100  CLR: DIME$(1), E(10)

200  FOR Y = 1 TO 10
210      INPUT N                    Eingabe des Maschinenprogramms
220      POKE ADR(E$)+Y,N           (10 Bytes)
230  NEXT Y

300  GR.O: PRINT "INPUT A"          Eingabe einer ganzen, positiven
310  INPUT A                        Dezimalzahl, kleiner als 251
320  X = USR(ADR(E$)+1,A)           Aufruf des Unterprogramms
330  PRINT PEEK(6144)               Drucken der Summe Ihrer Zahl und 5
340  END
```

Bei der Ausführung des BASIC-Programms wird der Wert von A gestapelt, und das Unterprogramm nimmt ihn dann zur weiteren Verwendung vom Stapel herunter. Das Ergebnis wird bei Rückkehr zum BASIC-Programm ausgegeben.
Schauen wir uns den Inhalt des Akkumulators, des Speichers und des Stapelspeichers beim Ablauf des folgenden Unterprogramms in Maschinensprache an:





PROGRAMM MIT EINER VARIABLEN

Die Eingabe und Verwendung des Programms geschieht in folgenden Schritten:

1. Eingabe BASIC PROGRAM-ONE VARIABLE PASSED.
2. Starten des BASIC-Programms über RUN und Eingeben der folgenden Hex-Codes über ENTER (jeweils einen Code als Antwort auf das ?).

```
?104
?24
?104
?104
?105
?5
?141
?0
?24
?96
```

3. Nach dem letzten Eintrag wird der Bildschirm gelöscht und die Aufforderung INPUT erscheint.

```
INPUT A?■
```

Versuchen Sie, 95 zu tippen, und drücken Sie die RETURN-Taste.

```
INPUT A?95
100

READY
■
```

95+5 = 100

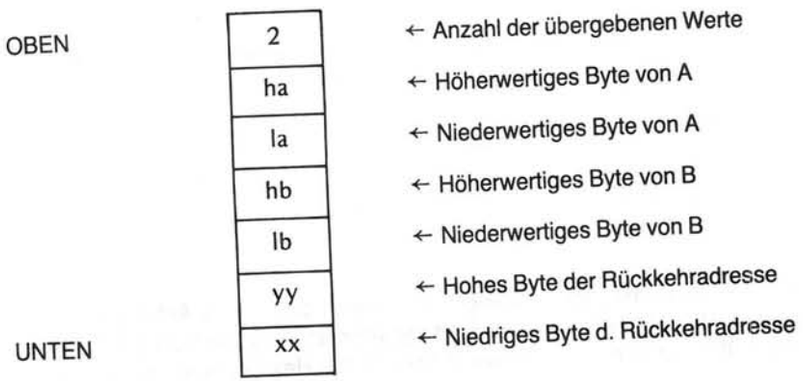
Sie können es mit anderen Zahlen versuchen, indem Sie als Antwort auf READY einfach GOTO 300 eintippen. Das führt zur Wiederholung von Schritt 3. Geben Sie hier nicht RUN, sonst müssen Sie wie in Schritt 2 alle Hex-Codes wieder neu eingeben.

ÜBERGABE MEHRERER VARIABLER

Die USR-Funktion kann vom BASIC-Programm aus mehr als eine Variable an das Maschinen-Unterprogramm übergeben. Das nächste Programm macht dies deutlich. Es übergibt die beiden Variablen A und B, deren Werte dann durch das Unterprogramm summiert werden. Beachten Sie die sowohl in BASIC als auch im Unterprogramm auftretenden Änderungen.

```
          BASIC PROGRAMM – ZWEI VARIABLE UEBERGEBEN

100  CLR: DIM E$(1), E(16)                    Diesmal 16 Bytes

200  FOR Y = 1 TO 16
210       INPUT N                             POKE-Unterprogramm
220       POKE ADR(E$)+Y,N
230  NEXT Y

300  GR. O: PRINT "INPUT A";
310  INPUT A                                  Eingabe zweier Zahlen
320  PRINT "INPUT B";
330  INPUT B                                  A und B werden
340  X=USR(ADR(E$)+1,B,A)                     beide gestapelt
350  PRINT PEEK(6144)
360  PRINT "PRESS RETURN TO REPEAT";
370  INPUT A$:
380  GOTO 300                                 Gehe zurück, um es noch-
                                              mals zu versuchen
```

Bei der Bearbeitung von Zeile 340 sorgt die USR-Funktion für folgende Belegung des Stapelspeichers:



Wie gesagt, werden Datensätze aus dem Stapelspeicher immer von oben herausgeholt, gewissermaßen vom Stapel heruntergenommen. Der zuletzt gestapelte Satz wird als erster wieder geholt. Offenbar werden für jede gestapelte Zahl zwei Bytes verwendet.
Wird mit einer Zahl gearbeitet, die zwei Bytes benötigt, so wird das eine Byte als LSB (mit dem niedrigen Wert – Least Significant Byte), das andere als MSB (mit dem höheren Wert – Most Significant Byte) bezeichnet.

Beispiel:

MSB (Höherwertiges Byte)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	1	0	0	1

LSB (Niederwertiges Byte)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	0	0	1	1	1

Man beachte den Unterschied zwischen MS- und LS-Bytes und MS- und LS-Bits. Jedes Byte hat ein MS-Bit (mit größtem Stellenwert) und ein LS-Bit (mit kleinstem Stellenwert).

HÖHERWERTIGES BYTE

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	1	0	0	1

↑ Bit mit höchstem Stellenwert — ↑ Bit mit niedrigstem Stellenwert

NIEDERWERTIGES BYTE

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	0	0	1	1	1

↑ Bit mit höchstem Stellenwert — ↑ Bit mit niedrigstem Stellenwert

Arbeiten wir mit einer zwei Byte langen Zahl, so fassen wir das MS-Byte als Erweiterung des LS-Bytes auf. Den Stellenwerten des LS-Bytes werden die Zweierpotenzen 2^0 bis 2^7 zugeordnet.

LSB

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	0	0	1	1	0

= 64+4+2 = 70 (dezimal)

Den Stellenwerten des MS-Bytes werden die nächsthöheren Potenzen 2^8 bis 2^{15} zugeordnet.

MSB

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8
1	0	0	0	1	0	0	1

= 32768+2048+256 = 35072 (dezimal)

Der den kombinierten Bytes (als eine Zahl aufgefaßt) entsprechende Dezimalwert lautet:

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	0

Dezimal : 32768+2048+256+64+4+2 = 35142
In 4-Bit Anteile zerlegt:

	1000	1001	0100	0110	← Dieser Binärwert ist äquivalent
Ziffern	8	9	4	6	← zu diesem Hexwert

Hex-Format:

MSB	LSB
89	46

=

16^3	16^2	16^1	16^0
8	9	4	6

→ 8*4096 = 32768
\+ 9*256 = 2304
\+ 4* 16 = 64
\+ 6*1 = 6
35142 (dezimal)

Die Verwendung von Zwei-Byte Zahlen ermöglicht die Eingabe größerer Zahlen (bis 65535). Unser Maschinenprogramm ist jedoch nur für die Addition ganzer Zahlen mit der Länge von einem Byte, d.h. von 0 bis 255, geeignet. Daher sind die MS-Bytes der Variablen A und B beide Null. Die Eingabe von Zahlen, deren Summe größer als 255 ist, liefert ein falsches Ergebnis.
Das Maschinenprogramm ist diesmal länger; es besteht aus 16 Hex-Codes (16 Bytes).

UNTERPROGRAMM IN MASCHINENSPRACHE

Adresse	*Hex Code*	*Funktion*
E$+1	68	Hole den obersten Wert vom Stapel und speichere ihn im Akkumulator
E$+2	18	Lösche das Carry-flag-Bit zwecks späterer Addition der LS-Bytes von Zahlen
E$+3	68	Bringe das MS-Byte (höherwertiges Byte) von A in den Akku
E$+4	68	Bringe das LS-Byte (mit niedrigem Wert) von A in den Akku
E$+5	8D	Speichere das LS-Byte von A im nachfolgenden Speicherplatz
E$+6	01	LS-Byte dieses Platzes
E$+7	18	MS-Byte dieses Platzes
E$+8	68	Bringe das MS-Byte von B in den Akkumulator
E$+9	68	Bringe das LS-Byte von B in den Akkumulator
E$+10	6D	Addiere mit Carry den Wert im Akkumulator und den im nachfolgenden Speicherplatz befindlichen Wert
E$+11	01	LS-Byte des betreffenden Speicherplatzes
E$+12	18	MS-Byte des betreffenden Speicherplatzes
E$+13	8D	Lege das Ergebnis im nachfolgenden Speicherplatz ab
E$+14	00	LS-Byte dieses Platzes
E$+15	18	MS-Byte dieses Platzes
E$+16	60	Rückkehr vom Unterprogramm

Geben Sie nun das BASIC-Programm ein und starten Sie es mit RUN, um den Maschinencode in der üblichen Weise eingeben zu können. Nach der Eingabe von 16 Hex-Codes wird der Bildschirm gelöscht und es erscheint folgendes:

```
INPUT A?■
```

Versuche 135

```
INPUT A?135
INPUT B?■
```

Versuche 99

```
INPUT A?135
INPUT B?99
234
PRESS RETURN TO REPEAT?■
```

In unserem Programm mit zwei Variablen taucht mit der Adresse E$+10 ein neuer Befehl auf, ADC (ADd mit Carry) mit dem Hex-Code 6D. In den vorangegangenen Programmen wurde zur Addition der Hex-Code 69 verwendet. Weshalb dieser Unterschied? Einige Funktionen, wie die Addition, können in verschiedenem Rechenmodus ausgeführt werden, womit wir uns in Kapitel 5 genau befassen werden. Der Befehl 69 addiert den ihm unmittelbar folgenden Wert zu dem bereits im Akkumulator befindlichen. Der Befehl 6D addiert den im unmittelbar danach angegebenen Speicherplatz befindlichen Wert zu dem bereits im Akkumulator gespeicherten Wert. Einige Befehle können nur in einem ganz bestimmten Rechenmodus ausgeführt werden, andere in mehreren. Selbst wenn die gegebene Anweisung die gleiche Operation zur Folge hat, hängt die Art der Ausführung vom Befehlsmodus ab. Daher hat jeder Modus einen anderen hexadezimalen Befehlscode (Operationscode, Op Code als Abkürzung).
Bislang sind Sie nur Maschinenprogrammen begegnet, deren Befehle genau in der Reihenfolge der sie enthaltenden Speicherzellen ausgeführt werden. Sämtliche Rechenoperationen wurden dabei im Akkumulator, d.h. dem Register A ausgeführt. Der 6502 Mikroprozessor besitzt noch weitere Register, die verschiedenen Zwecken dienen.
Da der Rechner in BASIC vermöge der FOR-NEXT-Schleife gleiche Operationen wiederholt ausführen kann, muß es etwas entsprechendes auch in Maschinensprache geben. In Kapitel 1 haben wir uns kurz mit den Registern X und Y befaßt. Wir wollen uns nun anschauen, wie man sie verwendet.

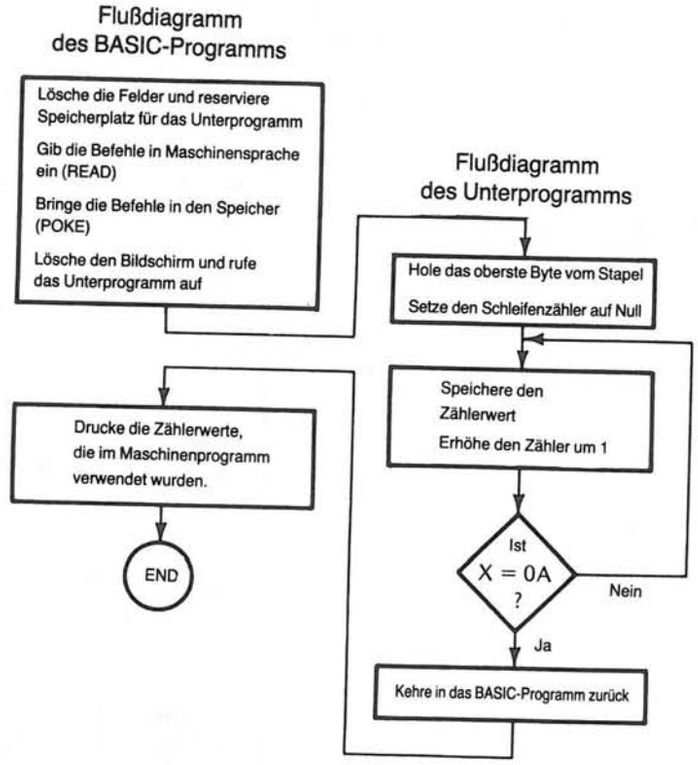
EINE SCHLEIFE IN MASCHINENSPRACHE

In BASIC haben Sie sicher schon oft FOR-NEXT-Schleifen benutzt. Im nächsten Maschinen-Unterprogramm werden wir mit Hilfe des Registers X als Zähler eine entsprechende Technik zum Programmieren einer Schleife verwenden. Vergleichen Sie im folgenden die FOR-NEXT-Schleife mit der Beschreibung einer Schleife in Maschinensprache, die etwa dasselbe leistet.

BASIC	*MASCHINENSPRACHE*
FOR X = 0 TO 9	Lade das X-Register mit Null.
	Speichere den Wert aus dem X-Register.
PRINT X	Erhöhe das X-Register um 1.
NEXT X	Vergleiche das X-Register mit 10.
Sprung →	Springe zurück, wenn X nicht gleich 10 ist.
	Kehre in das Hauptprogramm zurück, wenn X gleich 10 ist.

Das Maschinenprogramm ist detaillierter als das BASIC-Programm. Jeder einzelne Schritt muß in Maschinencode implementiert werden.

Die folgende Darstellung zeigt den Ablauf des BASIC-Programms mit Unterprogramm.



Die Hex-Codes der Maschinenbefehle sind in Anhang A bzw. B aufgeführt. Eine ausführlichere Beschreibung der Maschinenbefehle findet sich im MCS 6500 Microcomputer Family Programming Manual*. Die Befehle sind mit ihrem Buchstabencode, die Operationscodes in hexadezimaler Form aufgelistet. Sie müssen daher diese Codes in dezimale Form konvertieren (umwandeln), um sie mittels POKE in den Speicher Ihres BASIC-Programms bringen zu können. Berechnen Sie die zum folgenden Maschinenprogramm gehörenden Dezimalcodes und füllen Sie die Leerstellen in der Tabelle aus.

* Veröffentlicht von MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401.

UNTERPROGRAMM IN MASCHINENSPRACHE

Adresse	Hex Code	Dezimal Äquivalent	Buchstaben Code	Tabelle
E$+1	68	———	PLA	Hole das Byte vom Stapel
E$+2	A2	———	LDX 0	Lade das X-Register mit Null
E$+3	00	———		
E$+4	8A	———	TXA	Bringe den Wert aus dem X-Register in den Akkumulator
E$+5	9D	———	STA 1800,X	Lege den Wert aus dem Akkumulator im Speicher ab
E$+6	00	———		
E$+7	18	———		
E$+8	E8	———	INX	Erhöhe den Zählerwert im X-Register um 1
E$+9	E0	———	CPX 0A	Vergleiche den Wert im X-Register mit 10 (dezimal)
E$+10	0A	———		
E$+11	D0	———	BNE F7	Springe zu Adresse E$+4 zurück, wenn X = 10 (dezimal)
E$+12	F7	———		
E$+13	60	———	RTS	Kehre in das Hauptprogramm zurück

Abb. 3-4 Übungen in Maschinensprache

Dieses Unterprogramm verwendet einige neue Befehle.

NEU HINZUGEKOMMENE BEFEHLE

Das Maschinenprogramm enthält sechs neue Befehle. Unter der Adresse E$+2 steht A2 (hexadezimal). Dabei handelt es sich um den Operationscode des Befehls „Lade das Register X mit dem Wert, der auf A2 folgt." In unserem Unterprogramm wird das Register X mit Null geladen. Daher gehören der Befehl aus E$+2 und der Wert aus E$+3 (Null) zusammen.

Unter der Adresse E$+4 steht 8A (hexadezimal). Dabei handelt es sich um den Operationscode des Befehls „Bringe den Wert aus dem Register X in den Akkumulator." Um diesen Wert abzuspeichern, was im nächsten Schritt ausgeführt wird, muß er erst in den Akkumulator gebracht werden. Sehr häufig benutzt man den Akku als Zwischenspeicher, wenn Daten von einer bestimmten Stelle an eine andere gebracht werden sollen.

Unter der Adresse E$+5 steht 9D (hexadezimal). Dabei handelt es sich um den Operationscode für den Befehl „Bringe den Wert aus dem Akkumulator in den Speicher." Wir haben hier einen anderen Modus des Befehls „Speichere den Akkumulator." In diesem Modus wird der Wert, der unter der dem Befehl folgenden Adresse steht (1800), zu dem im Register X befindlichen Wert addiert. Der Inhalt des Akkumulators wird unter der so berechneten Adresse abgelegt. Das Register X wird zur Indizierung der Speicherplätze verwendet. Derselbe Befehl kann wiederholt gegeben werden, um über eine entsprechende Änderung des Wertes im Register X Daten in verschiedene Speicherzellen zu bringen.

Unter E$+8 steht E8 (hexadezimal). Dabei handelt es sich um den Operationscode des Befehls „Erhöhe den Wert im Register X um eins." Auf diese Weise wird bei jedem Lauf durch die Schleife der Speicherbefehl unter E$+5 mit einer neuen Adresse ausgeführt.

Unter der Adresse E$+9 steht E0 (hexadezimal). Dabei handelt es sich um den Operationscode „Vergleiche den im Register X stehenden Wert mit dem auf E0 folgenden Wert." Dem Befehl folgt im Beispiel der Wert 0A (dezimal 10). Demnach wird der Wert im Register X mit dem Wert A0 (hexadezimal) verglichen. Damit gelangt der Rechner zur Bedingung X = 0A, oder X ≠ 0A, die ihm im nächsten Schritt eine Verzweigungsentscheidung erlaubt.

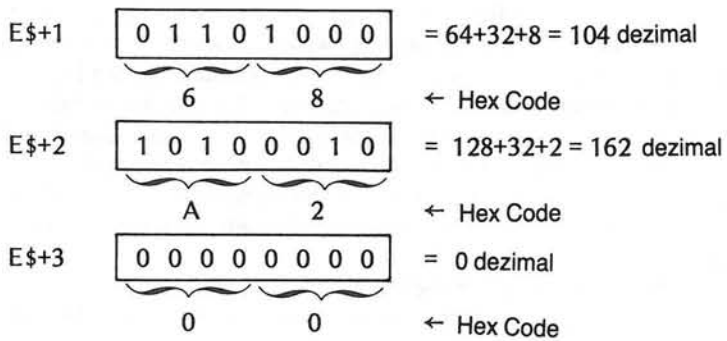
Unter der Adresse E$+11 steht D0 (hexadezimal). Dabei handelt es sich um den Operationscode für den Befehl „Springe, wenn in der vorangegangenen Bedingung das Gleichheitszeichen nicht gilt." Andernfalls gehe in der üblichen Reihenfolge zum nächsten Befehl weiter. Findet der Rechner im Register X einen kleineren Wert als 0A vor, so springt er zum Befehl unter der Adresse E$+4 zurück. Wenn der Wert im Register X auf 0A angewachsen ist, fährt der Rechner mit der Ausführung des Befehls unter E$+13 fort. Der Rechner durchläuft die Schleife 10 mal (nämlich für die Indexregisterwerte (X) von 0 bis 9) und kehrt dann in das Hauptprogramm zurück. Der unter E$+12 verwendete Wert F7 ist die mit Vorzeichen versehene hexadezimale Darstellung der Dezimalzahl -9, d.h. der Rechner muß, wenn die Bedingung X ≠ 0A erfüllt ist, neun Schritte zurückspringen. Größe und Richtung von Programmsprüngen werden in Kapitel 5 behandelt.

Bevor wir das Maschinenprogramm Schritt für Schritt durchgehen, wollen wir uns zunächst das BASIC-Programm anschauen, welches die Werte für das Maschinenprogramm lädt, aufruft und ausgibt.

BASIC PROGRAM

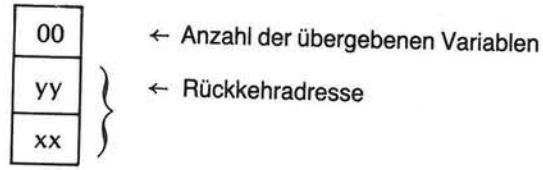
```
100   REM INITIALIZE MEMORY
110   CLR:DIM E$(1), E(13)

120   REM ENTER MACHINE CODES
130   FOR N = 1 TO 13
140       READ C                    ← Lies Codes aus DATA
150       POKE ADR(E$)+N,C          ← Bringe in Unterprogramm
160   NEXT N

170   REM CALL SUBROUTINE
180   X=USR(ADR(E$)+1)

190   REM PRINT RESULTS
200   GR.0                          ← Lösche Schirm
210   FOR N = 0 TO 9
220       PRINT PEEK(6144+N)        ← Drucke die vom
230   NEXT N                          Maschinenprogramm
240   END                             verwendete Schleifenzahl

250   REM DECIMAL DATA
260   DATA 104, 162, 0, 138, 157, 0, 24, 232, 22
4, 10, 208, 247, 96
                                    ← Maschinencodes in
                                      dezimaler Form
                                      (aus Abb. 3–4)
```

Die FOR-NEXT-Schleife in den Zeilen 130 bis 160 liest die Hex-Codes in dezimaler Darstellung ein (READ) und bringt sie an die für das Unterprogramm richtigen Speicherplätze (POKE). Wir halten noch einmal fest, daß die Daten in Binärdarstellung abgespeichert werden, obwohl sie zunächst für BASIC in dezimaler Darstellung vorliegen müssen. Die ersten Bytes des Programms werden in folgender Form abgelegt:



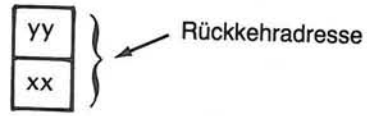
DER ABLAUF DES UNTERPROGRAMMS

Um ein Programm zu verstehen, übernimmt man am besten die Rolle des Rechners und führt jeden Befehl, wie er im Programm vorkommt, aus. Dies wollen wir nun mit unserem Maschinenprogramm durchführen. Dazu müssen wir das Geschehen im Indexregister (Register X), im Akkumulator und im Speicher verfolgen. Auch der Inhalt des Stapelspeichers ist interessant.
Beim Aufruf des Unterprogramms enthält der Stapel:



Wenn das erste Maschinenprogramm ausgeführt wird, ändert sich der Stapelinhalt.

Adresse E$+1 — Befehl „Hole Byte vom Stapel"



Nur die Rückkehradresse bleibt übrig. Wenden wir uns nun dem übrigen Maschinenprogramm zu.

Adresse	*Befehl*	*X*	*Akkumulator*	*Speicher*
E$+2 E$+3	Lade das Register X mit Null	0	–	
E$+4 E$+5	Bringe X nach A Speichere den Inhalt d. Akkumulators unter d. Adr. (1800+X) ab	0	0	
E$+6 E$+7		0	0	1800→ 0
E$+8	Erhöhe den Wert im Register X um eins	1	0	
E$+9 E$+10	Vergleiche X mit 0A (X = 1, nicht 0A)			
E$+11 E$+12	Springe zu E$+4 zurück, wenn X = 1, ≠ 0A	1	0	
E$+4	Bringe X nach A	1	1	

Adresse	Befehle	X	Akkumulator	Speicher
E$+5	Speichere den Inhalt	1	1	1801 → [1]
	d. Akkumulators unter			
E$+6	d. Adr. (1800+X) ab			
E$+7				
E$+8	Erhöhe den Wert im	2	1	
	Register X um eins	2	1	
E$+9	Vergleiche X			
E$+10	mit 0A			
	(X = 2, nicht OA)			
E$+11	Springe zu E$+4	2	1	
E$+12	zurück, wenn X = 2, ≠ 0A			
E$+4	Bringe X nach A	2	2	
E$+5	Speichere den Inhalt	2	2	1802 → [2]
	d. Akkumulators unter			
E$+6	d. Adr. (1800+X) ab			
E$+7				
E$+8	Erhöhe den Wert im	3	2	
	Register X um eins	3	2	
E$+9	Vergleiche X			
E$+10	mit 0A			
	(X = 3, nicht OA)			
E$+11	Springe zu E$+4	3	2	
E$+12	zurück, wenn X = 3, ≠ 0A			
E$+4	Bringe X nach A	3	3	
E$+5	Speichere den Inhalt	3	3	1803 → [3]
	d. Akkumulators unter			
E$+6	d. Adr. (1800+X) ab			
E$+7				
	.			
	.			
	.			

Dieser Prozess wird mit der Erhöhung des Wertes im Register X um eins fortgesetzt. Dieser Wert wird in den Akkumulator und von dort in den Speicherplatz 1800+X gebracht.
Das Register X wird sowohl als Zähler für die Schleife, als auch zum Indizieren der Speicherplätze verwendet (speichere unter den Adressen von 1800 an aufwärts). Wir verfolgen den Ablauf des Programms weiter, nachdem X auf 9 erhöht und der Rücksprung auf E$+4 erfolgt ist.

Adresse	Befehle	X	Akkumulator	Speicher
E$+4	Bringe X nach A	9	9	
E$+5	Speichere den Inhalt	9	9	1800 → [9]
	d. Akkumulators unter			
E$+6	d. Adr. (1800+X) ab			
E$+7				
E$+8	Erhöhe den Wert im	A	9	
	Register X um eins			
E$+9	Vergleiche X	A	9	
E$+10	mit 0A			
	(X=0A, at last)			
E$+11	Springe zu E$+4	A	9	
E$+12	zurück, wenn X ≠ 0A			
E$+13	Kehre vom Unterpro-			
	gramm zurück			

Dem Stapelspeicher wird die Rückkehradresse entnommen. Er ist danach leer. Nach Abarbeitung des Unterprogramms sind die im Register X zugewiesenen Werte (0-9) abgespeichert.

VOM UNTERPROGRAMM GESPEICHERTE DATEN

Hex-Adresse	Dezimal-adresse	Gespeicherter Wert
1800	6144	0
1801	6145	1
1802	6146	2
1803	6147	3
1804	6148	4
1805	6149	5
1806	6150	6
1807	6151	7
1808	6152	8
1809	6153	9

Beachten Sie, daß dabei der letzte in das Register X gebrachte Wert (0A) nicht gespeichert wurde!
Bei der Rückkehr des Unterprogramms in das Hauptprogramm (BASIC-Programm) werden die unter den obigen Adressen gespeicherten Werte auf dem Bildschirm ausgegeben.

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

READY
■
```

ZUSAMMENFASSUNG

In diesem Kapitel haben Sie gelernt:

- Wie man eine hexadezimale Zahl in dezimale Form umwandelt (konvertiert);
- Wie im Atari-Rechner Speicherplatz zugewiesen wird;
- Wie man Speicherplatz für ein Unterprogramm reserviert;
- Wie man das Register X verwendet, um zu zählen, wie oft eine Schleife im Unterprogramm durchlaufen wird;
- Wie man das Register X abfragt, um im richtigen Moment eine Schleife zu beenden;
- Wie man „zu Fuß" durch ein Programm geht, indem man sich in die Rolle des Rechners begibt und die Befehle selbst ausführt;
- Daß einige Befehle in verschiedener Weise (mehr als einem Modus) verwendet werden können;
- Daß die USR-Funktion Werte aus dem BASIC-Programm an das Unterprogramm übergeben kann;

```
1080 A=5: B=6
1090 X=USR(ADR(E$)+1,A,B
```

- Daß die von der USR-Funktion übergebenen Werte als Zwei-Byte Zahlen gestapelt werden;
- Die folgenden neuen Befehle in Maschinencode:

A2	LDX	Lade das Register X mit dem Wert, der auf A2 folgt.
8A	TXA	Bringe den Wert aus dem Register X in den Akkumulator

9D	STA	Speichere den Inhalt des Akkumulators unter der unmittelbar folgenden Adresse ab. Diese wird durch das Register X angezeigt (indiziert).
E8	INX	Erhöhe den Wert im Register X um eins
E0	CPX	Vergleiche den im Register X stehenden Wert mit dem auf E0 folgenden Wert.
D0	BNE	Springe um die angegebene Schrittzahl vorwärts oder rückwärts, wenn die Bedingung nicht Null ist.

ÜBUNGEN

1. Wandle die folgenden hexadezimalen Zahlen in die entsprechenden Dezimalzahlen um.

   A2 hex = _____ dezimal
   3B hex = _____ dezimal
   9E hex = _____ dezimal

2. Es ist gefährlich (d. h. äußerste Vorsicht ist geboten), mittels POKE _____ (BASIC, Maschinen) Befehle in den Speicher unter Ihr _____ (BASIC, Maschinen) Programm zu bringen.

3. Wenn Ihr BASIC-Programm die String-Variable E$ auf eins dimensioniert, reservieren Sie Speicherplatz über Ihrem BASIC-Programm. Bei welcher Adresse können Sie risikolos ein vom BASIC-Programm aufgerufenes Unterprogramm in Maschinencode beginnen? _____

4. Hängt der für E$ reservierte Speicherplatz von der Länge Ihres BASIC-Programms ab? _____

5. Datenwerte, die Variablen zugeordnet sind, können von einem BASIC-Programm an ein Maschinen-Unterprogramm übergeben werden. Aber auch dann, wenn kei-

ne Variablen an das Unterprogramm übergeben werden, muß dieses eine bestimmte Operation mit dem Stapelspeicher durchführen, ehe ein Rücksprung ins Hauptprogramm möglich ist. Um welche Operation handelt es sich?

_____

6. Das Register X enthalte den Wert 7 und der Akkumulator den Wert 1A. Der Rechner führt damit folgenden Befehl aus:

```
E$+7    9D    STA 1800,X
E$+8    00
E$+9    18
```

Unter welcher Adresse ist der Wert 1A gespeichert? _____

7. Es soll das BASIC-Programm von Seite 44 mit dem Unterprogramm von Seite 42 ausgeführt werden. Die einzige Änderung findet im BASIC-Programm statt. Zeile 210 wird ersetzt durch:

```
210 FOR N = 3 TO 5
```

Schreiben Sie auf, was nach Ausführung des gesamten Programms auf dem Bildschirm erscheint.

Antworte hier

8. Wie müßten Sie das Unterprogramm von S. 42 abändern, wenn die Schleife bis 20 anstatt bis 10 (dezimal) zählen soll?

Adresse	Hex-Code

9. Wie würden Sie das BASIC-Programm von S. 44 ändern, damit es Ihnen die vom abgeänderten Unterprogramm gespeicherten 20 Werte (Übung 8) ausgibt?

_____

260 DATA _____

_____

ANTWORTEN

1. A2 hex = 162 dezimal
   3B hex = 59 dezimal
   9E hex = 158 dezimal

2. Maschinen BASIC

3. E$+1 oder ADR (E$)+1

4. Ja (Der Platz für E$ wird am Ende Ihres Programms reserviert).

5. Ein Byte muß vom Stapel heruntergeholt werden (Anzahl der an das Unterprogramm übergebenen Variablen hier Null), ehe die Rücksprungadresse verfügbar ist.

6. Adresse 1807 (1800 + 7)

   Wert im Register X

7.
```
3
4
5

READY
■
```

8. E$+10 14 (16+4 = 20)

9. Zeile 210 FOR N = 0 TO 19
   Zeile 260 DATA 104,162,0,138,157,0,24,232,22,
   4,20,208,247,95

   Ändern Sie diese Werte

Einführung in den Assembler

Um ein Programm in Assembler-Sprache zu schreiben, zu assemblieren und auszuführen, braucht man nur den Atari 400 oder 800 und den Assembler Modul. Es empfiehlt sich jedoch eine zusätzliche Speichermöglichkeit (ein Atari 410 Program Recorder oder Atari 810 Disk Drive - Band bzw. Platte). Andernfalls muß ein Programm jedesmal neu über die Tastatur eingegeben werden, wenn es gebraucht wird. Der Atari 820 Drucker kann zusätzlich angeschlossen werden. Er bietet Ihnen die Möglichkeit, Ihre Programme in leicht lesbarer Form zu dokumentieren. Bei der Arbeit an diesem Buch verwendeten die Autoren die im folgenden Diagramm geschlossen umrandeten Systemelemente. Die übrigen, wahlweise zu verwendenden Elemente, sind gebrochen umrandet.

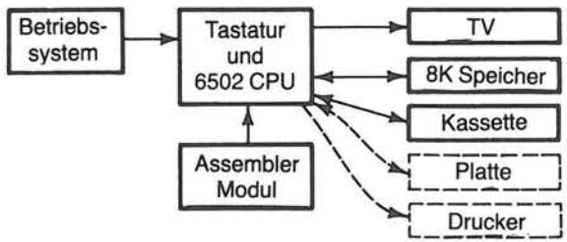


Abb. 4-1 Unser Atari 800 System

Die Kommunikation zwischen den einzelnen Elementen des Systems Atari 400/800 wird durch das Atari Betriebssystem gesteuert, das sich in einem 10K Festwertspeicher (ROM - 10,000- Byte Read Only Memory) befindet und zugeordnete Teile des programmierbaren Speichers RAM (Random Access Memory) mit verwendet. Der Assembler gelangt bei Bedarf in das Betriebssystem.
Der Atari Assembler Modul enthält drei getrennte Programme:
1. Writer/Editor
2. Assembler
3. Debugger
Das *Writer/Editor-Programm* dient, wie schon der Name sagt, zum Schreiben und Editieren Ihres Programms. Die Assemblersprache bzw. der Assembler ist eine Kurzschrift, die Englisch-ähnliche Abkürzungen für Rechnerbefehle verwendet. Der

Assembler benutzt auch Zahlen in hexadezimaler oder dezimaler Form als Programmdaten.
Das *Assembler-Programm* übersetzt die vom Writer/Editor kommenden Abkürzungen in Maschinencode und Daten, die der Rechner versteht. Er sorgt auch dafür, daß Befehle und Daten an die geeigneten Speicherplätze gelangen.
Das *Debugger-Programm* (Korrekturprogramm) dient der Ausführung, dem Testen oder dem schrittweisen Verfolgen des vom Assembler erzeugten Maschinenprogramms.
Die drei Programme werden, wie Sie sehen, in logischer Reihenfolge verwendet. Liefert der Debugger ein fehlerhaftes Programm, so können Sie zum Writer/Editor-Programm (WIE) zurückgehen und Änderungen vornehmen. Dieser Vorgang wiederholt sich solange, bis die gewünschten Ergebnisse erzielt werden. Das folgende Flußdiagramm zeigt den Ablauf der drei Programme.

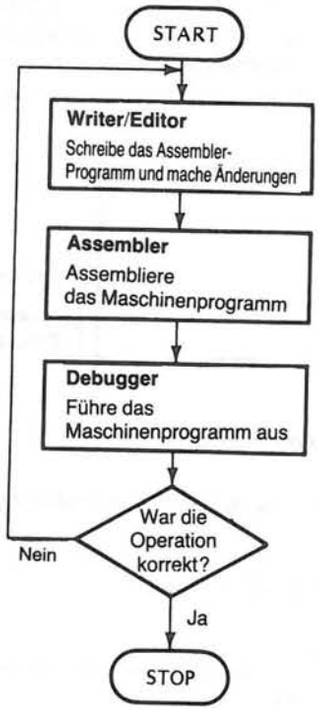
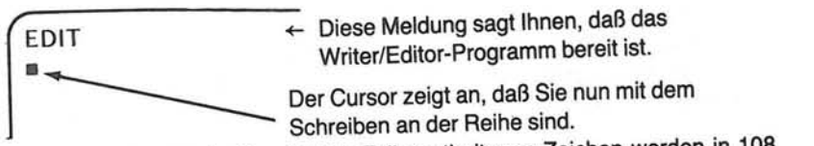


Abb. 4-2 Flußdiagramm des Assembler Moduls

DAS SCHREIB- UND EDITIONSPROGRAMM (WRITER/EDITOR)

Lassen wir zunächst das mit dem Assembler und der Maschinensprache einhergehende Fachchinesisch beiseite, schalten stattdessen unseren Rechner ein und versuchen es mit einem kurzen Programmbeispiel. Bringen Sie den Assembler Modul in den linken Schlitz des Rechners ein, sehen Sie zu, daß Ihr Fernseher eingeschaltet und mit

dem Rechner verbunden ist und schalten Sie den Rechner ein. Danach muß folgendes auf dem Bildschirm erscheinen:



Die in der von Ihnen gerade eingetippten Zeile enthaltenen Zeichen werden in 108 Speicherplätzen zwischengespeichert. Dieser kleine Teil des Speichers wird Zeilenpuffer genannt *(Current Line Buffer)*. Mit Beginn jeder neuen Zeile wird der Puffer überschrieben. Er ist Teil der 384 (180 HEX) für den Assembler Modul reservierten Speicherplätze.
Unmittelbar über dem für den Assembler reservierten Platz befindet sich der Edit Text Puffer. In ihm werden alle über die Tastatur eingegebenen Zeichen beim Schreiben eines Assembler-Programms gespeichert. Je mehr Sie eintippen, desto größer wird der Inhalt des Edit Text Puffers. Seine Aufnahmefähigkeit ist nicht durch eine definierte Datenmenge beschränkt. Man darf freilich die Speicherkapazität des Rechners nicht überschreiten.

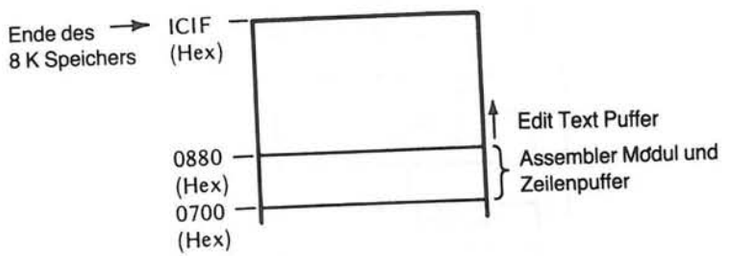
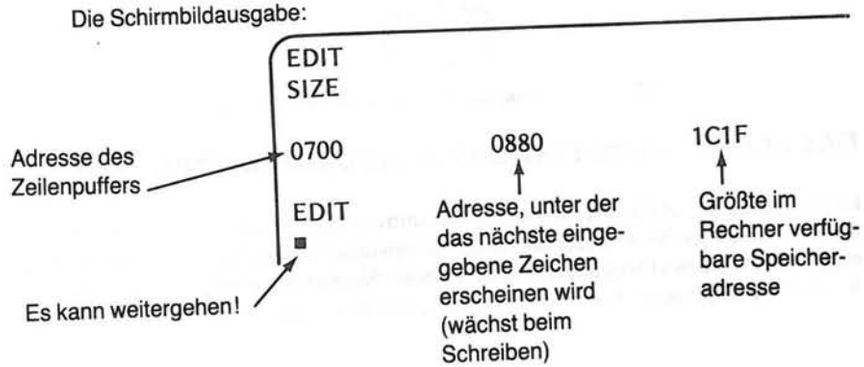


Abb. 4-3 Pufferspeicher

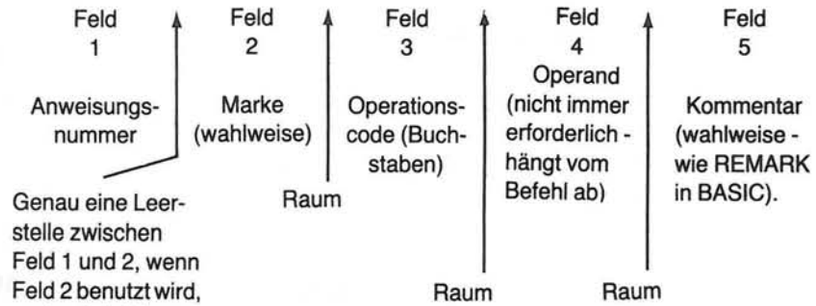
Um den Zeilen- und Edit Text Puffer im Speicher zu finden, geben Sie den Befehl:

SIZE (und drücken der RETURN-Taste)

Die Schirmbildausgabe:



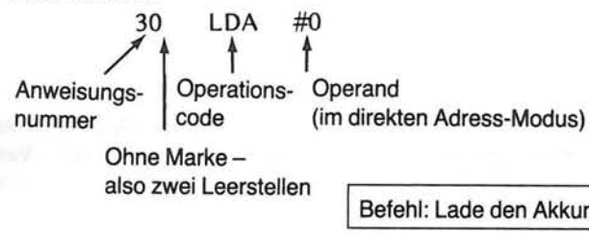
Die auf dem Bildschirm ausgegebenen Zahlen hängen natürlich von der Speicherkapazität Ihres Systems ab. Wir arbeiten hier, wie schon gesagt, mit dem Atari 800 mit 8K oder RAM und ohne Platte.
Mit dem SIZE-Befehl können Sie sich jederzeit über die Länge Ihres Programms informieren. Ziehen Sie die erste Zahl (in unserem Fall 700) + 180 von der zweiten Zahl ab. Das Ergebnis ist die ungefähre Zeichenanzahl in Ihrem Programm.
Bevor wir unser Programm eingeben, wollen wir uns das für das Schreiben von Assembler-Programmen verwendete Format anschauen. Das Programm, welches assembliert werden soll, wird Quellprogramm (Source Program) genannt. Es besteht ebenso wie ein BASIC-Programm aus Anweisungen. Diese Anweisungen werden numeriert, wie in BASIC die Zeilen numeriert werden. Jede Anweisung muß mit einer Zeilennummer zwischen 0 und 65535 beginnen. Nach dem Eingeben jeder einzelnen Anweisung, unter Zuhilfenahme des Writer/Editor-Programms des Assembler Moduls, wird das Ende der Anweisung durch Drücken der RETURN-Taste gekennzeichnet.
Das für jede Anweisung verwendete Format besteht aus fünf Feldern.



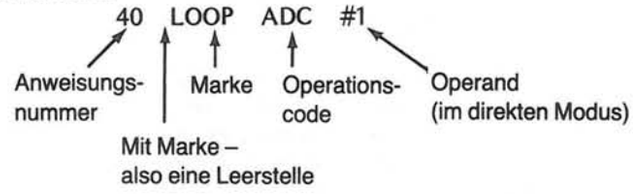
Beispiele:

1. Ohne Marke

2. Ohne Marke

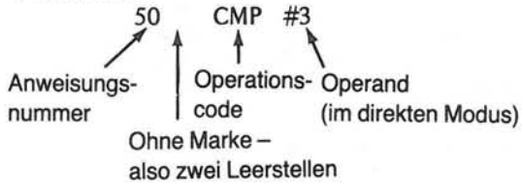


3. Mit Marke



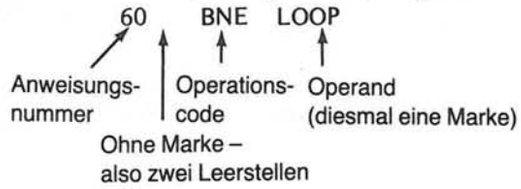
Befehl: Addiere eins zu dem Wert im Akkumulator

4. Ohne Marke



Befehl: Vergleiche den Wert im Akkumulator mit 3

5. Keine Marke – Operand gibt das Sprungziel an



Befehl: Wenn die beiden Werte nicht gleich sind, springe zu dem Befehl mit der LOOP-Marke zurück

Wir wollen nun mit den obigen Anweisungen ein Assembler-Programm schreiben. Der Rechner befand sich zuletzt im Edit-Modus.

```
EDIT
SIZE
0700                0800            1C1F

EDIT
■
```

Zunächst müssen wir dem Rechner sagen, an welcher Stelle er im Speicher mit dem Maschinenprogramm, dem sogenannten *Objekt-Programm*, beginnen soll. Wir verwenden dazu einen besonderen Befehl. Die Startadresse für das Maschinenprogramm muß weit genug über der des Edit Text Puffers, in unserem Fall 0880, liegen, damit ein Überlappen beider Programme vermieden wird.

Schreiben Sie: 10 *=$1000

Zeilen-nummer — 2 Leer-stellen — Startadresse ($ zeigt eine hexadezimale Zahl an)

Nun wird das restliche Programm eingegeben. Vergessen Sie nicht, am Ende jeder Zeile RETURN zu geben (genau wie in BASIC).

```
20      CLC
30      LDA #0
40      LOOP ADC #1
50      CMP #3
60      BNE LOOP
```

Das Programm muß mit einer END-Anweisung abgeschlossen werden (PSEUDO OPERATOR oder *Directive*). Jedes Programm sollte eine und nur eine END-Anweisung enthalten.

```
70    END
```

1 Leerstelle

DAS ASSEMBLER-PROGRAMM

Nachdem das Programm mit Hilfe des Writer/Editor-Programms eingegeben wurde, wird der nächste Schritt vom Assembler-Programm ausgeführt. Der Assembler konvertiert die Assembler-Befehle in Maschinencode und weist den Maschinenbefehlen Speicherplätze zu.
Vor dem Aufruf des Assembler-Programms sehen Sie auf dem Bildschirm:

```
EDIT
10    *=$1000
20    CLC
30    LDA #0
40  LOOP ADC #1
50    CMP #3
60    BNE LOOP
70  END
```

Wir wollen versuchen, den für unser Programm in Assembler Sprache verbrauchten Speicherplatz festzustellen. Beim Eintippen von SIZE erschien früher, wie Sie sich erinnern, auf dem Schirm:

```
EDIT
SIZE
0700                0880                1C1F
```

Um zu sehen, wieviel Speicherplatz verbraucht ist, tippen Sie nochmals SIZE.

```
60   BNE LOOP
70   END
SIZE
0700                08D4                1C1F
EDIT                 ↑
■                    |
                  Neuer Wert
```

Unser Programm hat 8D4-880 oder 54 (hex) Speicherplätze verbraucht:

Wir sehen, daß zwischen dem Programm in Assembler-Sprache (Edit Text Puffer) und dem Maschinenprogramm ungenutzter Speicherplatz liegt. Das ist auch gut so. Bei größeren Programmen in Assembler muß man aufpassen, daß sich das Assembler-Programm und das zugehörige Maschinenprogramm nicht überlappen. Wir können uns nun dem Assembler-Programm zuwenden.
Schreiben Sie: ASM (und geben Sie RETURN)
Danach erscheint jeder Maschinencode mit dem Assembler-Befehl, der ihn erzeugt hat, auf dem Bildschirm.

```
.
.
.
60   END
ASM
0000                10              *=    $1000

1000 18             20              CLC

1001 A900           30              LDA #0

1003 6901           40 LOOP         ADC #1
```



```
1005 C903           50              CMP #3

1007 D0FA           60              BNE LOOP

                    70 END

EDIT
■
```

Vom Assembler erzeugtes Maschinenprogramm **(Objekt-Programm)**

Assembler-Sprache **(Quellprogramm)**

Das von Ihnen in Assembler-Sprache eingegebene **Quell-Programm** erscheint auf der rechten Seite des Schirms. Das vom Assembler erzeugte **Objekt-Programm** sehen Sie auf der linken Seite des Schirms. Zu jedem Maschinenbefehl ist die Adresse angegeben, unter der er gespeichert ist. Wir wollen nun die Programme zeilenweise durchsprechen, um zu sehen, wie die Befehle einander zugeordnet sind.

Maschinencode	*Assembler-Sprache*
ASM 0000	10 *= $1000 Zeile 10 legt die Startadresse des Maschinenprogramms auf 1000 (hex) fest.
1000 18	20 CLC Der Befehl CLC wird in den Maschinencode 18 umgewandelt. Der Befehlscode 18 wird unter der Adresse 1000 abgelegt.
1001 A900	30 LDA #0 Die Anweisung LDA#0 wird in den Maschinencode A900 umgewandelt. Der Befehl A9 (entspricht LDA#) wird unter der Adresse 1001 abgelegt und der Wert (00) unter 1002.
1003 6901	40 LOOP ADC#1 Die Anweisung ADC #1 wird in den Maschinencode 6901 umgewandelt. Der Befehl 69 (entspricht ADC #) wird unter der Adresse 1003 und der Wert (01) unter 1004 abgelegt.

1005 C903 50 CMP #3

Die Anweisung CMP #3 wird in den Maschinencode C903 umgewandelt. Der Befehl C9 (entspricht CMP #) wird unter der Adresse 1005 und der Wert (03) unter 1006 abgelegt.

1007 DOFA 60 BNE LOOP

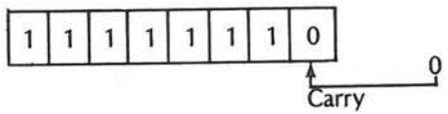
Die Anweisung BNE LOOP wird in den Maschinencode DOFA umgewandelt. Der Befehl DO (entspricht BNE) wird unter der Adresse 1007 und die Sprungweite (FA) unter 1008 abgelegt. Wenn der Rechner springt, dann nach 1003, wo die Schleife (LOOP) angefangen hat.

70 END

Zeile 70 sagt dem Assembler, wo er aufhören soll. Dazu wird kein Maschinencode erzeugt.

Wir haben in diesem Programm fünf Assembler-Anweisungen benutzt. Diese Anweisungen bestehen aus einer Buchstabenabkürzung, ihrer englischen Bedeutung und gewöhnlich einem Operanden. Im folgenden wird die jeweilige Bedeutung angegeben.

Buchstaben	*Operand*		*Bedeutung*
CLC			Lösche das Carry-Bit im Statusregister (auf Null setzen)

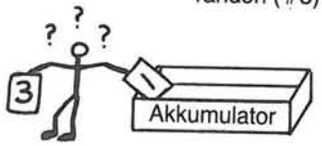


LDA	#0	=	Lade den Akkumulator mit dem unmittelbar folgenden Operanden (*#0*)

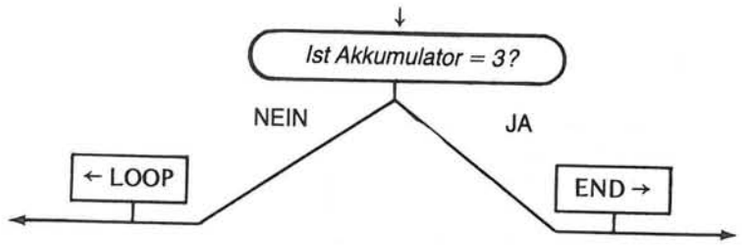
ADC	#1	=	Addiere zu dem Akkumulator-Inhalt mit Carry den unmittelbar folgenden Operanden (#1)

Füge 1 hinzu

CMP	#3	=	Vergleiche den Wert im Akkumulator mit dem Operanden (#3)



BNE	LOOP	=	Springe im Falle der Ungleichheit zum Befehl mit der Schleifenmarke von LOOP



Das Maschinenprogramm ist assembliert und wie folgt im Speicher abgelegt:

Speicher-platz	*Maschinen Code*	
1000	18	Anweisung CLC
1001	A9	Anweisung LDA #
1002	00	Daten geladen
1003	69	Anweisung ADC #
1004	01	Diese Daten sollen addiert werden
1005	C9	Anweisung CMP #
1006	03	Diese Daten sollen verglichen werden
1007	D0	Anweisung BNE
1008	FA	Sprungadresse

Abb. 4-4 Speicher für das Maschinenprogramm

DIE AUSFÜHRUNG DES OBJEKT-PROGRAMMS – DAS KORREKTURPROGRAMM (DEBUGGER)

Mit Hilfe des Writer/Editor Programms haben Sie Ihr Programm in Assembler geschrieben. Das Assembler-Programm hat den dazugehörenden Maschinencode erzeugt und gespeichert. Jetzt ist es Zeit, das Korrekturprogramm aufzurufen, um das Maschinenprogramm auszuführen.
Sicher ist Ihnen aufgefallen, daß der Rechner nach dem Assemblieren Ihres Programms zum Writer/Editor Programm zurückgekehrt ist, EDIT auf dem Bildschirm ausgegeben und den Cursor in die nächste Zeile plaziert hat.

```
.
.
.
EDIT
■
```

Sie geben nun durch die Buchstaben BUG das Korrekturprogramm ein →

Der Rechner antwortet: →

```
EDIT
BUG

DEBUG
■
```

Der Rechner wartet nun auf einen Korrektur-Befehl (DEBUGGER). Um nicht die Übersicht zu verlieren, wollen wir uns für den Moment auf zwei dieser Befehle (es gibt eine Reihe davon) beschränken. Der erste hat die Form:

```
GXXXX
```

G für GOTO

XXXX repräsentiert die hexadezimalen Ziffern derjenigen Adresse, unter der Ihr Programm beginnt.

Wenn Sie G1000 und anschließend RETURN tippen, wird das Programm sofort ausgeführt. Auf dem Schirm erscheint folgendes:


```
.
.
.
EDIT
BUG

DEBUG
G1000
1009        A=03 X=00 Y=00 P=33 S=00
DEBUG
■
```

Sie haben dies getippt

Der Rechner antwortet

Was soll das bedeuten?

1. Die Zahl 1009 besagt, daß das Programm bei dieser Adresse gestoppt wurde. Es hat bis dahin alle Befehle unter den Adressen 1000 bis 1008 ausgeführt.

2. Der einzige weitere Wert in der Zeile, der uns für den Moment interessiert, ist A =03. A steht für Akkumulator, 03 bedeutet, daß sich dieser Wert nach Beendigung des Programms im Akkumulator befindet. Das ist genau das, was wir wollten. Unser Programm hat den Akkumulator ursprünglich mit 0 geladen (LDA #0). Dann addierte es bei jedem Schleifendurchlauf eine 1, bis der Akkumulator den Wert 3 enthielt. Danach stoppte der Computer.

Wir wollen nun Schritt für Schritt das Zählen bei der Programm-Ausführung verfolgen. Entsinnen Sie sich, daß am Ende der letzten Programm-Ausführung die Meldung DEBUG auf dem Bildschirm erschien? Das bedeutet, daß wir uns noch immer im Korrekturprogramm befinden und unseren zweiten DEBUGGER-Befehl ausprobieren können. Wir können das Programm schrittweise verfolgen, indem wir tippen

```
T1000
```

(und die RETURN-Taste drücken):

T für TRACE

Startadresse 1000

Danach erscheint folgendes auf dem Bildschirm:

```
1000        18        CLC          } ← { Die beiden obersten Zeilen
   A=00 X=00 Y=00 P=32 S=00       }   { verschwinden nach oben

1001        A9 00     LDA   #$00
   A=00 X=00 Y=00 P=32 S=00
1003        69 01     ADC   #$01   ← Eins zum Akkumulator addiert
   A=01 X=00 Y=00 P=30 S=00
1005        C9 03     CMP   #$03
   A=01 X=00 Y=00 P=B0 S=00
1007        D0 FA     BNE   $1003
   A=01 X=00 Y=00 P=B0 S=00
1003        69 01     ADC   #$01   ← Springe zurück und addiere
   A=02 X=00 Y=00 P=30 S=00          nochmals 1
1005        C9 03     CMP   #$03
   A=02 X=00 Y=00 P=B0 S=00
1007        D0 FA     BNE   $1003
   A=02 X=00 Y=00 P=B0 S=00
1003        69 01     ADC   #$01   ← Springe zurück und addiere
   A=03 X=00 Y=00 P=30 S=00          nochmals 1
1005        C9 03     CMP   #$03
   A=03 X=00 Y=00 P=33 S=00          Jetzt ergibt sich Gleichheit –
1007        D0 FA     BNE   $1003    also kein Sprung
   A=03 X=00 Y=00 P=33 S=00
1009        00        BRK            BRK (Abkürzungen von break) –
   A=03 X=00 Y=00 P=33 S=00          Das Programm hält an
DEBUG
■
```

Abb. 4-5 Sichtbarmachen des Programmablaufs

Beachten Sie, daß sich der Wert im Akkumulator (A) jedesmal, wenn der unter der Adresse 1003 stehende Befehl ADC #01 ausgeführt wird, um eins erhöht. Nach jeder Anweisung in einer Zeile wird der Wert im Akkumulator in der Form A = nn angezeigt (wobei nn für die beiden hexadezimalen Ziffern steht). Die Werte in den anderen Registern, die Sie sehen, können sich von den hier gezeigten unterscheiden. Das braucht Sie aber vorderhand nicht zu interessieren. Ihre Aufmerksamkeit richtet sich im Moment ganz auf den Akkumulator. Die anderen Register werden wir in Kapitel 5 behandeln.

Wenn Sie einmal sehen möchten, wie schnell der Rechner bis 255 zählen kann (FF HEX), dann ändern Sie den Vergleichswert in Zeile 50:

```
50 CMP #$FF      Das $-Zeichen bedeutet eine Hex-Zahl
                 (ohne $ eine Dezimalzahl).
```

Diese Änderung bewirkt, daß der Rechner nicht bei 3, sondern erst bei FF aufhört. Um dies zu erreichen, hätten wir ein ganz neues Programm eingeben können. Es ist aber

einfacher, das alte entsprechend zu ändern.

Nachdem wir „zu Fuß" durch unser Programm gegangen sind, befinden wir uns noch immer im Korrekturprogramm. Um Zeile 50 zu ändern, müssen wir in das Writer/Editor Programm zurückkehren.

Tippen Sie : X (und RETURN)

```
.
.
.
1009        00              BRK
 A = 03 X = 00 Y=00 P=33 S=00
DEBUG                                ← Tippen Sie: X
X

EDIT
■
```

Werfen Sie einen Blick auf das vorangegangene Assembler-Programm und raten Sie, welcher Befehl verwendet wird.

Tippen Sie: LIST (und RETURN)

Genau wie BASIC

```
.
.
.
EDIT
LIST

10     *=$1000
20     CLC
30     LDA #0
40   LOOP ADC #1
50     CMP #3
60     BNE LOOP
70   END
■
```

Um die Zeile 50 zu ändern, müssen Sie lediglich die neue Zeile eingeben (natürlich mit Zeilennummer).

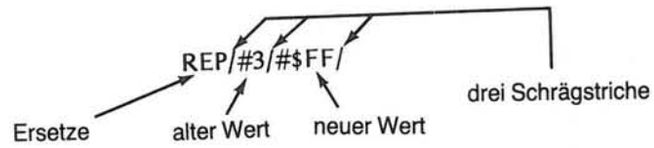
Tippen Sie: 50 CMP #$FF ($ nicht vergessen).

```
.
.
.
70   END
50    CMP #$FF
■
```

Um die Korrektheit der Änderung zu überprüfen, LISTen Sie Ihr Programm noch einmal auf.

```
.
.
.
70   END
50    CMP #$FF
LIST

10    *=$1000
20    CLC
30    LDA #0
40   LOOP ADC #1
50    CMP #$FF          ← Ja, Änderung vorhanden
60    BNE LOOP
70   END
```

Eine andere Möglichkeit zur Änderung der Vergleichszahl hätte die REPLACE-Anweisung im Editions-Modus geboten. Wir hätten diesen Modus wählen können und schreiben:



Das Ergebnis wäre das gleiche gewesen.
Wir haben nun zwar das in Assembler geschriebene Programm geändert, jedoch noch *nicht* das Maschinenprogramm. Das geänderte Programm muß *assembliert* werden.

Tippen Sie: ASM (und RETURN)

```
.
.
.
60    END
ASM
0000              10          *=     $1000
1000  18          20          CLC
1001  A900        30          LDA    #0
1003  6901        40 LOOP     ADC #1
1005  C9FF        50          CMP    #$FF
1007  D0FA        60          BNE    LOOP
                  70  END

EDIT
■
```

Zur Ausführung des Programms benötigen wir, wie Sie schon wissen, das Korrekturprogramm. Daher geben Sie ein: BUG

```
.
.
.
EDIT
BUG

DEBUG
■
```

Lassen Sie nun das Programm mittels G1000 laufen.

```
DEBUG
G1000
1009                A=FF X=00 Y=00 P=33 S=00
DEBUG                        ↖ Da haben wir es.
■
```

Das ging aber schnell! Wie können wir herausfinden, ob der Rechner tatsächlich durchgezählt hat und nicht etwa nur das fertige Ergebnis ausgibt? Dazu lassen Sie uns das Programm verfolgen.
Konzentrieren Sie sich auf die Spalte, in der der Inhalt des Akkumulators erscheint (A). Die Ergebnisse erscheinen nämlich ziemlich schnell, während sich der Schirm füllt und das Bild nach oben verschwindet. Der Akkumulator braucht etwa 2 bis 3 Minuten, um FF zu erreichen.

Tippen Sie: T1000 (und RETURN)

Munter laufen die Daten über den Schirm. Schließlich hält das Programm am Ende an.

```
.
.
.
1007            D0 FA           BNE  $1003
      A=FF X=00 Y=00 P=33 S=00
1009            00              BRK
      A=FF X=00 Y=00 P=33 S=00
DEBUG
■
```

Ja, es wurde wirklich bis FF gezählt.

Dazu hat der Rechner einige Zeit gebraucht. Er mußte ja aber auch laufend die Inhalte aller Register verfolgen und ausgeben. Es geht natürlich viel schneller, wenn man den Rechner sich selbst überläßt, ohne ihm Sonderaufgaben aufzubürden.
Wir wissen inzwischen, daß man den im Akkumulator befindlichen Wert durch Addition um eins erhöhen kann. Wir können aber auch Werte vom Akkumulator-Inhalt subtrahieren. Warum nicht bei FF anfangen und bei jedem Lauf durch die Schleife eins abziehen? Dazu brauchen wir zwei neue Befehle.

SBC (Subtrahiere vom Akkumulator mit „borgen" (borrow), d.h. negativem Übertrag, dem Gegenteil von Carry = positiver Übertrag)

SEC (Setze das Carry-Bit. Dies ersetzt die Anweisung zum Löschen des Carry-Bits)

Wir nehmen unser altes Programm als Muster für das neue.

ADDITIONS-PROGRAMM			SUBTRAKTIONS-PROGRAMM
10 *=$1000			10 *=$1000
20 CLC	← Lösche und setze →		20 SEC
30 LDA #0	← Beginne bei →		30 LDA #$FF
40 LOOP ADC #1	← Addiere und subtrahiere		40 LOOP SBC #1
50 CMP #$FF	← Gehe bis →		50 CMP #0
60 BNE LOOP			60 BNE LOOP
70 END			70 END

Wenn Sie das Addierprogramm noch im Speicher haben (Prüfen Sie dies. Gehen Sie in das Writer/Editor Programm und LISTen Sie das Programm auf), können Sie die Zeilen, 20, 30, 40 und 50 ändern und neu assemblieren. Andernfalls geben Sie das gesamte Subtraktionsprogramm neu ein und assemblieren dann.
Danach gehen Sie in das Korrekturprogramm und lassen Ihr neues Programm mit der Eingabe G1000 laufen.

G1000

```
EDIT
BUG

DEBUG
G1000
1009                A=00 X=00 Y=00 P=33 S=00
DEBUG
■
```

Es wird rasch bis 0 gezählt.

Mit der BREAK-Taste können Sie das Programm an passender Stelle anhalten. Wie das folgende Schirmbild zeigt, haben wir unser Programm kurz nach dem Start angehalten.

Tippen Sie: T1000

```
DEBUG
T1000
1000            38              SEC
      A=00 X=00 Y=00 P=B1 S=00
1001            A9 FF           LDA #$FF
      A=FF X=00 Y=00 P=B1 S=00
1003            E9 01           SBC #$01
      A=FE X=00 Y=00 P=B1 S=00
1005            C9 00           CMP #$00
      A=FE X=00 Y=00 P=B1 S=00
1007            D0 FA           BNE $100
      A=FE X=00 Y=00 P=B1 S=00
1003            E9 01           SBC #$01
      A=FD X=00 Y=00 P=B1 S=00
1005            C9 00           CMP #$00
      A=FD X=00 Y=00 P=B1 S=00
1007            D               ← An dieser Stelle haben wir die
DEBUG                             BREAK-Taste gedrückt
■
```

Der Akkumulator ist von FF auf FD herunter.

Lassen Sie das Programm durchlaufen, bis der Akkumulator auf Null heruntergezählt hat, so sehen Sie, daß der Rechner rückwärts genau so schnell zählt wie vorwärts. Schauen Sie auf den Akkumulator, während Ihr Programm über den Schirm läuft.
Lassen Sie uns, ehe wir fortfahren, zusammenfassen, was wir bisher gelernt haben. Sicher verdaut man den Assembler leichter in mehreren kleinen Portionen, als in wenigen dicken Brocken. Eine weitergehende Darstellung der Möglichkeiten des Assemblers findet sich in Kapitel 7.

ZUSAMMENFASSUNG

Der Assembler Modul besteht aus drei Programmen:

1. Dem **Writer/Editor Programm** – zum Schreiben und Editieren von Programmen in Assembler-Sprache.
2. Dem **Assembler-Programm,** das Assembler-Anweisungen in Maschinencode übersetzt und sie geeignet speichert.
3. Dem **Korrekturprogramm,** das der Ausführung von Programmen in Maschinensprache dient.

Verwendete Schlüsselworte
Writer/Editor

1. EDIT - Mitteilung des Rechners, daß Sie im Writer/Editor Programm sind.
2. SIZE - Gibt die Adressen des Zeilenpuffers, des EDIT Text Puffers und die höchste verfügbare Speicheradresse an.
3. LIST - Listet Ihr Assembler-Programm auf dem Bildschirm auf.
4. BUG - Ein Befehl, der den Übergang vom Writer/Editor- zum Korrekturprogramm bewirkt.
5. ASM - Ein Befehl, der den Übergang vom Writer/Editor- zum Assembler-Programm bewirkt.

Debugger (Korrekturprogramm)

1. DEBUG - Mitteilung des Rechners, daß Sie im Korrekturprogramm sind.
2. GXXXX - Befehl, mittels dessen die Ausführung des unter der Adresse XXXX (jedes X ist eine Hex-Ziffer) beginnenden Programms in Maschinencode bewirkt wird.
3. TXXXX - Befehl, der das Verfolgen jedes einzelnen Schrittes bei der Ausführung der Maschinenbefehle mit der Adresse XXXX auf dem Bildschirm ermöglicht (wieder ist jedes X eine Hex-Ziffer).
4. X - Befehl, der den Übergang vom Korrekturprogramm zum Writer/Editor-Programm bewirkt.

Verwendete Anweisungen:

Anweisungen in Assembler (für das Quellprogramm)	*Erzeugter Maschinencode (für das Objekt-Programm)*	*Beschreibung*
LDA#0	A9 00	Lade den Akkumulator mit 0 – Direkter Adress-Modus
ADC#1	69 01	Addiere 1 zum Akkumulator-Inhalt – Direkter Adress-Modus
CMP#3	C9 03	Vergleiche den Wert im Akkumulator mit 3 – Direkter Adress-Modus
BNE LOOP	D0 FA	Springe im Falle der Ungleichheit zum Befehl mit der Schleifenmarke von LOOP – Relativadress-Modus
END		Steueranweisung – es wird kein Maschinencode erzeugt
SBC#1	E9 01	Subtrahiere 1 vom Wert im Akkumulator – Direkter Adress-Modus
CLC	18	Lösche das Carry-Bit
SEC	38	Setze das Carry-Bit auf 1

ÜBUNGEN

1. Programme in Assembler-Sprache werden mittels des zum Assembler Modul gehörenden _____ -Programms geschrieben.
2. Das _____ -Programm des Assembler Moduls dient zum Übersetzen der Assembler-Befehle in Maschinencode.
3. Programme in Maschinencode werden vom _____ -Programm des Assembler Moduls ausgeführt.
4. Durch Assemblieren des Quell-Programms entsteht das _____ -Programm.
5. Mit der Anweisung LDA #0 wird der Akkumulator mit dem Wert 0 geladen. Geben Sie die Bedeutung der folgenden Assembler-Anweisungen an.
   (a) LDA #$FF _____
   (b) ADC #10 _____
   (c) CMP #$1E _____

Die Übungen 6 bis 10 beziehen sich auf das folgende assemblierte Programm.

```
ASM
0000                    10          *=   $1000
1000  18                 20          CLC
1001  A9 01              30          LDA #1
1003  69 02              40  LOOP    ADC #2
1005  C9 03              50          CMP #9
1007  D0 FA              60          BNE LOOP
                        70  END
```

Maschinenprogramm Assembler-Programm

6. Welche Zahl gelangt zuerst in den Akkumulator? _____
7. Welche Zahl wird bei jedem Schleifendurchlauf addiert? _____
8. Welche Zahl befindet sich im Akkumulator, wenn das Programm zum letzten Mal durch die Schleife gelaufen ist? _____
9. Verfolgen Sie die Ausführung des Maschinencodes auf dem Schirm, so erscheinen dort _____ Akkumulator-Werte von _____ bis _____.
   gerade, ungerade
10. Die Zeilen 30 und 50 des Assembler-Programms können so geändert werden, daß auf dem Bildschirm gerade Akkumulator-Werte von 0 bis 8 erscheinen. Geben Sie die entsprechend geänderten Zeilen des Assembler-Programms an.
    30 _____
    50 _____

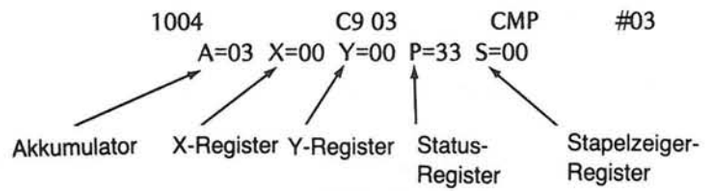
ANTWORTEN

1. **Writer/Editor-Programm**
2. **Assembler-Programm**
3. **Debugger-Programm**
4. **Objekt-Programm**
5. (a) Lade den Akkumulator mit FF (hex)
   (b) Addiere 10 zum Inhalt des Akkumulators (10 ist dezimal – kein $-Zeichen)
   (c) Vergleiche den Wert im Akkumulator mit 1 E (hex)
6. 1
7. 2
8. 9
9. Ungerade, 1, 9
10. 30 LDA #0
    50 CMP #8

Spezialregister und Adressierungsmodi

Der im Atari 400/800 verwendete Mikroprozessor 6502 besitzt einige Register, die ebenso wie Speicherzellen acht Bit enthalten (ein Byte), jedoch ganz bestimmten Zwecken dienen. Beim schrittweisen Durchgehen des Programms in Kapitel 4 (Verfolgen des Ablaufs) haben Sie bereits Dateninhalte dieser Register gesehen.

Beispiel:



Diese Spezialregister wollen wir in der unten angegebenen Reihenfolge besprechen.

1. Der Akkumulator: In ihm werden die meisten Operationen mit Daten ausgeführt. Das kurze Programm in Kapitel 4 zeigt die überaus lebhafte Tätigkeit dieses Registers.
2. Das X-Register: Es wird als schnelle Zwischenspeicherzelle oder als Index-Register in bestimmten Adressierungsmodi verwendet.
3. Das Y-Register: Es wird wie das X-Register benutzt.
4. Das Status-Register: Dieses Register enthält als Datensatz den jeweiligen Status des Mikroprozessors bei Ausführung eines Befehls. Jedes Bit des Registers enthält eine ganz bestimmte Status-Information.
5. Das Stapelzeiger-Register. In ihm befindet sich jeweils die Adresse der zuletzt gestapelten Information. Der Stapel ist ein besonderer Speicherblock mit den Adressen von 01FF bis zu 0100 (hex).

DER AKKUMULATOR

Die Programme in den Kapiteln 3 und 4 zeigten, wie der Akkumulator im direkten Adressmodus mit einer Zahl geladen werden kann. Eines der Programme addierte in diesem Modus eine Zahl und verglich diesen Wert mit 3 (wiederum im direkten Adressmodus). Für all diese Operationen wurden nur der Akkumulator und im Speicher befindliche Daten verwendet.
Es wurde auch eine Sprung-Anweisung gegeben (BNE im relativen Adressmodus). Um zu entscheiden, ob der Sprung ausgeführt werden soll oder nicht, befragte der Rechner das Status-Register durch Überprüfen eines Bits (Null-Bit) im Register.

DIE REGISTER X UND Y

Anhand eines kleinen Programms, das wir im folgenden entwickeln werden, wollen wir einige Vorgänge demonstrieren, die im Register X stattfinden. Unser Programm wird den Programmen in den Kapiteln 3 und 4 sehr ähnlich sein.

1. LDX #0 Lade das X-Register (LDX) im direkten Adressmodus mit Null (#).

2. INX Erhöhe den Wert im Register X (INX) mit dem implizierten Adressmodus (ohne Operand).

3. CPX #3 Vergleiche den Wert im Register X (CPX) im direkten Adressmodus mit 3 (#3).

4. BNE LOOP Springe, wenn das Ergebnis (Register X–3) *nicht* Null ist (BNE), zu der mit der *Marke* LOOP versehenen Anweisung zurück (INX in unserem Programm).

Das Programm wird wie zuvor mit Hilfe des Writer/Editor Programms des Assembler Moduls geschrieben. Wenn Sie im Edit-Modus sind, geben Sie NEW, um alle alten Programme, die sich eventuell noch im Edit-Puffer befinden, zu löschen. Der Speicherinhalt (das vorher assemblierte Programm) bleibt dabei erhalten. Geben Sie das Programm folgendermaßen ein.

```
EDIT
10   *=$1000
20   LDX #0
30 LOOP INX
40   CPX #3
50   BNE LOOP
60 END
■
```

← Ihre Eingabe

Sie gehen nun in das Assembler-Programm, um damit den entsprechenden Maschinencode zu erzeugen.

```
.
.
.
50   BNE LOOP          Sie tippen dies
60 END
ASM
0000                10          *=  $1000
1000 A200           20          LDX #0
1002 E8             30 LOOP     INX
1003 E003           40          CPX #3
1005 DOFB           50          BNE LOOP
                    60 END
EDIT
■
```

Führen Sie Ihre Eingabe mit dem Korrekturprogramm aus

```
.
.
.
1005 DOFB           50          BNE LOOP
                    60 END
EDIT
BUG                                  Sie tippen dies
DEBUG
G1000
1007                A=00 X=03 Y=00 P=33 S=00
DEBUG
■                   Ja, das X-Register endet mit
                    dem Wert 3
```

Verfolgen Sie jetzt das Programm Schritt für Schritt. Tippen Sie: T1000

Beachten Sie das jeweilige Ansteigen des X-Registers bei 1002

```
1000     A2   00            LDX  #$00
  A=00 X=00 Y=00 P=32 S=00
1002     E8                 INX
  A=00 X=01 Y=00 P=30 S=00
1003     E0   03            CPX  #$03
  A=00 X=01 Y=00 P=B0 S=00
1005     D0   FB            BNE  $1002   ← Der Assembler hat diese Sprungadresse berechnet.
  A=00 X=01 Y=00 P=B0 S=00
1002     E8                 INX
  A=00 X=02 Y=00 P=30 S=00
1003     E0   03            CPX  #$03
  A=00 X=02 Y=00 P=B0 S=00
1005     D0   FB            BNE  $1002
  A=00 X=02 Y=00 P=B0 S=00
1002     E8                 INX
  A=00 X=03 Y=00 P=30 S=00
1003     E0   03            CPX  #$03
  A=00 X=03 Y=00 P=33 S=00
1005     D0   FB            BNE  $1002
  A=00 X=03 Y=00 P=33 S=00
1007     00                 BRK
  A=00 X=03 Y=00 P=33 S=00
DEBUG
■
```

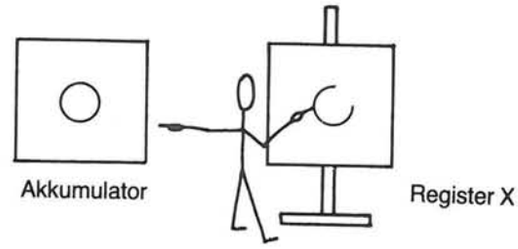
Ihr Register X sollte so aussehen. Die übrigen Register können einen anderen Inhalt haben.

Beachten Sie, daß sich der Wert im Register X jedesmal um 1 erhöht, wenn der Rechner den Befehl INX im Speicherplatz 1002 ausführt, und daß das Programm mit der Anweisung BRK (BREAK) bei 1007 anhält.
Der Rechner benutzt die BREAK-Anweisung, um sich selber am Ende eines Programms anzuhalten. In Kapitel 9 werden wir die BRK-Anweisung genauer durchnehmen.
Als nächstes wollen wir unser Programm so ändern, daß wir sowohl im Akkumulator wie auch im Register X zählen. Wir verwenden dabei eine neue Anweisung:

TAX

Dieser Buchstabencode ist die Abkürzung für: Übertrage den Inhalt des Akkumulators

in das Register X (Transfer Akkumulator to X register). Bei der Ausführung dieser Anweisung werden die momentan im Akkumulator befindlichen Daten in das Register X gebracht (kopiert). Der Inhalt des Akkumulators wird dabei nicht verändert.



Kopiere den Akkumulator in das Register X (TAX)

Drücken Sie die X- und dann die RETURN-Taste, um wieder in das Writer/Editor Programm zu gelangen. Sie können nun das alte Programm löschen mittels

NEW

Ganz wie in BASIC! Das alte Programm ist gelöscht und Sie können mit dem Schreiben des neuen beginnen.

EIN PROGRAMM, UM IM AKKUMULATOR UND IM REGISTER X ZU ZÄHLEN

```
EDIT
10   *=$1000
20   CLC
30   LDA #0
40   TAX          ←  Neuer Befehl, Übertrage den Inhalt des Akkumulators in das Register X
50 LOOP ADC #1    ←  Addiere 1 zum Akkumulator
60   INX          ←  Erhöhe das Register X um 1
70   CPX #3
80   BNE LOOP
90 END
■
```

Assemblieren Sie es dann:

```
.
.
.
80   BNE LOOP
90   END
ASM
0000                    10          *=      $1000
1000   18               20          CLC
1001   A900             30          LDA #0
1003   AA               40          TAX
1004   6901             50 LOOP     ADC #1
1006   E8               60          INX
1007   E003             70          CPX #3
1009   D0F9             80          BNE LOOP
                        90 END

EDIT
■
```

Geben Sie nun das Korrekturprogramm ein.

```
.
.
.
EDIT
BUG

DEBUG
■
```

Um zu sehen, was bei der Ausführung des Programms geschieht, ist die durch den Assembler (über T) gegebene Möglichkeit der schrittweisen Verfolgung des Ablaufs eine gewaltige Hilfe. Schauen wir also damit unserem Programm bei der Arbeit zu.
Tippen Sie: T1000 und drücken Sie die RETURN-Taste


```
1007   E0  03                   CPX   #$03
  A=01 X=01 Y=00 P=B0 S=00
1009   D0  F9                   BNE   $1004
  A=01 X=01 Y=00 P=B0 S=00
1004   69  01                   ADC   #$01
  A=02 X=01 Y=00 P=30 S=00
1006   E8                       INX
  A=02 X=02 Y=00 P=30 S=00
1007   E0  03                   CPX   #$03
  A=02 X=02 Y=00 P=B0 S=00
1009   D0  F9                   BNE   $1004
  A=02 X=02 Y=00 P=B0 S=00
1004   69  01                   ADC   #$01
  A=03 X=02 Y=00 P=30 S=00
1006   E8                       INX
  A=03 X=03 Y=00 P=30 S=00
1007   E0  03                   CPX   #$03
  A=03 X=03 Y=00 P=33 S=00
1009   D0  F9                   BNE   $1004
  A=03 X=03 Y=00 P=33 S=00
100B   00                       BRK
  A=03 X=03 Y=00 P=33 S=00
DEBUG
■
```

Was ist passiert? Der erste Teil des Programms ist so schnell an uns vorbeigeeilt, daß wir nicht mitlesen konnten. Der Schirm kann nicht mehr als 24 Zeilen auf einmal zeigen. Was nun?
Wenn Sie im Atari Assembler Handbuch nachlesen, finden Sie den Korrekturprogramm-Befehl (DEBUGGER):

SXXXX Einzelschritt-Ausführung

Wir wollen ihn ausprobieren. Die Gebrauchsanweisung im Manual verlangt die Eingabe von S und der Adresse des ersten Befehls. Tippen Sie S und drücken Sie die RETURN-Taste. Diese letzte Operation wiederholen Sie (mehrmals), um jeden einzelnen Schritt zu sehen. Auf geht's!

```
.
.
.
DEBUG
S1000
1000   18               CLC
  A=03 X=03 Y=00 P=32 S=00
DEBUG
■
```

Der Wert 3 ist vom letzten Programm übrig geblieben. NEW ändert nicht den Speicher, in dem das assemblierte Programm steht, auch nicht irgendwelche Register, sondern lediglich den Text Edit Puffer.

Der Rechner hält hier an, wartet aber nur darauf, daß Sie wieder S und RETURN geben. Also ...

```
.
.
.
.
.
S
1001   A9 00          LDA          #$00        Akkumulator
  A=00 X=03 Y=00 P=32 S=00                     auf Null gesetzt
DEBUG
S
1003   AA             TAX                      Register X
  A=00 X=00 Y=00 P=32 S=00                     auf Null gesetzt
DEBUG
S
1004   69 01          ADC          #$01        ← Eins zum
  A=01 X=00 Y=00 P=30 S=00                     Akkumulator
DEBUG                                          addiert
S
1006   E8             INX                      ← Register X
  A=01 X=01 Y=00 P=30 S=00                     um 1 erhöht
DEBUG
S
1007   E0 03          CPX          #$03
  A=01 X=01 Y=00 P=B0 S=00
DEBUG
S
1009   D0 F9          BNE          $1004
  A=01 X=01 Y=00 P=B0 S=00
DEBUG
S
1004   69 01          ADC          #$01        ← Noch einmal 1
  A=02 X=01 Y=00 P=30 S=00                     zum Akkumulator
DEBUG                                          addiert
S
1006   E8             INX                      ← Register X noch
  A=02 X=02 Y=00 P=30 S=00                     einmal um 1
                                               erhöht
                .
                .
                .
```



```
      Dies geht so weiter bis X = 03
                .
                .
                .
                .
                .
                .
1007   E0 03          CPX          #$03        ← Ist X = 3?
  A=03 X=03 Y=00 P=33 S=00
DEBUG                                            Ja
S
1009   D0 F9          BNE          $1003       ← Diesmal kein
  A=03 X=03 Y=00 P=33 S=00                       Sprung
DEBUG
S
100B   00             BRK
  A=03 X=03 Y=00 P=33 S=00
DEBUG
■
```

Damit haben wir Anweisungen des Akkumulators (A) und des Registers X gesehen. Da das Register Y genauso verwendet wird wie das Register X, übergehen wir es an dieser Stelle. Zweifellos werden Sie bemerkt haben, daß sich der Inhalt des Registers Y während des Ablaufs unserer Programme nicht geändert hat.

Bevor wir dies Programm verlassen, gehen wir noch einmal in das Writer/Editor Programm. Geben Sie X und RETURN, um dies auszuführen.

```
.
.
.
DEBUG
X

EDIT
■
```

Wir halten nochmals fest, daß auch Assembler-Programme mit dem Writer/Editor Programm aufgelistet werden können, geradeso wie man dies in BASIC mit LIST tut.

```
.
.
.
EDIT
LIST

10   *=$1000
20   CLC
30   LDA #0
40   TAX
50  LOOP ADC #1
60   INX
70   CPX #3
80   BNE LOOP
90  END
■
```

Diese durch das Writer/Editor Programm gegebene Möglichkeit kann sich als äußerst hilfreich erweisen, wenn man Änderungen am Programm vornehmen möchte, sei es, um Fehler zu korrigieren, oder auch aus anderen Gründen.

DAS STATUS-REGISTER

Dieses Register enthält sieben Bits, die Informationen über den Status des Mikroprozessors darstellen. Das achte Bit des Registers wird nicht verwendet. Die einzelnen Informationsbits heißen Flags oder auch Status-Bits. Diese Flags befinden sich jeweils in einem von zwei möglichen Zuständen: SET, oder auf Eins gesetzt; RESET, d.h. wieder auf Null gesetzt. Die Flags heißen: Carry, Zero result, Interrupt disable, Decimal mode, Break command, Overflow und Negative result. Abb. 5-1 zeigt die Positionen der entsprechenden Flags im Register.

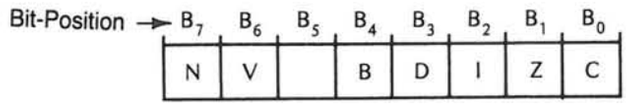


Abb. 5-1 Status-Register

wo : N = Negative result
V = Overflow
B = Break command
D = Decimal mode
I = Interrupt disable
Z = Zero result
C = Carry

B_0, Carry Bit. Dieses Bit ändert sich bei gewissen arithmetischen und logischen Operationen. Es kann auch durch entsprechende Befehle im Programm gesetzt (SET) bzw. gelöscht (RESET) werden. Wir werden es später anwenden.

B_1, Zero Bit. Dieses Flag wird automatisch gesetzt, wenn irgendein Datentransport oder irgend eine arithmetische Operation das Ergebnis Null hat. In den Kapiteln 4 und 5 wurde dieses Bit mittels BNE abgefragt (Springe, wenn das Ergebnis ungleich Null ist). Auf Seite 59 finden Sie ein Beispiel.

B_2, Interrupt disable Bit. Dieses Bit steuert die Wirkung des Interrupt request-Anschlusses des Computers. Vorläufig haben wir damit aber nichts zu tun.

B_3, Decimal mode Bit. Dieses Flag veranlaßt den Rechner, Additionen und Subtraktionen entweder binär oder dezimal auszuführen. Später mehr darüber.

B_4, Break command Bit. Dieses Bit wird nur vom Mikroprozessor gesetzt. Es wird bei Interrupts (Unterbrechungen) benutzt. Es erschien bei unserer Ausgabe jeweils am Ende der Programme, wenn wir die Ausführung in einzelnen Schritten oder mittels des T auf dem Schirm verfolgt hatten.

B_5, Expansion Bit. Dieses Bit wird gegenwärtig nicht benötigt. Es ist für eine Erweiterung des 6502 Mikrocomputers reserviert.

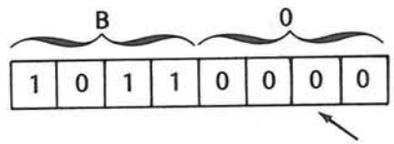
B_6, Overflow Bit. Dieses Bit zeigt einen eventuellen Überlauf bei einer binären arithmetischen Operation mit Vorzeichen an. Mehr darüber, wenn wir zu arithmetischen Operationen mit Vorzeichen kommen.

B_7, Negative Bit. Dieses Bit gibt Auskunft darüber, ob das Ergebnis einer arithmetischen Operation positiv oder negativ ist. Wir werden uns intensiv damit befassen, wenn wir zu den arithmetischen Operationen mit Zahlen, die mit Vorzeichen versehen sind, kommen.

Es sieht ganz so aus, als wollten wir eine ganze Reihe von Flag-Bits des Status-Registers auf später verschieben. Dem ist auch so! Wir werden jedes einzelne Flag-Bit dann besprechen, wenn es in einem unserer Programme verwendet wird. Das erste, dem Sie begegnet sind, ist das Zero Bit.
Wenn Sie in der Darstellung des Programmablaufs in Abb. 4-5 nachschauen, sehen Sie bei der ersten bzw. zweiten Ausführung des CPM *#3* Befehls in 1005:

```
Erste
Ausführung  → 1005  C9 03                    CMP   #$03
                  A=01 X=00 Y=00 P=B0 S=00
                .
                .
                .
Zweite          .
Ausführung  → 1005 C9 03                     CMP   #$03
                  A=02 X=00 Y=00 P=B0 S=00
```

Das Status-Register (P) enthält den Wert B0

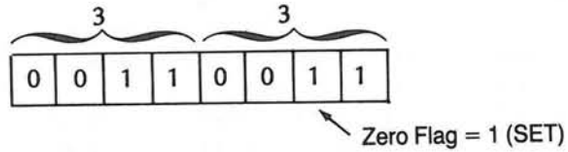


Das bedeutet, daß beim Vergleich des im Akkumulator befindlichen Wertes mit 3 das Ergebnis (1-3 oder 2-3) nicht gleich Null war.

Daher wurde der BNE-Sprung (Springe, wenn das Ergebnis nicht Null ist) ausgeführt. Bei der dritten Ausführung des gleichen Befehls jedoch zeigte der Schirm:

```
Dritte
Ausführung → 1005 C9 03                CMP   #$03
             A=03 X=00 Y=00 P=33 S=00
                                  ↑
                                  └ P hat sich geändert
```

Das Status-Register enthält jetzt 33.



Der Wert im Akkumulator (03) ist gleich 3.
Das Ergebnis (03-03) ist Null.

Der Sprung (BNE) wird *nicht* ausgeführt, und das Programm wird angehalten.

Schaut sich der Rechner gewisse Bits im Status-Register an, so kann er daraufhin, wie Sie sehen, in Abhängigkeit von bestimmten Bedingungen Entscheidungen treffen (etwa in welcher Reihenfolge Befehle ausgeführt werden), was die Möglichkeiten von Rechner und Programmierer außerordentlich vergrößert.

Einige Befehle verändern den Inhalt des Status-Registers, andere tun das nicht. In der folgenden Liste sind die bisher behandelten Befehle mit den durch sie betroffenen Status-Bits aufgeführt.

Anweisung	Betroffene Status-Flag
PLA	Z,C
LDA	N,Z
ADC	N,V,Z,C
STA	keine
RTS	keine
CLC	C
LDX	N,Z
TXA	N,Z
INX	N,Z
CPX	N,Z,C
BNE	keine
CMP	N,Z,C
SBC	N,V,Z,C
TAX	N,Z
SEC	C

Abb. 5-2 Wirkung von Befehlen auf Flag Bits

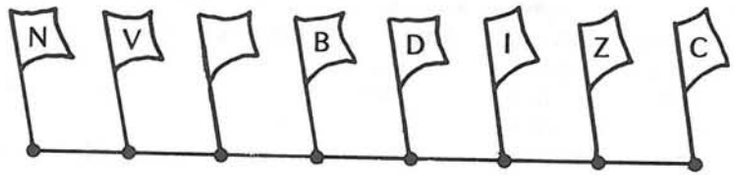
Die anschließende Tabelle zeigt alle bisher behandelten Befehle des 6502 mit den jeweils betroffenen Status-Flags. Dabei bedeutet X, daß das Flag-Bit betroffen ist. Das Ergebnis hängt vom Status ab, der sich aus der Durchführung des Befehls ergibt. Eine 1 bedeutet, daß das Bit gesetzt ist, eine 0, daß es gelöscht ist.

Buchstaben-code	*Durchgeführte Operation*	*Status Flags*							
		N	*V*		*B*	*D*	*I*	*Z*	*C*
ADC	Addiere den Inhalt des Speichers mit Carry zum Akkumulator	X	X					X	X
AND	AND Speicher mit Akkumulator	X						X	
ASL	Verschiebe ein Bit nach links (Speicher oder Akkumulator)	X						X	X
BCC	Springe, wenn kein Carry da (wenn C=0)								
BCS	Springe, wenn Carry da (wenn C=1)								
BEQ	Springe, wenn das Ergebnis Null ist (wenn Z=1)								
BIT	Vergleiche die Bits im Akkumulator mit dem Speicher	X	X					X	
BMI	Springe bei negativem Ergebnis (wenn N=1)								
BNE	Springe, wenn das Ergebnis ungleich Null ist (wenn Z=0)								
BPL	Springe bei positivem Ergebnis (wenn N=0)								
BRK	Force Break (Unterbrechung)						1		
BVC	Springe, wenn kein Überlauf (wenn V=0)								
BVS	Springe, wenn Überlauf (wenn V=1)								
CLC	Lösche das Carry-Flag								0
CLD	Lösche den Dezimalmodus					0			

Buchstaben-code	Durchgeführte Operation	Status Flags							
		N	V		B	D	I	Z	C
CLI	Lösche das Interrupt disable Flag						0		
CLV	Lösche das Overflow-Flag		0						
CMP	Vergleiche Speicher und Akkumulator	X						X	X
CPX	Vergleiche Speicher und Index X	X						X	X
CPY	Vergleiche Speicher und Index Y	X						X	X
DEC	Vermindere den Speicherinhalt um eins							X	X
DEX	Vermindere den Index X um eins							X	X
DEY	Vermindere den Index Y um eins							X	X
EOR	Ausschließlich OR (oder) Speicher mit Akkumulator							X	X
INC	Erhöhe den Speicherinhalt um eins							X	X
INX	Erhöhe den Index X um eins							X	X
INY	Erhöhe den Index Y um eins							X	X
JMP	Springe zur neuen Adresse								
JSR	Springe zur neuen Adresse, merke Rückkehradresse								
LDA	Lade den Akkumulator mit dem Speicherinhalt							X	X
LDX	Lade den Index X mit dem Speicherinhalt							X	X
LDY	Lade den Index Y mit dem Speicherinhalt							X	X
LSR	Verschiebe ein Bit nach rechts (Speicher oder Akkumulator)	0						X	X
NOP	Keine Operation								
ORA	OR Speicher mit Akkumulator	X						X	
PHA	Bringe Akkumulator in den Stapelspeicher								
PHP	Bringe Statusregister in den Stapelspeicher								
PLA	Hole Akkumulator vom Stapelspeicher							X	X
PLP	Hole Statusregister vom Stapelspeicher	X	X	X	X	X	X	X	X
ROL	Rotiere ein Bit nach links (Speicher oder Akkumulator)	X						X	X
ROR	Rotiere ein Bit nach rechts (Speicher oder Akkumulator)	X						X	X
RTI	Rückkehr vom Interrupt	X	X	X	X	X	X	X	X
RTS	Kehre vom Unterprogramm zurück								
SBC	Subtrahiere den Speicher und „borge" vom Akkumulator (negativer Übertrag)	X	X					X	X
SEC	Setze das Carry Flag								1
SED	Setze den Dezimalmodus					1			
SEI	Setze das Interrupt disable Flag						1		
STA	Bringe den Inhalt des Akkumulators in den Speicher								
STX	Bringe den Index X in den Speicher								
STY	Bringe den Index Y in den Speicher								
TAX	Übertrage den Akkumulator in das Indexregister X	X						X	
TAY	Übertrage den Akkumulator in das Indexregister Y	X						X	
TSX	Übertrage den Stapelkopf in das Indexregister X	X						X	
TXA	Übertrage Index X in den Akkumulator	X						X	
TXS	Übertrage Index X in den Stapelkopf								
TYA	Übertrage Index Y in den Akkumulator	X						X	

Flag-Abkürzungen:

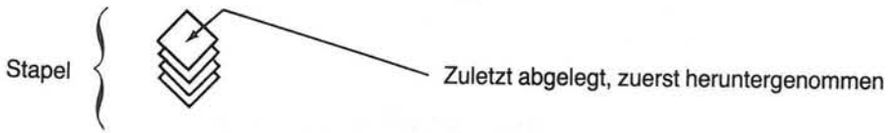
N	Negative result flag	D	Decimal mode flag
V	Overflow flag	I	Interrupt disable flag
	Expansion flag (nicht markiert)	Z	Zero result flag
B	Break command flag	C	Carry flag



Die Zustandsbits sind auf 1 gesetzt!

DAS STAPELSPEICHER-REGISTER

Manchmal muß der in einem Register befindliche Wert zwischengespeichert werden, um das Register anderweitig einsetzen zu können, wonach der ursprüngliche Wert wieder in das Register zurückgebracht wird. Der Mikroprozessor verwendet dazu einen Teil des RAM-Speichers, der wegen der besonderen Art seiner Verwendung *Stapel* genannt wird. Die in diesem Speicher befindlichen Werte sind wie die Karten eines Kartenstapels abgelegt, wobei jeder Karte ein Wert entspricht. Jeder neu hinzukommende Wert wird oben auf den Stapel gelegt. Genauso werden die Werte, wenn sie wieder gebraucht werden, von oben vom Stapel nacheinander heruntergenommen. Immer ist der zuletzt abgelegte Wert der zuerst wieder verwendete.

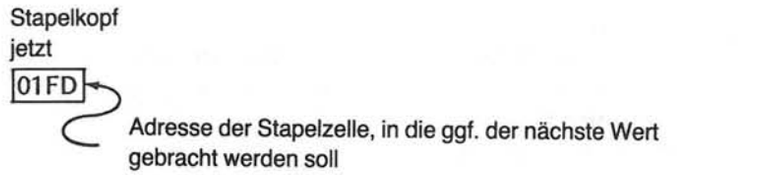


Das Stapelspeicher-Register ist ein 16-Bit Register, das die Adresse des Stapelkopfes (oberste Stapelzelle) führt.

Beispiel:
Der zuletzt gespeicherte Wert befindet sich in der Zelle 01FF. Wenn zwei weitere Werte auf den Stapel gebracht werden, enthält das Stapelspeicher-Register den Wert 01FD.

Stapelkopf
ursprünglich
01FF

01FD	?	← Nächster Wert hierhin
01FE	65	← Zweiter gestapelter Wert
01FF	10	← Erster gestapelter Wert



Der Vorgang verläuft umgekehrt, wenn dem Stapel Werte entnommen werden. Der Register-Wert wird automatisch um eins erhöht, enthält danach die Adresse 01FE, und der Wert 65 wird vom Stapel genommen. Soll der Wert 03 vom Stapel genommen werden, wird der Register-Wert abermals um eins erhöht und 03 heruntergenommen. Das Register enthält dann wieder die ursprüngliche Adresse 01FF.
Dem Programmierer stehen vier Maschinenbefehle zur Verfügung, um Daten zu stapeln oder vom Stapel herunterzuholen. Es können Daten vom Akkumulator oder vom Status-Register gespeichert und wieder geholt werden.

1. PHA (Bringe Akkumulator-Inhalt auf Stapel)
Op Code = 48
Implizierter Modus

2. PHP (Bringe P-Inhalt auf Stapel)
Op Code = 08
Implizierter Modus

3. PLA (Hole Akkumulator-Inhalt vom Stapel)
Op Code = 68
Implizierter Modus

4. PLP (Hole P-Inhalt vom Stapel)
Op Code = 28
Implizierter Modus

ADRESSIERUNGS-MODI

Der Mikroprozessor 6502 verwendet verschiedene Adressierungs-Modi. Einige Befehle werden in nur einem Modus verwendet, andere in mehr als einem. Bisher haben wir nur drei Modi kennengelernt: den direkten, den implizierten und den relativen Adressierungs-Modus.
Die Adressierungs-Modi kann man in zwei Klassen einteilen: indiziert und nicht-indiziert. Wenden wir uns zunächst den nicht-indizierten Adressierungs-Modi zu, da sie am leichtesten zu verstehen und anzuwenden sind.

Der implizierte Adressierungs-Modus

Befehle, die mit dem implizierten Modus arbeiten, sind ein Byte lang. Dieses Byte enthält den Operations Code, der eine rechnerinterne Operation bezeichnet. Es gibt keinen Operanden. In diesem Kapitel verwendete Beispiele sind INX und TAX.

Seite 74 INX (Erhöhe das Register X um eins)
Op Code E8
Betroffene Flag Bits: N und Z

Seite 76 TAX (Übertrage den Akkumulator in das Register X)
Op Code AA
Betroffene Flag Bits: N und Z

Die Operation, die zu einem im implizierten Adressierungs-Modus gegebenen Befehl gehört, wird durch den Operations-Code vollständig definiert. Daher ist für die Beschreibung dieser Operation nur ein Byte nötig. Befehle, die im implizierten Adressierungs-Modus arbeiten, können in *keinem* anderen Modus verwendet werden.

Der direkte Adressierungs-Modus

Befehle, die im direkten Adressierungs-Modus arbeiten, benötigen für die Beschreibung der zugehörigen Operation zwei Bytes. Das erste Byte enthält den Op Code, der die Operation *und* den Adressierungs-Modus spezifiziert. Das zweite Byte enthält einen bei Abfassung des Programms bekannten konstanten Wert. Die direkte Eingabe solcher Werte erspart es dem Programmierer, diese erst in den Speicher zu laden und sie dann bei Bedarf wieder aus dem Speicher zu holen.
In den Kapiteln 3, 4 und 5 haben Sie die folgenden Befehle im direkten Adressierungs-Modus verwendet.

Seite 42 CPX (Vergleiche mit Register X)
Op Code E0 (bei direkter Adressierung)
Zweites Byte 0A (Wert von X wird damit verglichen)
Betroffene Status-Flags: N, Z und C

Seite 42 LDX (Lade Register X)
Op Code A2 (bei direkter Adressierung)
Zweites Byte 00 (in X geladener Wert)
Betroffene Status-Flags: Z und C

Seite 57 LDA (Lade Akkumulator)
Op Code A9 (bei direkter Adressierung)
Zweites Byte 00 (geladener Wert)
Betroffene Status-Flags: N und Z

Seite 57 ADC (Addiere mit carry zum Akkumulator)
Op Code 69 (bei direkter Adressierung)
Zweites Byte 01 (addierter Wert)
Betroffene Status-Flags: N, Z, C und V

Seite 57 CMP (Vergleiche mit Akkumulator)
Op Code C9 (bei direkter Adressierung)
Zweites Byte 03 (Wert, der mit dem Akku verglichen wird)
Betroffene Status-Flags: N, Z und C

Die direkte Adressierung ist der einfachste Weg zur Behandlung von Konstanten. Alle in diesem Modus arbeitenden Befehle können auch in anderen Modi verwendet werden. Der Operationscode des Befehls ist für jeden Modus, in dem er verwendet werden soll, verschieden; der Rechner erkennt also am Code, welcher Modus verlangt ist.

Der relative Adressierungs-Modus

Alle Verzweigungs- bzw. Sprung-Befehle arbeiten ausschließlich in diesem Modus. Es handelt sich um zwei Byte lange Anweisungen. Der Op Code steht im erste Byte. Das zweite Byte enthält eine mit Vorzeichen versehene Zahl, die die Sprungweite (im Falle der Ausführung des Sprungs) spezifiziert. Das zweite Byte wird ignoriert, falls die Bedingungen für eine Ausführung des Sprunges nicht erfüllt sind. Zu dem Zeitpunkt, in dem entschieden wird, ob der Sprung ausgeführt wird oder nicht, zeigt der Befehlszähler auf die nächste Anweisung. Sie haben bisher einen Befehl im relativen Adressierungs-Modus verwendet: BNE.

```
Seite 60    BNE    (Springe, wenn das Ergebnis ungleich Null ist)
                   Operationscode D0
                   Zweites Byte FA (equivalent zu -6: 6 Adressen zurück)
                   Betroffene Flag Bits: keine
```

Die Ausführung oder Unterlassung von Sprüngen bzw. Verzweigungen hängt vom Status der Flag Bits im Status-Register ab (N,V,Z oder C). Lediglich Sprung- oder Verzweigungsanweisungen arbeiten in diesem Modus.

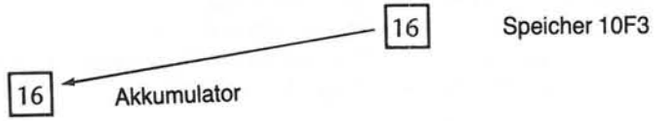
Der absolute Adressierungs-Modus

Anweisungen in diesem Modus haben eine Länge von drei Bytes. Eines für den Operationscode, die anderen beiden für den Adress-Operanden. Das niederwertige Adress Byte steht im zweiten, das höherwertige im dritten Byte. Der Programmierer kann so eine Adresse voller Länge, d. h. 16 Bit, spezifizieren und somit jeden Speicherplatz erreichen. In Kapitel 2 haben Sie in diesem Modus mit einer STORE-Anweisung gearbeitet. Im nächsten Kapitel wird dieser Modus abermals verwendet.

Beispiel:

LDA (Lade den Akkumulator)
Op Code AD (bei absoluter Adressierung)
Zweites Byte F3 (niedriges Adress-Byte)
Drittes Byte 10 (hohes Adress-Byte)
Betroffene Status-Flags: N und Z

In diesem Beispiel wird der Wert, der in der Speicherzelle 10F3 enthalten ist, in den Akkumulator gebracht. Der in 10F3 enthaltene Wert wird dabei nicht geändert.



Der Null-Seiten (Zero Page) Adressierungs-Modus

Befehle in diesem Modus sind zwei Byte lang. Das erste Byte enthält den Op Code, das zweite das niederwertige Byte des Adress-Operanden. Das höhere Adress Byte wird vom Rechner so behandelt, als enthielte es Null (daher der Name). Zum Zero Page Teil des Speichers gehören die Adressen von 0000 bis 00FF. Jeder nachfolgende Block

von 256 Speicherplätzen heißt Page („Seite") des Speichers. Der Vorteil von Zero Page-Befehlen liegt in der Einsparung eines Bytes im Befehl. Die Ausführungszeit für den Befehl ist daher kürzer als bei absoluter Adressierung. Der Benutzer sollte seine Speicherbelegung so organisieren, daß sich die am häufigsten benötigten Daten im Zero Page Teil des Speichers befinden (auf der Seite 0 stehen). Dies läßt sich nicht immer durchführen, da die Hersteller sehr häufig gerade diesen Teil des Speichers für ihr Betriebssystem verwenden. Bislang haben wir noch keinen Befehl in diesem Modus verwendet.

Beispiel:

LDA (Lade den Akkumulator)
Op Code A5 (Bei Zero Page Adressierung)
Zweites Byte 80 (niederwertiges Adress Byte)
Betroffene Flag Bits: N und Z

In diesem Beispiel wird der Akkumulator mit dem Inhalt des Speicherplatzes 0080 geladen.
Wir werden die Diskussion der indizierten Adressierungs-Modi bis zum Kapitel 8, wo sie zum ersten Mal verwendet werden, aufschieben.

BEFEHLSLISTE MIT ADRESSIERUNGS-MODI

Buchstaben Code	*Op Codes*												
	Akkumulator	*Unmittelbar*	*Null-Seite*	*Null-Seite, X*	*Null-Seite, Y*	*Absolut*	*Absolut, X*	*Absolut, Y*	*Impliziert*	*Relativ*	*Indiziert indirekt*	*Indirekt indiziert*	*Indirekt*
ADC	–	69	65	75	–	6D	7D	79	–	–	61	71	–
AND	–	29	25	35	–	2D	3D	39	–	–	21	31	–
ASL	0A	–	06	16	–	0E	1E	–	–	–	–	–	–
BCC	–	–	–	–	–	–	–	–	–	90	–	–	–
BCS	–	–	–	–	–	–	–	–	–	B0	–	–	–
BEQ	–	–	–	–	–	–	–	–	–	F0	–	–	–
BIT	–	–	24	–	–	2C	–	–	–	–	–	–	–
BMI	–	–	–	–	–	–	–	–	–	30	–	–	–
BNE	–	–	–	–	–	–	–	–	–	D0	–	–	–
BPL	–	–	–	–	–	–	–	–	–	10	–	–	–
BRK	–	–	–	–	–	–	–	–	00	–	–	–	–
BVC	–	–	–	–	–	–	–	–	–	50	–	–	–
BVS	–	–	–	–	–	–	–	–	–	70	–	–	–
CLC	–	–	–	–	–	–	–	–	18	–	–	–	–
CLD	–	–	–	–	–	–	–	–	D8	–	–	–	–
CLI	–	–	–	–	–	–	–	–	58	–	–	–	–
CLV	–	–	–	–	–	–	–	–	B8	–	–	–	–

Buchstaben Code	Op Codes												
	Akkumulator	*Unmittelbar*	*Null-Seite*	*Null-Seite, X*	*Null-Seite, Y*	*Absolut*	*Absolut, X*	*Absolut, Y*	*Impliziert*	*Relativ*	*Indiziert indirekt*	*Indirekt indiziert*	*Indirekt*
CMP	–	C9	C5	D5	–	CD	DD	D9	–	–	C1	D1	–
CPX	–	E0	E4	–	–	EC	–	–	–	–	–	–	–
CPY	–	C0	C4	–	–	CC	–	–	–	–	–	–	–
DEC	–	–	C6	D6	–	CE	DE	–	–	–	–	–	–
DEX	–	–	–	–	–	–	–	–	CA	–	–	–	–
DEY	–	–	–	–	–	–	–	–	88	–	–	–	–
EOR	–	49	45	55	–	4D	5D	59	–	–	41	51	–
INC	–	–	E6	F6	–	EE	FE	–	–	–	–	–	–
INX	–	–	–	–	–	–	–	–	E8	–	–	–	–
INY	–	–	–	–	–	–	–	–	C8	–	–	–	–
JMP	–	–	–	–	–	4C	–	–	–	–	–	–	6C
JSR	–	–	–	–	–	20	–	–	–	–	–	–	–
LDA	–	A9	A5	B5	–	AD	BD	B9	–	–	A1	B1	–
LDX	–	A2	A6	–	B6	AE	–	BE	–	–	–	–	–
LDY	–	A0	A4	B4	–	AC	BC	–	–	–	–	–	–
LSR	4A	–	46	56	–	4E	5E	–	–	–	–	–	–
NOP	–	–	–	–	–	–	–	–	EA	–	–	–	–
ORA	–	09	05	15	–	0D	1D	19	–	–	01	11	–
PHA	–	–	–	–	–	–	–	–	48	–	–	–	–
PHP	–	–	–	–	–	–	–	–	08	–	–	–	–
PLA	–	–	–	–	–	–	–	–	68	–	–	–	–
PLP	–	–	–	–	–	–	–	–	28	–	–	–	–
ROL	2A	–	26	36	–	2E	3E	–	–	–	–	–	–
ROR	6A	–	66	76	–	6E	7E	–	–	–	–	–	–
RTI	–	–	–	–	–	–	–	–	40	–	–	–	–
RTS	–	–	–	–	–	–	–	–	60	–	–	–	–
SBC	–	E9	E5	F5	–	ED	FD	F9	–	–	E1	F1	–
SEC	–	–	–	–	–	–	–	–	38	–	–	–	–
SED	–	–	–	–	–	–	–	–	F8	–	–	–	–
SEI	–	–	–	–	–	–	–	–	78	–	–	–	–
STA	–	–	85	95	–	8D	9D	99	–	–	81	91	–
STX	–	–	86	–	96	8E	–	–	–	–	–	–	–
STY	–	–	84	94	–	8C	–	–	–	–	–	–	–
TAX	–	–	–	–	–	–	–	–	AA	–	–	–	–
TAY	–	–	–	–	–	–	–	–	A8	–	–	–	–
TSX	–	–	–	–	–	–	–	–	BA	–	–	–	–
TXA	–	–	–	–	–	–	–	–	8A	–	–	–	–
TXS	–	–	–	–	–	–	–	–	9A	–	–	–	–
TYA	–	–	–	–	–	–	–	–	98	–	–	–	–

ZUSAMMENFASSUNG

Die Flexibilität des im Atari 400/800 verwendeten Mikroprozessors 6502 wird durch verschiedene Adressierungs-Modi und besondere Register erreicht. Die in diesem Kapitel behandelten Register sind:

- Akkumulator (Register A) - Die meisten an Daten vorgenommenen Operationen finden in diesem Register statt.
- Index-Register X - Wird als Zwischenspeicher oder in gewissen Adressierungs-Modi als Index benutzt.
- Index-Register Y - Hat die gleiche Funktion wie das Register X.
- Status-Register P (Register P) - Enthält bei Ausführung eines Befehls den Status des Rechners.
- Stapelspeicher-Register (Register S) - Enthält die Adresse des Stapelkopfes.

Die besprochenen Adressierungs-Modi sind:

- Implizierter Modus (implied mode) - Ein-Byte-Befehle, die keinen Operanden benötigen.
- Direkter Modus (immediate) - Das erste Byte spezifiziert die Operation; der Operand (zweites Byte) ist ein Datensatz von der Länge 1 Byte.
- Relativer Modus (relative) - Wird für Verzweigungen bzw. Sprünge verwendet. Das erste Byte spezifiziert die Art der Verzweigung; der ein Byte lange Operand enthält die Sprungweite und Sprungrichtung.
- Absoluter Modus (absolute) - Ein Drei-Byte-Befehl; das erste Byte spezifiziert die Operation. Die beiden anderen Bytes enthalten den Operanden, der hier eine benötigte Adresse ist.
- Zero Page Modus - ähnlich dem absoluten Adressierungs-Modus, jedoch nur mit einem ein Byte langen Operanden als benötigte Adresse.

ÜBUNGEN

1. Wie viele Bits an Informationen können im Akkumulator enthalten sein? _____
2. Nenne die beiden Index-Register.
   _____ und _____
3. In welchem Register sind die Status-Flags enthalten?
   _____ _____ Register
4. Das Register, in welches Daten geladen, von dem aus Daten abgespeichert und in dem arithmetische Operationen ausgeführt werden, heißt: _____

Die Fragen 5-8 beziehen sich auf das folgende Programm ...

```
10   *=$1050
20   LDX #0
30 LOOP INX
40   CPX #2
50   BNE LOOP
60 END
```

5. Welches Register wird im Programm verwendet? _____
6. Welches Status-Flag wird benutzt, um über die Ausführung eines Sprungs zu entscheiden? _____
7. Welche Werte erscheinen bei Ausführung des Programms im Index-Register?
_____ _____ _____
8. Wie heißt nach dem Assemblieren die Startadresse für das Maschinenprogramm?
_____
9. Der Writer/Editor Befehl des Assembler Moduls zur Ausgabe des Maschinenprogramms auf dem Bildschirm lautet: _____
(NEW, LIST, RUN)
10. Die Bytes des Status-Registers sind folgendermaßen markiert:

N	V		B	D	I	Z	C

(a) Bit B_1 ist das _____ -Flag Bit
(b) Bit B_7 ist das _____ -Flag Bit
(c) Das Carry-Flag-Bit hat die Marke _____

11. Bei Verwendung des Stapelspeichers enthält das Stapelspeicher-Register die Adresse des Stapelkopfes. Der zuletzt gestapelte Datensatz ist der _____
zuerst, zuletzt
vom Stapel geholte Datensatz.
12. Wie viele Bytes benötigt ein Befehl im implizierten Adressierungs-Modus? ____
13. Welcher Adressierungs-Modus wird bei Verzweigungen benutzt? _____
14. Welcher Befehl ist kürzer (weniger Bytes):
Absolut oder Zero Page? _____

ANTWORTEN

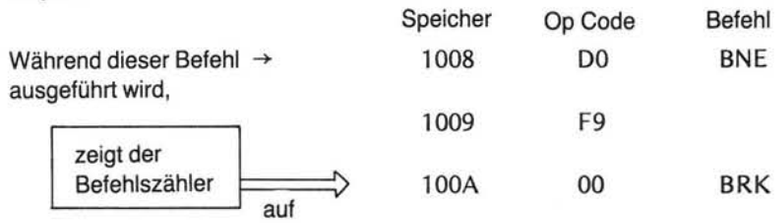
1. 8 bits
2. X und Y
3. Status Register
4. Der Akkumulator (oder Register A)
5. X
6. Zero (oder Zero Flag)

7. 0,1,2 (oder 00000000,00000001,00000010 binär)
8. 1050 (hex)
9. LIST
10. (a) Z (oder Zero Flag)
(b) N (oder Negativ Flag)
(c) B_0
11. zuerst
12. 1 (Eins)
13. Relativ
14. Zero Page

Verzweigungen

In den Kapiteln 3, 4 und 5 haben wir zur Ausführung einer Schleife einen Sprungbefehl (BNE) verwendet. Sprungbefehle sind Zwei-Byte-Anweisungen im relativen Adressierungs-Modus, was besagt, daß die Adresse des Sprungzieles *relativ* zum momentanen Stand des *Befehlszählers* (Program counter) berechnet wird. Der Befehlszähler dient als Zeiger für die Adresse, unter der nach Ausführung des gerade akuten Befehls der Befehl steht, der als nächster ausgeführt werden soll. Er ist der Befehlsausführung des Rechners immer einen Schritt voraus.

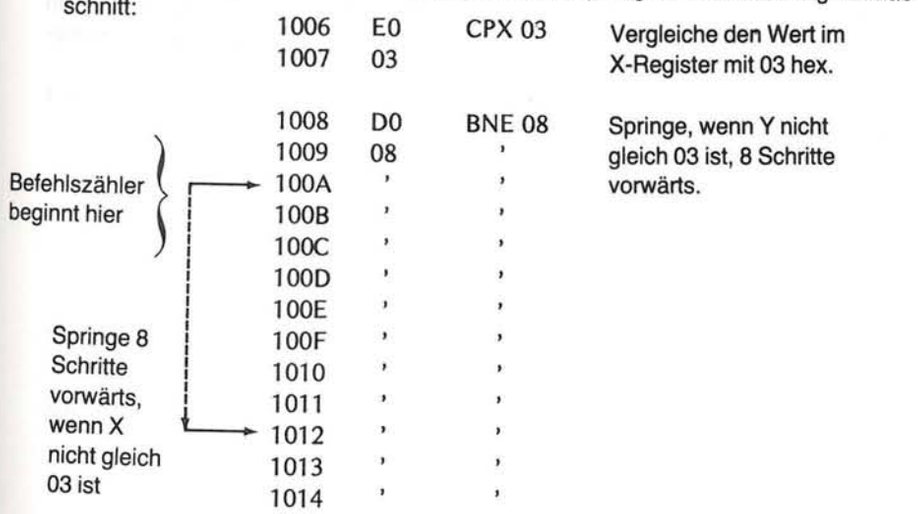
Beispiel:



Daher wird der Sprung relativ zum Befehlszählerstand ausgeführt (oder einem Speicherplatz nach dem zweiten Byte der Sprunganweisung). Der Operand (in diesem Beispiel das zweite Byte-F9) der Sprunganweisung gibt die Richtung und Weite des Sprunges an.
Bei der Verwendung als Operand in einem Sprungbefehl wie BNE verursachen alle Hex-Werte von 01 bis 7F, vom Befehlszählerstand aus gerechnet, einen Vorwärtssprung. Der folgende Befehl würde einen Vorwärtssprung von der Speicherzelle 100 A (die vom Befehlszähler angezeigte Adresse bei Ausführung des BNE Befehls) zur Speicherzelle 1012 (100A+8) (hexadezimal) zur Folge haben.

```
1008    D0        BNE    08
1009    08
```


Es folgt ein Beispiel für die Anwendung des oben gesagten in einem Programmabschnitt:

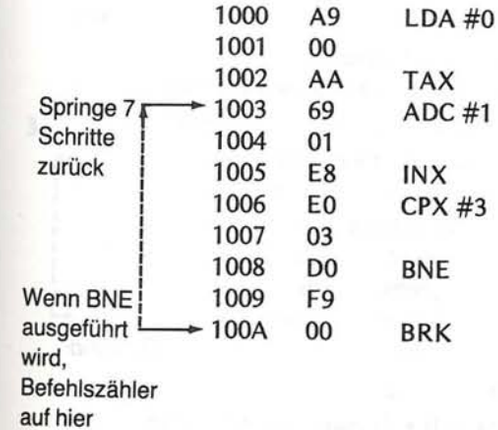


Alle Hex-Werte von 80 bis FF werden von Sprungbefehlen als Rückwärtssprünge verwendet (auch Negativsprung). Im „Zähle im Akkumulator"-Programm (Kapitel 5) lautet der verwendete Befehl:

```
1008    D0        BNE LOOP    Der Assembler suchte sich die Marke LOOP
                              und zählte rückwärts, um sie zu finden.
1009    F9                    Springe, wenn X nicht gleich 03 ist, 7 Schritte
                              zurück.
```

Es ist ein Rückwärtssprung (oder Sprung in negativer Richtung), da F9 zwischen 80 und FF liegt. Zählt man von der Adresse 100A sieben Schritte zurück, so ergibt sich als Sprungziel die Adresse 1003, d.h. der Anfang der Schleife.

„ZÄHLE IM AKKUMULATOR"-PROGRAMM



Wir wollen hier nicht in die vom Rechner verwendete Methode zur Bestimmung der Werte negativer Zahlen einsteigen. Stattdessen geben wir Tabellen an, aus denen man den beim Sprung verwendeten Operanden bestimmen kann. Der Assembler erledigt die Details, wenn Sie ihm die geeigneten Marken geben.
Wir wollen noch einmal festhalten, daß der Operand bei Vorwärtssprüngen zwischen 1 und 7F, bei Rückwärtssprüngen zwischen 80 und FF liegt.

BEISPIELE MIT VORWÄRTSSPRÜNGEN

1.
```
      100E   D0   BNE 07
      100F   07
  ┌─► 1010
  │     .
  │     .
  └─► 1017
```
Befehlszähler beginnt bei 1010
Gewünschter Sprung nach 1017 (7 Schritte)

Sieh in der Tabelle nach:

Vorwärts-schritte (dezimal)	Sprung-operand (hex)	
7	07	← Operand

Wenn das Prüfergebnis nicht gleich Null ist,
springe vorwärts nach 1017 (1010 + 7 Schritte).

2.
```
      1010   D0   BNE 1F
      1009   1F
  ┌─► 1010    .
  │     .     .
  │     .     .
  └─► 102F    .
```
Befehlszähler beginnt bei 1010
Gewünschter Sprung nach 102F
(31 Dezimalschritte)

Sieh in der Tabelle nach:

Vorwärts-schritte (dezimal)	Sprung-operand (hex)	
31	1F	← Operand

Wenn das Prüfergebnis nicht gleich Null ist,
springe vorwärts nach 102F (1010 + 1F Hexadezimalschritte)

3.
```
      100E   D0   BNE 77
      100F   77
  ┌─► 1010    .
  │     .     .
  │     .     .
  └─► 1087    .
```
Befehlszähler beginnt bei 1010
Gewünschter Sprung nach 1087
(119 Dezimalschritte)

Sieh in der Tabelle nach:

Vorwärts-schritte (dezimal)	Sprung-operand (hex)	
119	77	← Operand

Wenn das Prüfergebnis nicht gleich Null ist,
springe vorwärts nach 1087 (1010 + 77 Hexadezimalschritte).

4.
```
      100E   D0   BNE 5B
      100F   5B
  ┌─► 1010    .
  │     .     .
  │     .     .
  └─► 106B    .
```
Befehlszähler beginnt bei 1010
Gewünschter Sprung nach 106B
(91 Dezimalschritte)

Sieh in der Tabelle nach:

Vorwärts-schritte (dezimal)	Sprung-operand (hex)	
91	5B	← Operand

Wenn das Prüfergebnis nicht gleich Null ist,
springe vorwärts nach 106B (1010 + 5B Hexadezimalschritte).

BEISPIELE MIT RÜCKWÄRTSSPRÜNGEN

1.
```
  ┌─► 100B
  │     .
  │     .
  │   1010   D0   BNE F9
  │   1011   F9
  └─ 1012    .
        .    .
```
Befehlszähler beginnt bei 1012
Gewünschter Sprung nach 100B (-7 Schritte)

Sieh in der Tabelle nach:

Rückwärts-schritte (dezimal)	Sprung-operand (hex)	
7	F9	← Operand

Wenn das Prüfergebnis nicht gleich Null ist,
springe rückwärts nach 100B (1012 - 7 Schritte).

2.
```
  ┌─► OFF3
  │     .
  │     .
  │   1010   D0   BNE E1
  │   1011   E1
  └─ 1012    .
        .
```
Befehlszähler beginnt bei 1012
Gewünschter Sprung nach FF3
(-31 Dezimalschritte)

Sieh in der Tabelle nach:

Rückwärts-schritte (dezimal)	Sprung-operand (hex)	
31	E1	← Operand

Wenn das Prüfergebnis nicht gleich Null ist,
springe rückwärts nach FF3 (1012 - 1F Hexadezimalschritte)

3.
```
  ┌─► 0F9B
  │     .
  │     .
  │   1010   D0   BNE 89
  │   1011   89
  └─ 1012    .
        .    .
```
Befehlszähler beginnt bei 1012
Gewünschter Sprung nach F9B
(-119 Dezimalschritte)

Sieh in der Tabelle nach:

Rückwärts-schritte (dezimal)	Sprung-operand (hex)	
119	89	← Operand

Wenn das Prüfergebnis nicht gleich Null ist,
springe rückwärts nach F9B (1012 - 77 Hexadezimalschritte).

TABELLE ZUR BESTIMMUNG DER VORWÄRTSSPRÜNGE

Vorwärts-schritte (dezimal)	Sprung-operand (hex)	Vorwärts-schritte (dezimal)	Sprung-operand (hex)	Vorwärts-schritte (dezimal)	Sprung-operand (hex)
1	01	49	31	97	61
2	02	50	32	98	62
3	03	51	33	99	63
4	04	52	34	100	64
5	05	53	35	101	65
6	06	54	36	102	66
7	07	55	37	103	67
8	08	56	38	104	68
9	09	57	39	105	69
10	0A	58	3A	106	6A
11	0B	59	3B	107	6B
12	0C	60	3C	108	6C
13	0D	61	3D	109	6D
14	0E	62	3E	110	6E
15	0F	63	3F	111	6F
16	10	64	40	112	70
17	11	65	41	113	71
18	12	66	42	114	72
19	13	67	43	115	73
20	14	68	44	116	74
21	15	69	45	117	75
22	16	70	46	118	76
23	17	71	47	119	77
24	18	72	48	120	78
25	19	73	49	121	79
26	1A	74	4A	122	7A
27	1B	75	4B	123	7B
28	1C	76	4C	124	7C
29	1D	77	4D	125	7D
30	1E	78	4E	126	7E
31	1F	79	4F	127	7F
32	20	80	50		
33	21	81	51		
34	22	82	52		
35	23	83	53		
36	24	84	54		
37	25	85	55		
38	26	86	56		
39	27	87	57		
40	28	88	58		
41	29	89	59		
42	2A	90	5A		
43	2B	91	5B		
44	2C	92	5C		
45	2D	93	5D		
46	2E	94	5E		
47	2F	95	5F		
48	30	96	60		

Abb. 6-1 Vorwärtssprünge

TABELLE ZUR BESTIMMUNG DER RÜCKWÄRTSSPRÜNGE

Rückwärts schritte (dezimal)	Sprung-operand (hex)	Rückwärts-schritte (dezimal)	Sprung-operand (hex)	Rückwärts-schritte (dezimal)	Sprung-operand (hex)
1	FF	49	CF	97	9F
2	FE	50	CE	98	9E
3	FD	51	CD	99	9D
4	FC	52	CC	100	9C
5	FB	53	CB	101	9B
6	FA	54	CA	102	9A
7	F9	55	C9	103	99
8	F8	56	C8	104	98
9	F7	57	C7	105	97
10	F6	58	C6	106	96
11	F5	59	C5	107	95
12	F4	60	C4	108	94
13	F3	61	C3	109	93
14	F2	62	C2	110	92
15	F1	63	C1	111	91
16	F0	64	C0	112	90
17	EF	65	BF	113	8F
18	EE	66	BE	114	8E
19	ED	67	BD	115	8D
20	EC	68	BC	116	8C
21	EB	69	BB	117	8B
22	EA	70	BA	118	8A
23	E9	71	B9	119	89
24	E8	72	B8	120	88
25	E7	73	B7	121	87
26	E6	74	B6	122	86
27	E5	75	B5	123	85
28	E4	76	B4	124	84
29	E3	77	B3	125	83
30	E2	78	B2	126	82
31	E1	79	B1	127	81
32	E0	80	B0	128	80
33	DF	81	AF		
34	DE	82	AE		
35	DD	83	AD		
36	DC	84	AC		
37	DB	85	AB		
38	DA	86	AA		
39	D9	87	A9		
40	D8	88	A8		
41	D7	89	A7		
42	D6	90	A6		
43	D5	91	A5		
44	D4	92	A4		
45	D3	93	A3		
46	D2	94	A2		
47	D1	95	A1		
48	D0	96	A0		

Abb. 6-2 Rückwärtssprünge

4.

```
      .                                  Befehlszähler beginnt bei 1012
  ┌─► 0FB7                               Gewünschter Sprung nach FB7
  │   .                                  (-91 Dezimalschritte)
  │   .                                  Sieh in der Tabelle nach:
  │   1010   D0    BNE A5                Rückwärts-    Sprung-
  │   1011   A5                          schritte      operand
  └─► 1012   .                           (dezimal)     (hex)
      .      .                           91            A5
```

Wenn das Prüfergebnis nicht gleich Null ist,
springe rückwärts nach FB7 (1012 - 5B Hexadezimalschritte).

Hinweis: Die obigen Rechnungen wurden alle mit Hex-Zahlen ausgeführt. Die hexadezimale Subtraktion werden wir in Kapitel 9 behandeln. Die Tabelle in Abb. 6-2 liefert Ihnen jeweils den nötigen Operanden für Rückwärtssprünge.

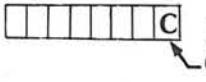
Alle in der folgenden Tabelle aufgeführten Sprungbefehle verwenden den relativen Adressierungs-Modus. Die Status Flag-Bits, die die Bedingung für die Sprungausführung bzw. -unterlassung liefern, sind in der folgenden Tabelle aufgeführt.

Buchstaben code	*Befehl*	*Status Flag*	*Sprung-bedingungen*
BCC	Springe, wenn Carry gelöscht	C	0
BCS	Springe, wenn Carry gesetzt	C	1
BEQ	Springe, wenn Ergebnis Null	Z	1
BMI	Springe, wenn Ergebnis negativ	N	1
BNE	Springe, wenn Ergebnis ungleich Null	Z	0
BPL	Springe, wenn Ergebnis positiv	N	0
BVC	Springe, wenn Überlauf gelöscht	V	0
BVS	Springe, wenn Überlauf gesetzt	V	1

Abb. 6-3 Status Flag Bits für Sprünge (Branches)

Um einige Sprunganweisungen vorzuführen, schreiben wir ein paar kleine Demonstrationsprogramme, die arithmetische Operationen bewirken.

ANWENDUNG DES CARRY FLAG-BIT



Nehmen Sie an, Sie hätten das Maschinenprogramm von Seite 39 mit den eingegebenen Werten 123 und 133 laufen lassen. Das Ergebnis der Summe würde als Null ausgegeben, da es zu groß ist, um in ein Byte zu passen. Es ist ein Extra-Bit nötig, um das richtige Ergebnis auszudrücken.

Summe dieses Problems:

Dezimal		Binär
123	=	01111011
133	=	10000101
Summe =		100000000

extra — 8 Bits ausgegeben

Wenn, wie in diesem Beispiel, bei der Addition von zwei 8-Bit Zahlen ein Extra-Bit vorkommt, dann setzt der Rechner das Carry Flag-Bit automatisch auf 1 (C = 1). Sie können sich davon überzeugen, indem Sie mit dem Assembler Modul das folgende Programm eingeben.

```
EDIT
10   *=$1000
20   CLC          ← Lösche das Carry Flag
30   LDA #$7B     ← Lade hex 7B (dezimal 123)
40   ADC #$85     ← Addiere hex 85
50 END              (dezimal 133)
■
```

Assemblieren Sie nun das Programm, indem Sie ASM tippen.

```
EDIT
10   *=$1000
20   CLC
30   LDA #$7B
40   ADC #$85
50 END
ASM
0000                      10    *= $1000
1000      18              20    CLC
1001      A97B            30    LDA #$7B
1003      6985            40    ADC #$85
                          50 END
EDIT
■
```

Beachten Sie die mit den Operanden in den Zeilen 30 und 40 des Quellprogramms (Programm in Assembler) verwendeten Zeichen (# und $). Das # Zeichen sagt dem

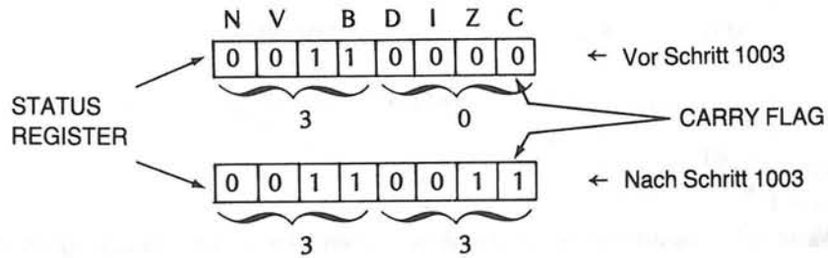
Rechner, daß die Anweisungen LDA und ADC im direkten Modus verwendet werden. Das Zeichen $ sagt dem Rechner, daß er die Zahlen im Operanden als hexadezimale Werte zu behandeln hat. Die Zeichen sind notwendiger Bestandteil der Assembler-Anweisung.

Geben Sie das Korrekturprogramm ein (Debugger) und verfolgen Sie den Programmablauf (trace). Sie können so die Änderungen im Status-Register (P), in dem ja die Flag-Bits gespeichert sind, beobachten. Richten Sie Ihr Augenmerk auf das Carry Flag-Bit im Status-Register vor und nach der Additionsanweisung.

```
.
.
.
.
EDIT
BUG                    ← Geben Sie BUG, um in das Korrekturprogramm einzutreten

DEBUG
T1000■                 ← Geben Sie T1000, um den Ablauf des bei 1000 beginnenden Programms zu verfolgen.
```

```
1000   18              CLC
  A=00 X=00 Y=00 P=B0 S=00
1001   A9  7B          LDA #$7B
  A=7B X=00 Y=00 P=30 S=00
1003   69  85          ADC #$85
  A=00 X=00 Y=00 P=33 S=00
1005   00              BRK
  A=00 X=00 Y=00 P=33 S=00
DEBUG
■
```

Sie sehen, daß der Akkumulator bei Schritt 1003 seinen Wert ändert, und zwar auf Null; das Status-Register ändert seinen Wert von 30 auf 33. Der Akkumulator zeigt uns als Summe Null an, beachten Sie jedoch, was das Register P (Status-Register) zeigt.



Wir können das Ergebnis als eine Kombination des Carry-Bits und des im Akkumulator befindlichen Wertes auffassen.

In Hexadezimal

1 + 0 0 = 100

Carry Bit — Im Akkumulator

Dezimal wäre dies 256 + 0 = 256

Durch Erweiterung unseres Programmes können wir die Ergebnisse unter Verwendung der BCC-Anweisung (Branch on Carry Clear, Springe, wenn kein Carry Bit) in den Speicher bringen. Löschen Sie den EDIT-Puffer mit dem Befehl NEW.

Mit dem Writer/Editor Programm des Assembler Moduls schreiben wir:

```
EDIT
10   *=$1000
20   CLC
30   LDA #$00          Lade und speichere 0 in 1050
40   STA $1050
50   LDA #$7D          (7x16)+13 = 125  dezimal
60   ADC #$A4          (10x16)+4 = 164 dezimal
70   STA $1051           Summe = 289 dezimal
80   BCC END
90   INC $1050
100  END                     = 256 + 32 + 1
■                            = 1   2   1 hex
                    gespeichert in 1050   gespeichert in 1051
```

Das Programm addiert zwei Hex-Zahlen und speichert die niedrigen 8 Bits aus dem Akkumulator in der Zelle 1051. Im Falle eines Übertrags (Carry) wird der Inhalt der Speicherzelle 1050 in Zeile 90 um 1 erhöht. Findet kein Übertrag statt, sorgt der BCC-Befehl in Zeile 80 dafür, daß der Rechner die INC-Anweisung in Zeile 90 ausläßt und zum Ende des Programms geht. Die Zelle 1050 enthält also bei Übertrag das Extra-Bit und ohne Übertrag den Wert Null. Die Kombination der Zellen 1050 und 1051 liefert das vollständige Ergebnis. Assemblieren Sie das Programm und lassen Sie es laufen. Sehen Sie sich dann die Ergebnisse an:

Assembliere
Tippen Sie: ASM

```
0000              10    *= $1000

1000  18          20    CLC

1001  A900        30    LDA #$00
```

```
1003  8D5010     40          STA $1050
1006  A97D       50          LDA #$7D
1008  69A4       60          ADC #$A4
100A  8D5110     70          STA $1051
100D  9003       80          BCC END
100F  EE5010     90          INC $1050
                 0100 END

EDIT
■
```

Wir sehen, daß die Operanden der Anweisungen STA bzw. INC zwei Bytes für die beteiligten Adressen belegen. Wieder zeigt das Zeichen $ einen Hex-Wert an. Der BCC-Operand zeigt einen Vorwärtssprung um 3 Schritte von 100F nach 1012, wo das Programm anhält.

Sie haben hier zum ersten Mal die INC-Anweisung verwendet. Mit ihr wird der Wert in einer spezifizierten Speicherzelle um 1 erhöht. Sie gleicht der früher von Ihnen benutzten INX-Anweisung (erhöhe Inhalt Register X).

Beispiel:
INC (Erhöhe den Speicher-Inhalt)
Op Code EE (im absoluten Adressierungs-Modus)
Zweites Byte 50 (niederwertiges Adress-Byte)
Drittes Byte 10 (höherwertiges Adress-Byte)
Betroffene Status-Flags: Z und C

DEBUG und laufen lassen

Tippen Sie: BUG

```
.
.
BUG

DEBUG
■
```

Tippen Sie: T1000 (Um das Programm zu verfolgen)


```
DEBUG
T1000
1000   18              CLC
  A=D7 X=00 Y=00 P=B0 S=00
1001     00            LDA  #$00
  A=00 X=00 Y=00 P=32 S=00
1003   8D 50 10        STA  $1050
  A=00 X=00 Y=00 P=32 S=00
1006   A9 7D           LDA  #$7D
  A=7D X=00 Y=00 P=30 S=00
1008   69 A4           ADC  #$A4
  A=21 X=00 Y=00 P=31 S=00
100A   8D 51 10        STA  $1051
  A=21 X=00 Y=00 P=31 S=00
100D   90 03           BCC  $1012
  A=21 X=00 Y=00 P=31 S=00
100F   EE 50 10        INC  $1050
  A=21 X=00 Y=00 P=31 S=00
1012   00              BRK
  A=21 X=00 Y=00 P=31 S=00
DEBUG
■
```

Carry ist gesetzt
Kein Sprung

P 00110001 C Flag
3 1

Beachten Sie, daß bei 100D *kein* Sprung gemacht wurde (das Carry-Bit war nicht gelöscht, es war da). Deshalb wurde in Zeile 100 F der Inhalt von 1050 um 1 erhöht.

Ergebniskontrolle

Um die Ergebnisse zu überprüfen, möchten Sie den Inhalt von 1050 und 1051 auf dem Schirm haben, um sich von der korrekten Antwort zu überzeugen.

Tippen Sie: D1050,1051

D für Display
(Bringe den Inhalt von 1050 bis 1051 auf den Schirm)

```
.
.
DEBUG
D1050,1051

1050      01 21
DEBUG
■
```

das hexadezimale Ergebnis $= (0X16^3)+(1X16^2)+(2X16)+1$
$= 0 + 256 + 32 + 1$
$= 289$ dezimal

Anstelle der BCC-Anweisung hätten wir im vorausgegangenen Programm auch die BCS-Anweisung (Branch on Carry Set, Springe, wenn Carry-Bit da) verwenden können.

Verwendung von BCC

```
010     *=$1000
020     CLC
030     LDA #$0
040     STA $1050
050     LDA #$7D
060     ADC #$A4
070     STA $1051
080     BCC END
090     INC $1050
100   END
```

Änderung

Verwendung von BCS

```
010     *=$1000
020     CLC
030     LDA #$0
040     STA $1050
050     LDA #$7D
060     ADC #$A4
070     STA $1051
080     BCS SET
090     JMP END
100   SET INC $1050
110   END
```

Beide Programme liefern das gleiche Ergebnis. Das Programm mit dem BCS-Befehl *springt* (JMP) zu END in Zeile 90, *wenn kein* Carry-Bit da ist. Es *springt* von Zeile 80 zu Zeile 100, *wenn ein* Carry-Bit da ist.

Die JMP-Anweisung in Zeile 90 im zweiten Programm ist der Sprunganweisung (BNE) eng verwandt. Der Unterschied besteht lediglich darin, daß ein JMP-Sprung an keinerlei Bedingung gebunden ist, d.h. immer ausgeführt wird. Man nennt ihn daher auch *„unbedingte (unconditional) Anweisung"*. Er wird im absoluten Adressierungs-Modus verwendet.

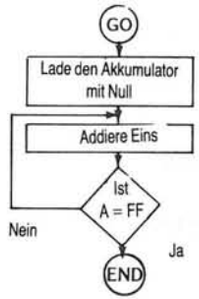
Alle BRANCH-Anweisungen hängen von einer durch die Flag-Bits des Status-Registers gegebenen Bedingung ab.

Alle JUMP-Anweisungen sind unbedingt

VERWENDUNG DES ZERO FLAG BITS

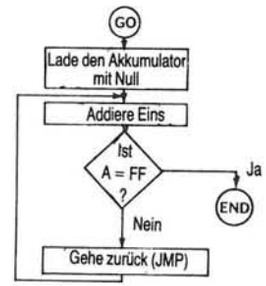
Im 4. Kapitel benutzten Sie ein Programm, das von 1 bis FF zählte. Die Zählschleife wurde durch Vergleich des Wertes im Akkumulator mit FF und einem BNE-Sprung (Springe, wenn das Ergebnis nicht Null ist) erzeugt. Das Assembler-Programm sah folgendermaßen aus:

```
10      *=$1000
20      CLC
30      LDA #0
40    LOOP ADC #1
50      CMP #$FF
60      BNE LOOP
70    END
```



Das Pendant zur BNE-Anweisung ist der BEQ-Befehl (Branch on result EQual to zero, Springe, wenn das Ergebnis gleich Null). Um diesen Befehl zu benutzen, können wir das Zählprogramm folgendermaßen schreiben:

```
10      *$1000
20      CLC
30      LDA #0
40    LOOP ADC #1
50      CMP #$FF
60      BEQ END
70      JMP LOOP
80    END
```



Geben Sie beide Programme mit dem Assembler ein und *verfolgen* Sie den Ablauf. Das Flußdiagramm zeigt Ihnen, daß das erste Programm (mit BNE) geradliniger ist. Sie haben aber die Möglichkeit, beide Anweisungen zu benutzen (BNE oder BEQ). In manchen Programmen ist BEQ vorzuziehen.

Vergleichen Sie die Ausführungszeiten beider Programme miteinander, so bemerken Sie, daß das erste etwas schneller ist. Bei der Beobachtung des Ablaufs auf dem Schirm können Sie sehen, daß beim Durchlaufen der Schleife des zweiten Programms jedesmal ein Befehl mehr ausgeführt werden muß.

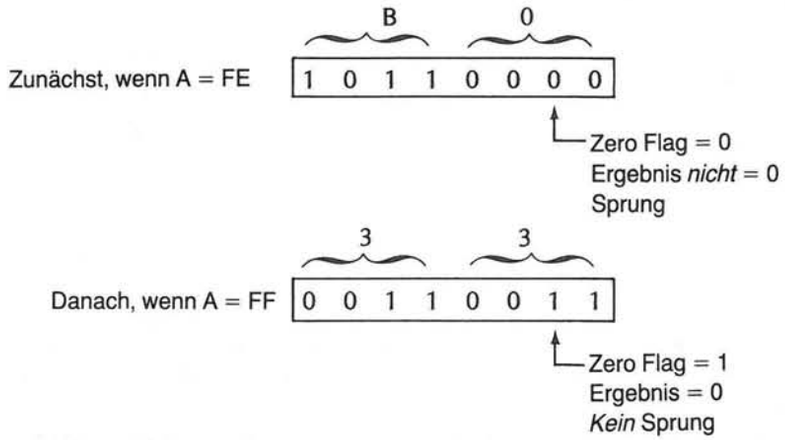
Ablauf des BNE-Programms:

Ausgabe auf dem Schirm am Ende des Laufs:

Drei Befehle in dieser Schleife

```
.
.
.
1007    D0 FA          BNE   $1003
  A=FD X=00 Y=00 P=B0 S=00
1003    69 01          ADC   #$01
  A=FE X=00 Y=00 P=B0 S=00          A=FE   P=B0
1005    C9 FF          CMP   #$FF
  A=FE X=00 Y=00 P=B0 S=00          Sprung zurück
1007    D0 FA          BNE   $1003  nach 1003
  A=FE X=00 Y=00 P=B0 S=00
1003    69 01          ADC   #$01
  A=FF X=00 Y=00 P=B0 S=00
1005    C9 FF          CMP   #$FF   A=FF   P=33
  A=FF X=00 Y=00 P=33 S=00
1007    D0 FA          BNE   $1003  Kein Sprung,
  A=FF X=00 Y=00 P=33 S=00          Programm endet
1009    00             BRK
  A=FF X=00 Y=00 P=33 S=00
DEBUG
■
```

Dieser Lauf dauerte etwa 2.3 Minuten.
Achten Sie auf das Status-Register nach der Vergleichsanweisung:



Ablauf des BEQ-Programms:

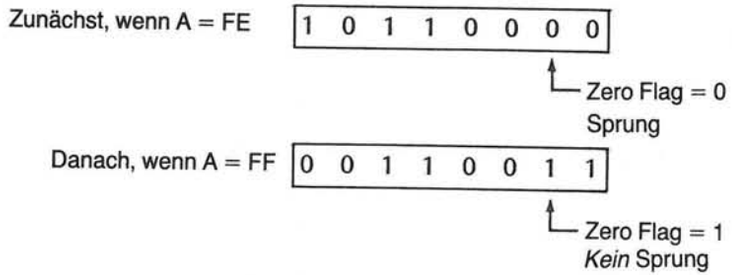
Ausgabe auf dem Schirm am Ende des Laufs:

4
Befehle in dieser
Schleife

```
.
.
.
1009    4C 03 10        JMP   $1003
  A=FD X=00 Y=00 P=B0 S=00
1003    69 01           ADC   #$01
  A=FE X=00 Y=00 P=B0 S=00
1005    C9 FF           CMP   #$FF
  A=FE X=00 Y=00 P=B0 S=00      ← Erstes A = FE
1007    F0 05           BEQ END
  A=FE X=00 Y=00 P=B0 S=00        Kein Sprung
1009    4C 03 10        JMP   $1003
  A=FE X=00 Y=00 P=B0 S=00
1003    69 01           ADC   #$01
  A=FF X=00 Y=00 P=B0 S=00
1005    C9 FF           CMP   #$FF
  A=FF X=00 Y=00 P=33 S=00      ← Zweites A = FF
1007    F0 05           BEQ END
  A=FF X=00 Y=00 P=33 S=00        Sprung zum Ende
100C    00              BRK
  A=FF X=00 Y=00 P=33 S=00
DEBUG
■
```

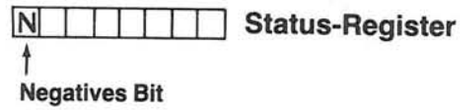

Dieser Lauf dauerte etwa 3.1 Minuten.

Und wieder: Achten Sie auf das Status-Register nach der Vergleichsanweisung:



Das BEQ-Programm benötigt beim Beobachtungslauf mehr Zeit als das BNE-Programm. Dieser Zeitunterschied würde beim normalen Programmablauf überhaupt nicht bemerkt werden. Bei Verfolgung des Programmablaufs auf dem Schirm ist der Zeitverlust für die Ausgabe des zusätzlichen Schrittes bemerkbar.

ANWENDUNG DES NEGATIVE FLAG BIT



Wir wollen wieder eine Variation des Zählprogramms verwenden, um zu sehen, wie einige Sprungbefehle entscheiden, ob eine Zahl positiv oder negativ ist. Wir nehmen uns als erstes den Befehl BPL vor (Branch on result Plus, Springe, wenn das Ergebnis positiv ist). Das N (negative) Flag-Bit wird bei Ausführung bestimmter Befehle für positive Zahlen auf Null und für negative Zahlen auf 1 gesetzt. Ist das N-Bit auf Null gesetzt, wird durch den BPL-Befehl ein Sprung ausgelöst (da das Ergebnis positiv ist). Ist das N-Bit auf 1 gesetzt (Ergebnis negativ), findet *kein* Sprung statt.

EIN BPL-DEMONSTRATIONSPROGRAMM

Mit dem Writer/Editor Programm des Assembler Moduls geben Sie das folgende Programm ein. Zunächst NEW, um das letzte Programm zu löschen, und dann:

```
10      *=$1000
20      CLC
30      LDA #0
40    LOOP ADC #1
50      BPL LOOP
60      END
```

Assemblieren Sie das Demonstrationsprogramm. Gehen Sie dann zum DEBUGGER und beobachten Sie den Ablauf des Programms.

```
.
.
.
ASM
0000                10          *= $1000

1000 18             20          CLC

1001 A900           30          LDA #$00

1003 6901           40 LOOP     ADC #$01

1005 10FA           50          BPL LOOP

                    60 END

EDIT
BUG                             ← Debugger Eingabe

DEBUG
T1000                           ← Programm-Verfolgung
```

Das Ende der Bildschirmausgabe sieht so aus:

```
.
.
.
1003   69 01          ADC  #01
  A=7F X=00 Y=00 P=30 S=00
1005   10 FA          BPL  $1003  ← Springe zurück nach 1003
  A=7F X=00 Y=00 P=30 S=00
1003   69 01          ADC  #$01
  A=80 X=00 Y=00 P=F0 S=00       ← Beachte die Änderung in P,
1005   10 FA          BPL  $1003     wenn A = 80
  A=80 X=00 Y=00 P=F0 S=00
1007   00             BRK         ← Kein Sprung
  A=80 X=00 Y=00 P=F0 S=00
DEBUG
■
```

Sie sehen, daß die Schleife so lange durchlaufen wurde, bis der Wert im Akkumulator 80 erreichte. Die erste negative Zahl, die erreicht wurde, lautete:

1	1	1	1	0	0	0	0

Status-Register

↑
N auf 1 gesetzt

Man kann negative Zahlen eindrucksvoller beobachten, wenn man im Akkumulator mit 0 anfängt und mit einer Schleife in jedem Durchlauf 1 subtrahiert.

Operation	*Ergebnis*	*Dezimales Äquivalent mit Vorzeichen*
0 - 1	FF	-1
FF - 1	FE	-2
FE - 1	FD	-3
FD - 1	FC	-4
.	.	.
.	.	.
etc	.	.
.	.	.
.	.	.
83 - 1	82	-126
82 - 1	81	-127
81 - 1	80	-128

Man kann sich 8-Bit Zahlen mit Vorzeichen vielleicht besser auf einem großen Zahlenrad aufgetragen denken, als auf der üblichen Zahlengeraden. Das würde so aussehen:

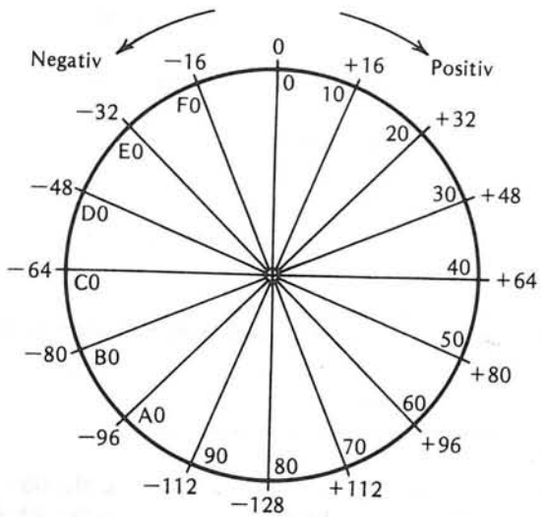


Abb. 6-4. Rad mit Vorzeichen-behafteten Zahlen. Hex-Werte innen; Dezimale Äquivalente außen

Um zu demonstrieren, wie man 1 vom Akkumulator subtrahiert, benutzen wir zum Erzeugen einer entsprechenden Schleife den BMI-Befehl (Branch on result Minus - Springe bei negativem Ergebnis). Bei der Subtraktion wird das Carry-Bit für den negativen Übertrag verwendet. Bei der Addition haben wir vor deren Ausführung das Carry-Bit *gelöscht*. Bei der Subtraktion müssen wir es vor deren Ausführung auf 1 *setzen*. Hier ist das im Writer/Editor Modus des Assemblers eingegebene Programm:

```
10    *=$1000
20    LDA #0
30    LOOP SEC      <- Setze das Carry Bit zur Subtraktion
40    SBC #1
50    BMI LOOP      <- Schleife bei Minus
60    END
```

Assemblieren Sie das Programm. Gehen Sie in den DEBUGGER Modus und führen Sie den ersten Teil des Programms in Einzelschritten aus.

```
ASM
0000              10              *= $1000

1000  A900         20              LDA #$00

1002  38           30 LOOP         SEC

1003  E901         40              SBC #$01

1005  30FB         50              BMI LOOP

                   60 END

EDIT
BUG                          <- Debugger Eingabe

DEBUG
S1000                        <-  Beginn der Einzelschritte bei 1000
```

```
DEBUG
S1000                                   <- Erster Schritt
1000   A9 00          LDA    #$00          Laden des Akkumulators mit 0
  A=00 X=00 Y=00 P=33 S=00
```



```
DEBUG                                     Tippen Sie S und drücken
S                                       <- Sie RETURN
1002   38             SEC                 Setzen des Carry Flags
  A=00 X=00 Y=00 P=33 S=00
DEBUG
S                                       <- Dritter Schritt
1003   E9 01          SBC    #$01         Subtraktion von Eins
  A=FF X=00 Y=00 P=B0 S=00                A = FF; Negatives Flag da
DEBUG
S                                       <- Vierter Schritt
1005   30 FB          BMI    $1002        Springe, wenn negativ
  A=FF X=00 Y=00 P=B0 S=00
DEBUG
S                                       <- Fünfter Schritt
1002   38             SEC                 Setzen des Carrv Flag
  A=FF X=00 Y=00 P=B1 S=00
DEBUG
S                                       <- Sechster Schritt
1003   E9 01          SBC    #$01         Nochmals Subtraktion von Eins
  A=FE X=00 Y=00 P=B1 S=00
DEBUG
S                                       <- Siebter Schritt
1005   30 FB          BMI    $1002        Sprung, wenn negativ
  A=FE X=00 Y=00 P=B1 S=00
DEBUG
S                                       <- Achter Schritt
1002   38             SEC                 Setzen des Carry Flag
  A=FE X=00 Y=00 P=B1 S=00
DEBUG
■
```

STOP hier

Nun, Sie sehen, daß das noch einige Zeit so weitergeht, bis wir schließlich eine nicht negative Zahl erreichen. Ebenso sehen Sie, daß der Rechner FF und FE als negative Werte ansieht und deshalb zurückspringt, um wieder zu subtrahieren. Er springt so lange zurück, bis der Akkumulator zu 7F kommt. Ein Blick auf unser Zahlenrad in Abb. 6-4. verrät Ihnen, daß 80 als negativer Wert angesehen wird (-128 dezimal). 7F dagegen wird als positiv angesehen (+ 127 dezimal). Daher unterbleibt der Rücksprung an dieser Stelle.

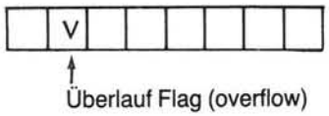
Mühen Sie sich nicht durch alle negativen Zahlen bis einschließlich -128, ändern Sie stattdessen den Trace Modus, um das Programm zu Ende zu bringen. Nach der letzten DEBUG- Aufforderung geben Sie T 1000 ein und drücken auf die RETURN-Taste.

Jetzt arbeitet der Rechner das Programm ab und Sie können den Vorgang auf dem Schirm sehen, insbesondere das END-Ergebnis.
So sieht es auf dem Schirm aus:

```
.
.
.
1005    30 FB          BMI  $1002
  A=80 X=00 Y=00 P=B1 S=00          ← 80 sei immer noch negativ -
1002    38             SEC              Sprung
  A=80 X=00 Y=00 P=B1 S=00
1003    E9 01          SBC  #$01
  A=7F X=00 Y=00 P=71 S=00          ← Negativ Flag gelöscht (P = 71)
1005    30 FB          BMI  $1002
  A=7F X=00 Y=00 P=71 S=00          ←  7F sei positiv - kein Sprung
1007    00             BRK
  A=7F X=00 Y=00 P=71 S=00
DEBUG
■
```

DAS OVERFLOW-FLAG BIT

Auch die beiden Sprunganweisungen BVC (Branch on oVerflow Clear - Springe, wenn Überlauf 0) und BVS (Branch on oVerflow Set - Springe bei Überlauf 1) können den normalerweise sequentiellen Ablauf eines Programms ändern. Wenn Sie keine Arithmetik mit Vorzeichen-behafteten Zahlen betreiben, können Sie das Overflow-Flag einfach ignorieren. Arithmetik mit Zahlen, die Vorzeichen haben, werden wir in Kap. 9 diskutieren.



Status-Register

ZUSAMMENFASSUNG

In diesem Kapitel wurden Sprunganweisungen behandelt - was sie sind und wie man sie benutzt. Sie haben gelernt, daß

- alle Sprunganweisungen im relativen Adressierungs-Modus gegeben werden;
- Sprunganweisungen einen Sprung im Programm verursachen, wenn eine spezifizierte Bedingung erfüllt ist;
- der Befehlszähler bei einem Sprung relativ zu seinem augenblicklichen Stand geändert wird;
- das zweite Byte einer Sprunganweisung angibt, wie weit und in welcher Richtung (vorwärts oder rückwärts) der Befehlszähler geändert wird;

- Vorwärtssprünge ausgeführt werden, wenn das zweite Byte der Sprunganweisung einen Wert zwischen 1 und 7F (incl. 1 und 7F) enthält;
- Rückwärtssprünge ausgeführt werden, wenn das zweite Byte einen Wert zwischen 80 und FF (beide Werte incl.) enthält;
- die folgenden Status-Flags zur Spezifikation der Sprungbedingung verwendet werden:
  - Negative
  - Overflow
  - Zero
  - Carry
- die Bedingung erfüllt ist, wenn das spezifizierte Flag auf 1 gesetzt ist.
- die Bedingung nicht erfüllt ist, wenn das spezifizierte Flag auf 0 gesetzt ist.
- Hexadezimale Zahlen als negative Zahlen angesehen werden können, wenn sie zwischen 80 und FF, und als positiv, wenn sie zwischen 1 und 7F liegen.
- alle Sprunganweisungen Null wie einen positiven Wert behandeln.

ÜBUNGEN

1. Welche Adresse enthält der Befehlszähler, wenn der folgende Befehl ausgeführt wird? _____

   _____
   (1010, 1011, or 1012)

Speicher	Op Code	Anweisung
1010	D0	BNE
1011	F9	

2. Welches Status-Flag entscheidet, ob in Aufgabe 1 ein Sprung durchgeführt wird oder nicht? _____ Flag
3. In welcher Richtung (vorwärts oder rückwärts) erfolgt der Sprung in Aufgabe 1?

   _____
4. Geben Sie, unter Verwendung der Tabellen in Abb. 6-1 und 6-2, die hexadezimalen Operanden an, die für die folgenden Sprünge benutzt werden sollen:
   (a) 38 Schritte vorwärts (dezimal) _____
   (b) 100 Schritte rückwärts (dezimal) _____
   (c) 93 Schritte vorwärts (dezimal) _____
   (d) 42 Schritte rückwärts (dezimal) _____
5. Nennen Sie die Zwei-Byte Sprunganweisung für 11 Schritte rückwärts (dezimal) bei positivem Ergebnis einer Operation.

Op Code	Buchst. Code
_____	_____
_____	

6. Welches Flag wird gesetzt, wenn zwei Hex-Werte addiert werden und das Ergebnis größer als FF ist?

   _____

   Negative, Null oder Carry

7. Welcher Hex-Wert befindet sich nach Ausführung folgender beider Anweisungen im Akkumulator? _____

```
LDA #$7A
ADC #$87
```

8. Welchen Inhalt haben die folgenden Stellen des Status-Registers (Flag 0 oder 1) nach Ausführung der beiden Anweisungen von Aufgabe 7?
   Zero flag _____
   Negative flag _____
   Carry flag _____

9. Welcher Assembler-Modus wird im folgenden verwendet (EDIT, ASM, DEBUG)
   (a) um ein Quell-Programm zu LISTen _____
   (b) um ein Objekt-Programm laufen zu lassen _____
   (c) um den Inhalt einer Speicherzelle, die zum Speichern des Ergebnisses eines Maschinen-Programms benutzt wurde, abzubilden _____
   (d) um ein Objekt-Programm zu verfolgen _____

10. Nennen Sie die Bedeutung der folgenden Buchstaben- Codes:
    (a) BCC _____
    (b) BEQ _____
    (c) BPL _____
    (d) BCS _____
    (e) BNE _____
    (f) BMI _____

ANTWORTEN

1. 1012 (stets die Adresse des Befehls, der als nächster ausgeführt werden soll)
2. Zero flag
3. Rückwärts
4. (a) 26
   (b) 9C
   (c) 5D
   (d) D6

5. Op Code: 10 F5 — Buchst.-Code: BPL
6. Ganz bestimmt Carry Flag (Möglicherweise auch N, Z).
7. 01
8. Zero flag 0
   Negative flag 0
   Carry flag 1
9. (a) EDIT
   (b) DEBUG
   (c) DEBUG
   (d) DEBUG
10. (a) Branch on Carry Clear (Sprünge bei Carry-Flag 0)
    (b) Branch on result Equal zero (Sprünge bei Ergebnis Null)
    (c) Branch on result Plus (Sprünge bei Ergebnis Plus)
    (d) Branch on Carry Set (Sprünge bei Carry-Flag 1)
    (e) Branch on result Not Equal zero (Sprünge bei Ergebnis nicht Null)
    (f) Branch on result Minus (Sprünge bei Ergebnis negativ)

Assembler - Überblick

Dieses Kapitel enthält eine Zusammenfassung dessen, was Sie über den Atari Assembler gelernt haben, und bringt einiges, was wir bisher noch nicht behandelt haben.
Ein Assembler-Programm besteht aus numerierten Anweisungen, Marken, Buchstaben-Befehlscodes, Operanden und Kommentaren. Das im Writer/Editor Modus des Assembler Moduls geschriebene Programm wird *Quell-Programm* genannt. Das Quell-Programm wird im Assembler Modus assembliert. Dadurch wird das Maschinen-Programm, auch Objekt-Programm genannt, erzeugt. Der Assembler bringt dieses Programm in den Speicher. Um die gewünschten Ergebnisse zu erhalten, wird das Objekt-Programm anschließend im Debug-Modus ausgeführt.

FORMAT DES QUELL-PROGRAMMS

Das Quell-Programm besteht aus Anweisungen, die mit einer Zeilennummer beginnen und durch Drücken der RETURN-Taste abgeschlossen werden. Die Anweisungen sind in folgende Felder unterteilt, von denen einige wahlweise zu verwenden sind.

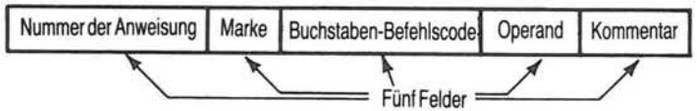


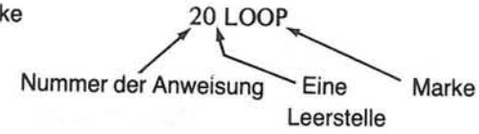
Abb. 7-1 Anweisungsfelder

Jede Anweisung muß mit einer Anweisungsnummer, die zwischen 0 und 65,535 liegt, beginnen. Die Numerierung der Anweisungen sollten Sie dabei so vornehmen, daß Sie später neue einfügen können, falls Sie Ihr ursprüngliches Programm ändern wollen. Vielfache von 10 (d.h. 10, 20, 30 usw.) eignen sich dafür sehr gut. Der Writer/Editor hat geeignete Befehle für die Numerierung von Programmen (siehe Seite 129). Eine Anweisung kann bis zu 107 Zeichen lang sein.

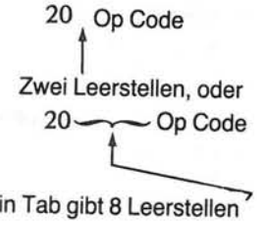
Das Markenfeld kann wahlweise verwendet werden. Bei Benutzung des Feldes muß beachtet werden, daß zwischen der letzten Ziffer der Zeilennummer und dem ersten Buchstaben der Marke *genau eine* Leerstelle liegt. Eine Marke muß stets mit einem Buchstaben beginnen und darf nur aus Buchstaben und Zahlen bestehen. Sie kann so lang wie eine Anweisung sein, muß aber aus mindestens zwei Zeichen bestehen. Wird *keine Marke benutzt,* müssen zwei Leerstellen (oder tab) zwischen der Anweisungsnummer und dem Feld für den Buchstaben-Befehlscode (das nach dem Markenfeld kommt) liegen.

Beispiele:

Mit Marke



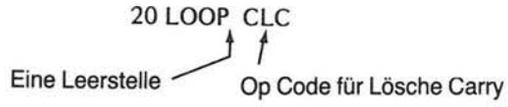
Ohne Marke



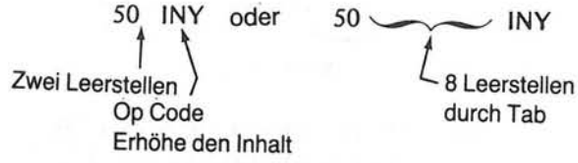
In Anhang A und B sind die erlaubten Befehlscodes aufgeführt. Das Befehlsfeld ist durch *mindestens* zwei Leerstellen getrennt, wenn keine Marke verwendet wird, andernfalls durch genau *eine Leerstelle.* Wenn Sie einen Buchstabencode in ein falsches Feld eintragen, so wird dieser Fehler vom Rechner im Writer/Editor Modus nicht bemerkt. Er wird aber beim Assemblieren des Programms als Error - 6 (Fehler) gemeldet (Siehe Anhang D, Error codes - d.h. Fehlermeldungen).

Beispiele:

Mit Marke



Ohne Marke



Das Feld für den *Operanden* ist durch mindestens eine Leerstelle (oder tab) vom Befehlsfeld getrennt. Einige Befehle *verlangen* einen Operanden, andere nicht. Wird ein Befehl mit einem Operanden benutzt, muß der Operand eine diesem Befehl entsprechende Form haben, die in Anhang A angegeben ist.

Beispiele:

Für Hex-Operanden 30 LDA #20

Zwei Leerstellen — Op Code — Hex-Operand

Direkter Adressierungs-Modus

30 LDA $1100

Lade Akkumulator aus Speicherzelle 1100

Für Dezimal-Operanden 30 LDA #32

Direkte Adressierung — Operand (dezimal 32)

10 LDA 4352

Absolute Adressierung (kein # Zeichen) — Operand (dezimal 4352)

Das letzte Feld (Kommentar) erscheint zwar in der Auflistung des Programms, wird aber nicht assembliert, weshalb es im Objekt-Programm nicht mehr auftritt. Es gibt zwei Kommentar-Möglichkeiten.

- Man kann den Kommentar in das Feld schreiben, das für die Anweisung nach dem Operanden (falls einer vorhanden ist) oder nach dem Befehlscode (falls kein Operand vorhanden ist) übrig bleibt. Reicht der vorhandene Platz zur Unterbringung des gesamten Kommentars nicht aus, so kann man diesen in der nächsten Zeile fortsetzen.

Beispiele:

20 CLC BEREITE ADDITION VOR

Kommentar

30 BNE LOOP WENN DER ZAEHLER
NICHT NULL ZURUECK ZU LOOP

Kommentar auf nächste Zeile ausdehnen (keine Zeilennummer)

- Die zweite Möglichkeit besteht darin, dem Kommentar eine eigene Zeile zuzuordnen. In diesem Fall folgt auf die Anweisungsnummer eine Leerstelle und ein Semikolon. Danach beginnt der Kommentar.

Beispiele:

```
30   CLC
40   ;BEREITE ADDITION VOR
```

und

```
30   BNE LOOP WENN DER ZAEHLER
40   ;NICHT NULL ZURUECK ZU LOOP
```

VERWENDUNG VON OPERANDEN

Zeilennummern, Befehlscodes und Kommentare sind unkompliziert und lassen sich leicht handhaben. Operanden dagegen sind zuweilen recht verzwickt und bedürfen daher weiterer Erläuterungen.

Operanden können als hexadezimale Zahlen, dezimale Zahlen oder als Buchstabengruppe auftreten. Es ist dem Programmierer überlassen, die jeweils gewünschte Form zu wählen.

Eine als Operand verwendete Zahl wird vom Assembler als dezimale Zahl angesehen, es sei denn, es ist ihr das Zeichen $ vorangestellt. Steht $ vor der Zahl, wird sie als hexadezimale Zahl behandelt.

Beispiele:

```
40   LDA 4500     ← 4500 wird als Dezimalzahl angesehen
50   STA $1100    ← $1100 wird als Hexadezimalzahl angesehen
```

Wenn einer Buchstabengruppe ein numerischer Wert zugewiesen oder sie als Marke benutzt wurde, kann sie als Operand verwendet werden.

Beispiele:

```
20   ABC=$33
30   DEF=51
40   LOOP CLC
.
.
.
.
80   BNE LOOP    ← LOOP ist vorher als Marke benutzt worden
90   CMP DEF     ← Dezimalwert 51
     oder
.
.
90   CMP $ABC    ← Hexadezimalwert 33
```

Der Operand liefert nicht nur Daten, er teilt dem Computer auch den Adressierungsmodus mit, der für die durch den Befehlscode spezifizierte Operation zu verwenden ist.

Beispiele:

20	LDA # 12	Das # Zeichen zeigt den direkten Adressierungs-Modus an
30	CPY $1212	Absoluter Adressierungs-Modus (kein # Zeichen)
40	STA $1250,Y	Absoluter indizierter Modus (beachte: , Y hinzugefügt)
50	ADC ($2B,X)	Indizierter indirekter Modus (beachte: Klammern)
60	CMP ($3C),Y	Indirekter indizierter Modus (beachte: Stellung der Klammern)
70	INC $3C	Null-Seiten Modus
80	INC $20,X	Indizierter Null-Seiten Modus

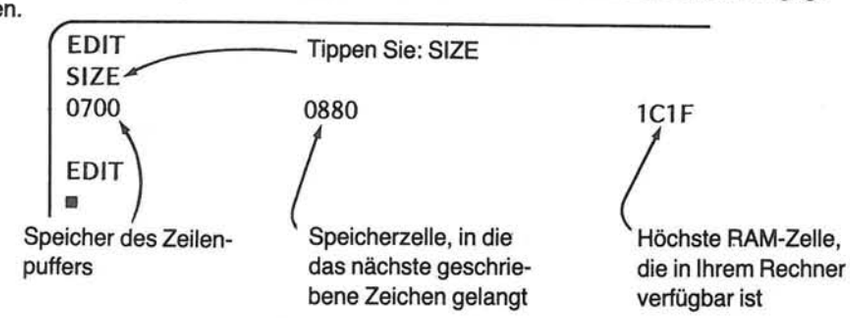
DER WRITER/EDITOR MODUS DES ASSEMBLERS

Der Writer/Editor Modus (auch einfach Edit Modus genannt) steuert über die Tastatur die Kommunikation zwischen Ihnen und dem Schirm. Wenn Sie Ihren Atari mit dem Assembler Modul einschalten, erscheint auf dem Schirm die Aufforderung Edit. Kommt diese Mitteilung nicht, so liegt irgend ein Irrtum oder Fehler vor. Schauen Sie im Operator's Manual nach, um sicher zu gehen, daß Sie alles richtig gemacht haben.

Alle Eingaben über die Tastatur erscheinen nun an aufeinanderfolgenden Stellen auf dem Schirm, wobei die Position des Cursors ( ) den Anfang markiert. Der Schirm hat pro Zeile 38 Plätze zur Verfügung. Ist das Ende einer Zeile erreicht, sorgt ein automatischer Zeilenvorschub dafür, daß Ihre Eintragungen in der nächsten Zeile weitergehen.

Einige der nachfolgenden Befehle, die im Edit Modus verwendet werden, haben wir bereits diskutiert; es kommen ein paar neue hinzu.

1. SIZE wird verwendet, um den Zeilenpuffer und Edit Text Puffer im Speicher zu finden. Bei Ausführung des SIZE-Befehls werden auf dem Schirm drei Zahlen ausgegeben.



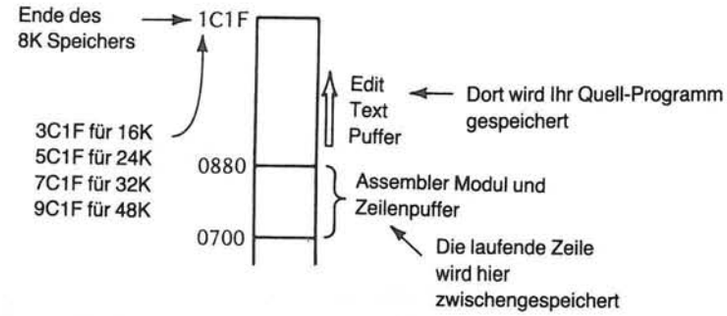
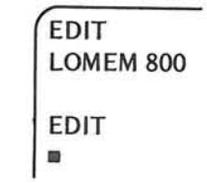


Abb. 7-2 Pufferspeicher

Die mittlere Zahl in unserem Beispiel (0880) wächst, während der Edit Text Puffer Ihr Programm aufnimmt. Man kann daher mit SIZE feststellen, wieviel Speicherplatz bei der Eingabe Ihres *Quell-Programms* gebraucht wird. Das *Objekt-Programm* (in Maschinensprache) darf nicht mit dem Quell-Programm im Speicher kollidieren. Es muß daher unter höheren Adressen als das Quell-Programm abgelegt werden.

2. LOMEM wird zur Verschiebung der Puffer im Speicher benutzt. Sie haben diesen Befehl noch nicht verwendet und werden ihn möglicherweise auch nie benötigen. Wenn Sie in Ihrem assemblierten Programm 256 Speicherplätze zwischen 0700 bis 0800 verwenden wollen, wenden Sie den LOMEM-Befehl folgendermaßen an:



Danach hätte Ihre Speicherorganisation das in Abb. 7-3 gezeigte Aussehen.

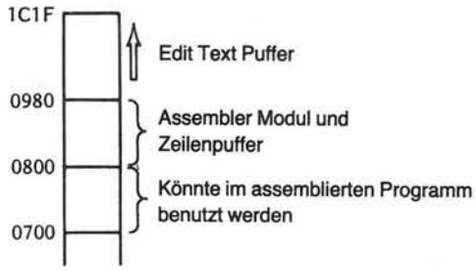


Abb. 7-3 Veränderter Pufferspeicher

Geben Sie nun wieder den SIZE-Befehl.

```
EDIT
LOMEM 800

EDIT
SIZE
0800            0980            1C1F

EDIT
■
```

Haben Sie erst einmal mit dem Schreiben eines Programms begonnen, können Sie die Lage der Puffer im Speicher nicht mehr ändern. Wollen Sie den LOMEM-Befehl anwenden, *dann muß dies der erste Befehl unmittelbar nach dem Einschalten sein.*

3. LIST dient zur Ausgabe des gerade im Edit Text Puffer befindlichen Programms auf dem Bildschirm. Der Befehl kann in verschiedenen Formen verwendet werden:

(a) LIST um das ganze Programm auszugeben

```
EDIT
LIST

10   *=$1000
20   CLC
30   LDA #0
40  LOOP ADC #1
50   CMP #$FF
60   BNE LOOP
70  END

EDIT
■
```


(b) LIST 40 um Zeile 40 auszugeben

```
EDIT
LIST 40

40  LOOP ADC #1

EDIT
■
```

(c) LIST 50, 70, um alle Zeilen von 50 bis 70 auszugeben.

```
EDIT
LIST 50,70

50     CMP #$FF
60     BNE LOOP
70    END

EDIT
■
```

4. NEW dient zum Löschen des Edit Text Puffers. Der Befehl löscht nicht ein zuvor assembliertes Maschinenprogramm, das sich im Speicher befindet, sondern nur den Edit Text Puffer. Nach Ausführung von NEW ist jedes vorher im Edit Text Puffer gespeicherte **Quell-Programm** verloren.
Befindet sich das Programm von 3a unter LIST im Edit Text Puffer und Sie geben NEW, wird der Edit Text Puffer gelöscht.

```
EDIT
NEW          <-- Sie haben NEW getippt

EDIT
LIST         <-- Dann LIST

EDIT         <-- Nichts ist zu sehen
■
```

5. DEL dient zum Löschen von Anweisungen im Edit Text Puffer. Der Befehl wurde noch nicht verwendet. Er erlaubt Ihnen, eine Zeile oder mehrere aufeinanderfolgende Zeilen zu löschen.

Wäre das Programm 3a noch im Edit Text Puffer und Sie wollten Zeile 20 im Programm löschen, sähe dies so aus:

```
EDIT
DEL 20

EDIT
LIST

10   *=$1000
30    LDA #0
40  LOOP ADC #1
50    CMP #$FF
60    BNE LOOP
70  END

EDIT
■
```

← Zeile 20 ist verschwunden

Löschen Sie nun die Zeilen 40, 50 und 60.

```
EDIT
DEL 40,60

EDIT
LIST

10   *=$1000
30    LDA #0
70  END

EDIT
■
```

← Nicht mehr viel übrig

6. REP dient zum Austausch einer spezifizierten Zeichenkette im Edit Text Puffer gegen eine andere spezifizierte Zeichenkette. Es läßt sich so eine Programmänderung mittels einer der verschiedenen Formen dieses Befehls rasch durchführen. Wir gehen wieder davon aus, daß sich das Programm 3a im Edit Text Puffer befindet.

(a) Um die Zeichenkette „LDA #0" an der Stelle ihres ersten Auftretens durch die Zeichenkette „LDA #5" zu ersetzen:


```
EDIT
REP/LDA #0/LDA #5/

EDIT
■
```

Alte Zeichenkette
Neue Zeichenkette
Schrägstriche

(b) Um die Zeichenkette „LOOP" überall, wo sie im Programm auftritt, durch „CIRCLE" zu ersetzen:

```
EDIT
REP/LOOP/CIRCLE/,A

EDIT
■
```

,A nach dem letzten Schrägstrich hinzugefügt

Die Zeilen 40 und 60 werden dabei folgendermaßen geändert:

```
40   CIRCLE ADC #1
50   BNE CIRCLE
```

Weitere Anwendungsmöglichkeiten dieses Befehls finden sich im Atari Assembler Handbuch.

7. NUM dient zur automatischen Numerierung von Anweisungen in einem Assembler-Programm. Auch NUM hat verschiedene Anwendungsmöglichkeiten.

(a) Erhöhung der Anweisungsnummer um 10:

```
EDIT
NUM

10 ■
```

← Gibt eine 10 und eine Leerstelle aus

Geben Sie die erste Zeile ein und drücken Sie RETURN

```
EDIT
NUM

10  *=$1000
20  ■
```

← Tippen Sie noch eine Leerstelle, # = $1000 und drücken Sie die RETURN-Taste

Gibt eine 20 und eine Leerstelle aus

(b) Erhöhung um einen von 10 verschiedenen Wert:

```
EDIT
NUM 3

3                    ← Fängt jetzt bei 3 an
```

Schreiben Sie die erste Zeile und drücken Sie die RETURN-Taste.

```
EDIT
NUM 3

                     Tippen Sie eine Leerstelle,
3   *=$1000    ←     dann # = $1000 und drücken Sie
6 ■                  RETURN
```

Gibt eine 6 und eine Leerstelle aus

(c) Erzwingung einer neuen Zeile u. Änderung des Zuwachses (od. der Anfangszeile).

```
EDIT
NUM                  ← Reguläre NUM-Anweisung

10  *=$1000
20  CLC
NUM 50,3             ← Neue NUM-Anweisung

50  LDA #0           ← Neue Zeilennummer, 50
53 LOOP ADC #1       ← Neuer Zuwachs, 3
56 ■
```

Drücken Sie RETURN, um NUM zu löschen.

```
.
.
50  LDA #0
53 LOOP ADC #1
56                   Geben Sie hier RETURN
■                    NUM ist nicht mehr vorhanden
```

8. REN dient zur neuerlichen Numerierung von Anweisungen im Edit Text Puffer.

(a) Tippen Sie REN, wenn die neuen Anweisungsnummern jeweils um 10 (beginnend mit 10) größer sein sollen. Danach drücken Sie die RETURN-Taste.

(b) Tippen Sie REN 5, wenn die neuen Anweisungsnummern jeweils um 5 (beginnend mit der Zeilennummer 10) größer sein sollen. Danach drücken Sie die RETURN-Taste.

(c) Tippen Sie REN 20,2, wenn alle neuen Anweisungsnummern um jeweils 2 (beginnend mit der Zeilennummer 20) größer sein sollen. Danach drücken Sie die RETURN-Taste.

9. ASM wird benutzt, wenn man vom **Writer/Editor Programm** ins **Assembler Programm** gehen will.

10. BUG wird benutzt, wenn man vom **Writer/Editor Programm** ins **Debug-Programm** gehen will.
Darüber hinaus gibt es Befehle zur Sicherung und zum Wiederauffinden von Programmen oder bestimmten Speicherblöcken. Ihre Verwendung wird im Atari Assembler Handbuch beschrieben, wir wollen hier nicht näher darauf eingehen. Jeder der folgenden Befehle
LIST, ENTER, ASM, SAVE und LOAD
kann in verschiedenen Formen verwendet werden, die davon abhängen, ob Sie den Bildschirm, den Drucker, die Kassette oder Platte benutzen.

DER DEBUG MODUS

Der Debug Modus, der vom Writer/Editor Modus aus mittels „BUG" eingegeben wird, ermöglicht Ihnen die Abänderung oder Ausführung des assemblierten Objekt-Programms. Außerdem können Sie auf diese Weise Datenlisten, die das Objekt-Programm benutzt, eingeben und ändern. Die Mitteilung auf dem Bildschirm, daß Sie sich in diesem Modus befinden, lautet DEBUG.

```
EDIT
BUG  ←               Tippen Sie BUG, um DEBUG einzugeben

DEBUG
■
```

Der Buchstabe X ist der Befehl, mit dem Sie aus dem Debug Modus in den Writer/Editor Modus zurückkehren können.

```
EDIT
BUG                  ← Tippen Sie BUG, um in den Debug Modus zu gehen

DEBUG
X                    ← Tippen Sie X, um Debug zu verlassen

EDIT                 ← Nun sind Sie wieder im Writer/Editor Modus
■
```

Geben Sie das folgende Programm ein, um die Befehle, die im Debug Modus verwendet werden, zu demonstrieren.

```
EDIT
NUM                    ← Zur automatischen Numerierung mit Zehn

10   *=$1000
20   LDY #0
30   LDA #0
40 LOOP CLC
50   INY
60   ADC #1
70   CPY #3
80   BNE LOOP
90 END
100 ■
```

Assemblieren Sie nun das Programm.

```
.
.
.
80   BNE LOOP
90 END

ASM■                   Tippen Sie: ASM und drücken Sie RETURN
```

```
0000                   10          *=      $1000

1000 A000              20          LDY     #0

1002 A900              30          LDA     #0

1004 18                40 LOOP     CLC

1005 C8                50          INY

1006 6901              60          ADC #1

1008 C003              70          CPY #3

100A D0F8              80          BNE LOOP

                       90 END

EDIT
■
```


Gehen Sie jetzt in den Debug Modus.

```
.
.
.
EDIT
BUG

DEBUG
■
```

Es gibt drei Möglichkeiten, ein Programm auszuführen. Die von Ihnen gewählte Methode hängt davon ab, ob Sie die einzelnen Schritte bei ihrer Ausführung auf dem Bildschirm sehen möchten oder nicht.

1. Tippen Sie zur Ausführung des Programms ohne Bildschirmausdruck G1000 und drücken Sie die RETURN-Taste.

```
DEBUG
G1000
100C          A=03 X=00 Y=03 P=33 S=00   ←
DEBUG                                    |
■                                        |
                                         |
   Die Speicherzelle, die unmittelbar auf Ihr
   Programm folgt, wird bei Programmende zu-
   sammen mit dem Inhalt der Register
   ausgegeben.
```

Das Register A (der Akkumulator) enthält eine 3. Das Register Y enthält ebenfalls eine 3. Wie erwartet.

2. Um ein Programm zu verfolgen (TRACE), tippen Sie T1000 und drücken Sie die RETURN-Taste. Die Ausgabe auf dem Bildschirm gibt jede Anweisung, die ausgeführt wurde, aus, und zeigt außerdem den nach der Ausführung vorhandenen Registerinhalt.

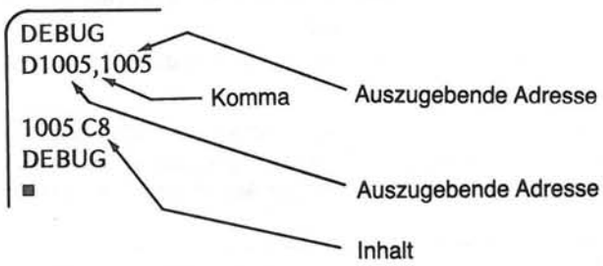
```
DEBUG
T1000
1000          A0 00                LDY     #$00
  A=00 X=00 Y=00 P=33 S=00
1002          A9 00                LDA     #$00
  A=00 X=00 Y=00 P=33 S=00
1004          18                   CLC
  A=00 X=00 Y=00 P=32 S=00
1005          C8                   INY
  A=00 X=00 Y=01 P=32 S=00
    :
  etc. bis zum Ende des Programms
```

3. Um ein Programm in einzelnen Schritten zu verfolgen, tippen Sie S 1000 und drükken Sie RETURN. Die Ausgabe zeigt die Ergebnisse der ersten Anweisung. Zur Fortsetzung der Prozedur tippen Sie S und drücken Sie RETURN für die jeweilige Anweisung, die Sie ausgeführt haben möchten.

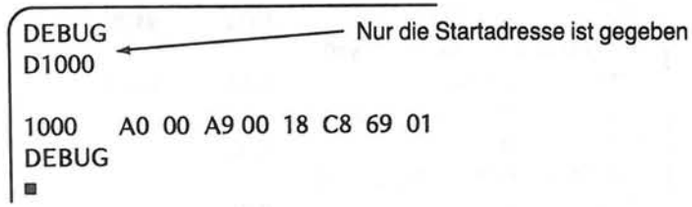
```
DEBUG
S1000
1000        A0 00              LDY    #$00
   A=03 X=00 Y=00 P=33 S=00
DEBUG
S
1002        A9 00              LDA    #$00
   A=00 X=00 Y=00 P=33 S=00
DEBUG
S
1004        18                 CLC
   A=00 X=00 Y=00 P=32 S=00
DEBUG
■
```

4. Um den Speicherinhalt auszugeben, tippen Sie D und die Adresse der Speicherzelle, die Sie untersuchen möchten.

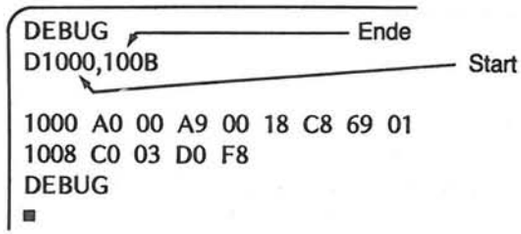
(a) Ausgabe einer einzelnen Speicherzelle.



(b) Ausgabe von bis zu 8 aufeinanderfolgenden Zellen.

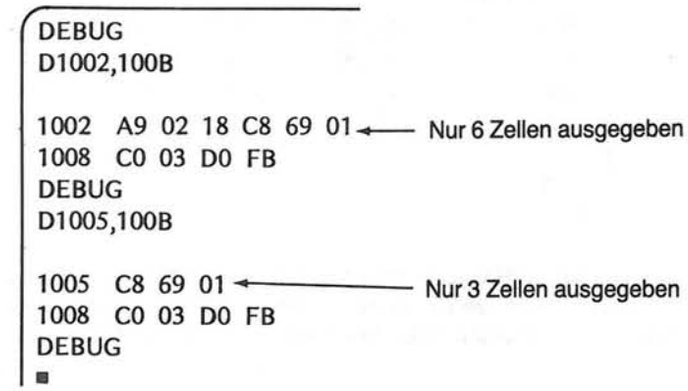


(c) Zur Ausgabe von mehr als 8 aufeinanderfolgenden Zellen tippen Sie D, die Startadresse, ein Komma und die Endadresse.



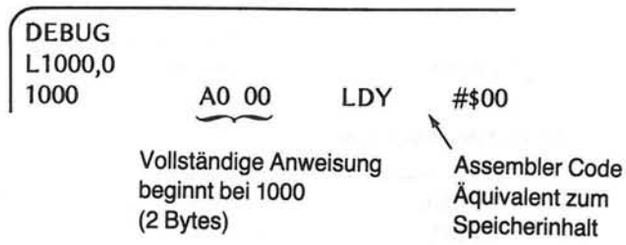
(d) Sie sehen, daß der Speicher in b und c in einem Block von 8 Zellen pro volle Zeile ausgegeben wurde. Dies ergibt sich dann, wenn die geforderte Startadresse als letzte Stelle eine 0 oder 8 hat (d. h. 1000, 1008, 1010, 1018 etc.). Endet die Startadresse nicht mit 8 oder 0, so wird in der ersten Zeile ein Block mit weniger als 8 Werten ausgegeben.

Beispiele:



5. Sie können einen Speicherblock aufLISTen, dessen Inhalt von der Maschinensprache in die Assemblersprache zurückassembliert ist (DISASSEMBLED). Tippen Sie L und die Startadresse oder die Startadresse und die Endadresse, wie Sie es beim Ausgabe-Befehl tun.

(a) für nur eine Anweisung:



```
.
.
DEBUG
L1004,0
1004          18          CLC
```

Vollständige Anweisungen beginnen bei 1004 (ein Byte)

Assembler Code

(b) für mehrere Anweisungen:

```
.
.
.
DEBUG
L1000,1005
1000          A0 00       LDY  #$00
1002          A9 00       LDA  #$00
1004          18          CLC
1005          C8          INY

DEBUG
■
```

(c) da der DEBUGGER von der spezifizierten Startadresse an rückassembliert, muß diese spezifizierte Adresse zum ersten Byte einer Anweisung Ihres Programms gehören, da sonst der DEBUGGER durcheinander gerät oder den Code falsch interpretiert.

(1) Konfusion:

```
DEBUG
L1009,0
1009,03     ???
DEBUG
■
```

Falsche Adresse

Es gibt keine legale Anweisung mit dem Code 03

(2) Irreführende Adresse:

```
DEBUG
L1001,0

1001        00          BRK
```

Eine legale Anweisung, die hier aber nicht hingehört

6. Wie wir bereits in früheren Programmen gesehen haben, kann der Speicherinhalt geändert werden.
(a) Änderung bezüglich einer Adresse:

```
DEBUG
C1003<05

DEBUG
■
```

Ändert den Inhalt von 1003 auf 5

(b) Änderung von aufeinanderfolgenden Zellen. Das Komma erhöht den zugehörigen Speicherplatz.

```
DEBUG
C1000<A9,00,A0

DEBUG
C1007<02,,06

DEBUG
■
```

Bringt A 9 nach 1000
00 nach 1001
A0 nach 1002

Bringt 02 nach 1007
06 nach 1009
aber ändert 1008 nicht

7. Der Inhalt eines Speicherblocks kann in eine andere Speicherregion verschoben (MOVE) werden.

Beispiel:

Originaladressen	*Speicherinhalt*
1100	40
1101	41
1102	42
1103	43

Der Move-Befehl wird gegeben.

```
DEBUG
M1150<1100,1103

DEBUG
■
```

Block, der verschoben wird

Anfangsadresse für die verschobenen Daten

Der Speicher hat nun folgendes Aussehen:

Adresse	*Inhalt*
1100	40
1101	41
1102	42
1103	43
.	.
.	.
.	.
1150	40
1151	41
1152	42
1153	43

Diese Daten kopiert

8. Um sich von der gewünschten Verschiebung zu überzeugen (VERIFY), oder davon, daß zwei Speicherblöcke die gleichen Daten enthalten, kann man zwei Speicherblöcke miteinander vergleichen.

Beispiel:

Wir verwenden die Ergebnisse von Beispiel 7.

```
DEBUG
V1100<1150,1153

DEBUG
■
```

Block, der mit 1100-1103 verglichen werden soll

Stimmen beide Blöcke überein, erfolgt keine Ausgabe

Enthält 1151 etwa 31 anstatt 41:

```
DEBUG
V1100 1150,1153

1151 31  1101 41

DEBUG
■
```

Diskrepanz wird ausgegeben

Weitere Befehle und Variationen der in diesem Kapitel gezeigten finden sich im Atari Benutzer Manual. Sie sollten das Manual sorgfältig durchgehen und mit allen Befehlen experimentieren, um die Möglichkeiten des Assembler Moduls voll auszuschöpfen.

Da dieses ganze Kapitel eine Zusammenfassung von Assembler-Befehlen ist, ersparen wir uns die übliche Zusammenfassung. Gehen Sie gleich an die Übungen.

ÜBUNGEN

1. Die fünf Anweisungs-Felder eines **Quell-Programms** sind unten gegeben. Welche werden immer und welche nur manchmal in einer Quell-Programm-Anweisung benutzt?
   (a) Nummer der Anweisung _____
   (b) Marke _____
   (c) Op Code (Buchstaben) _____
   (d) Operand _____
   (e) Kommentar _____

2. Der Assembler sieht, abhängig vom spezifizierten Operanden, einige Zahlen als dezimal und andere als hexadezimal an. Wie werden folgende Zahlen interpretiert?
   (a) LDA #$13 _____
   (b) LDY #14 _____

3. Der SIZE-Befehl wird gegeben, und der Rechner gibt folgendes aus:

```
EDIT
SIZE
0700          0880          3C1F

EDIT
■
```

Der Rechner wird erst abgeschaltet, dann wieder angeschaltet und folgende Befehle werden erteilt. Tragen Sie die Bildschirm-Ausgabe ein, wie sie nach dem letzten SIZE-Befehl erscheint.

```
EDIT
LOMEM 900

EDIT
SIZE
_____    _____    _____
EDIT
■
```

Die Aufgaben 4 bis 10 beziehen sich auf folgendes Programm:

```
EDIT
LIST

10   *=$1000
20   CLC
30   LDA #0
40 LOOP ADC #5
50   CMP #$50
60   BNE LOOP
70 END
■
```

4. Geben Sie den Wert im Akkumulator nach jedem der ersten fünf Läufe durch die Schleife
   1. _____ 4. _____
   2. _____ 5. _____
   3. _____
5. (a) Kann der Akkumulator jemals den hexadezimalen Wert 50 haben? _____
   (b) Wenn ja, wie oft wird die Schleife ausgeführt? _____
6. Wie lautet der Befehl, um alle Zeilen von 30 bis einschließlich 60 aufzulisten?
   _____
7. Wie heißt der Befehl, der Zeile 20 löscht? _____
8. Wie heißt der Befehl, der das Wort LOOP in Zeile 40 und 60 in das Wort ROUND umändert? _____
9. Wie heißt der Befehl zur Neunumerierung des Programms, das mit der Anweisungsnummer 100 beginnt und bei dem die nachfolgenden Anweisungsnummern jeweils um 5 größer sind?
10. Geben Sie den Befehl für den Debugger Modus und schreiben Sie die sich daraus ergebende Antwort des Rechners an den Bildschirm auf.

```
EDIT
_____
_____
■
```

Bei den folgenden Aufgaben wird angenommen, daß das oben vor Nr. 4 angegebene Programm assembliert wurde und daß Sie sich jetzt im Debugger Modus befinden. Das assemblierte Objekt-Programm befindet sich nun in den folgenden Speicherzellen.

Speicher	*Inhalt*
1000	18
1001	A9
1002	00
1003	69
1004	05
1005	C9
1006	50
1007	D0
1008	FA
1009	00

1. Wie lautet der Befehl:
   (a) zur Ausführung des Programms _____
   (b) zur Verfolgung des Programms _____
   (c) zur Ausführung der ersten Anweisung _____

12. Wie sieht das Ergebnis nach Ausführung des Befehls: D1000 aus?

```
DEBUG
D1000
_____

DEBUG
■
```

13. Wie sieht die Bildschirmausgabe nach dem Befehl: L1000,0 aus?

```
DEBUG
L1000,0
_____

DEBUG
■
```

14. Welche Änderung müßten Sie an einer Speicherzelle vornehmen, wenn Sie das Programm so abändern möchten, daß es bei jedem Durchlaufen der Schleife 10 (hex 0A) statt 5 addiert?

```
DEBUG


DEBUG
■
```

15. Welcher Befehl verschiebt das Programm so, daß die erste Anweisung nach Speicherzelle 1120 kommt?

```
DEBUG


DEBUG
■
```

ANTWORTEN

1. (a) Immer
   (b) Manchmal
   (c) Immer
   (d) Manchmal (Einige Op Codes verlangen Operanden)
   (e) Manchmal

2. (a) Hexadezimal
   (b) Dezimal

3.
```
.
.
SIZE
0900                0A80                3C1F
```

4. 1. 05
   2. 0A
   3. 0F
   4. 14
   5. 19

5. (a) Ja
   (b) Sechzehnmal

6. LIST 30,50

7. DEL 20

8. REP/LOOP/ROUND/ ,A

9. REN 100,5

10.
```
EDIT
BUG

DEBUG
■
```

11. (a) G1000
    (b) T1000
    (c) S1000

12.
```
DEBUG
D1000
1000 18 A9 00 69 05 C9 50 D0

DEBUG
■
```

13.
```
DEBUG
L1000,0

1000            18              CLC

DEBUG
■
```

14.
```
DEBUG
C1004<0A

DEBUG
■
```

15.
```
DEBUG
M1120<1000,1009

DEBUG
■
```

Entwurf eines Programms (Plan)

Gewöhnlich wird ein Programm geschrieben, um damit ein bestimmtes Problem zu lösen. Dabei sollte der Lösungsweg klar festgelegt sein, *ehe* irgendein Versuch unternommen wird, ein Programm zur Lösung zu schreiben. Da die Assembler-Sprache, im Gegensatz zu BASIC, nicht interaktiv ist, bedarf jedes Assembler-Programm einer detaillierten Planung.

Muster-Problem: Addiere fünf Paare zweistelliger Zahlen und speichere die Ergebnisse

1. Bei der Analyse des gegebenen Problems sehen Sie, daß für die fünf Zahlenpaare, die addiert werden sollen, Speicherplatz benötigt wird, den wir als Datentabelle bezeichnen.
2. Auch für die fünf Ergebnisse muß Sorge getragen werden. Da zweistellige Zahlen *addiert* werden sollen, kann die Summe eine größere Stellenzahl als zwei haben. Daher müssen für jede Summe zwei Speicherplätze verfügbar sein. Diese Gruppe von Speicherplätzen bezeichnen wir als Ergebnistabelle.
3. Die Addition wird bezüglich zweier Zahlen ausgeführt, die während dieses Vorgangs aus der Datentabelle geholt und in einer anderen Tabelle abgespeichert werden müssen.
4. Sie müssen den notwendigen Speicherplatz sowohl für das Programm als auch für die Daten- und Ergebnistabellen festlegen.

Noch wissen Sie nicht, wie lang Ihr assembliertes Programm sein wird.
Sie müssen daher die Anfangsadresse für die Daten schätzen.
Wir haben für das Programm reichlich Platz gelassen.

Da Sie das Programm jetzt greifbar formuliert haben, sollten Sie es nunmehr in Funktionsblöcke unterteilt durchdenken.

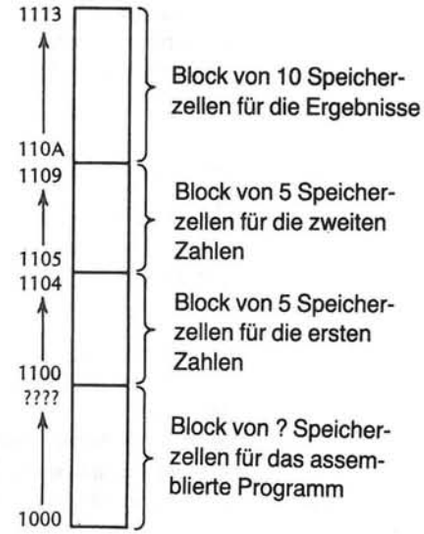


Abb. 8-1 Verwendete Speicherblöcke

Abb. 8-2 Funktionsblöcke

Betrachten wir zunächst Block A. Mittels des Debug Modus des Assembler Moduls können Sie die Daten in die dafür vorgesehenen Speicherplätze bringen. Sie können aber die Datentabellen auch vom Programm selbst laden. Die letztere Methode braucht Zeit und Speicherplatz, weshalb wir die erste verwenden wollen. Sie ist direkter, im Bedarfsfall leicht zu ändern, hält das Programm einfach und eröffnet die Möglichkeit, den Assembler Modul noch anderweitig zu nutzen. Daher reduziert sich Ihr Programm auf Block B.
Lassen Sie uns nun den Block B nach seinen einzelnen Funktionen entwickeln.

Block B

1. Lade den Akkumulator mit einer zweistelligen Zahl
2. Addiere die 2. zweistellige Zahl
3. Speichere das Ergebnis
4. Gehe zurück und wiederhole den Vorgang bis alle 5 Summen berechnet sind

Abb. 8-3 Bestandteile von Block B

Der absolute indizierte Adressierungs-Modus ist zum Laden, Addieren und Speichern der Zahlen am besten geeignet. Sie können das X- und das Y-Register zum Indizieren der Anweisungen verwenden. Das Programm könnte folgendermaßen aussehen:

```
                ADDIERE FÜNF ZAHLENPAARE

        LDY   #0        Lade das Y-Register mit dem Anfangswert 0.
                        Das indiziert die Lade- und Additionsbefehle
                        und dient gleichzeitig als Schleifenzähler.

        LDX   #0        Lade das X-Register mit dem Wert Null. X dient
                        der Indizierung des Paares gespeicherter Er-
                        gebnisse.
LOOP    CLC             Start der Schleife. Lösche das Carry Bit für die
                        spätere Addition eines Zahlenpaares.
        LDA   $1100,Y   Bringe den Inhalt von 1100 + Y in den Akkumu-
                        lator
        ADC   $1105,Y   Addiere mit Carry (Übertrag) die Zahl aus 1105
                        + Y
        STA   $110B,X   Speichere das niederwertige Byte der Summe in
                        110 B + X
        BCC   SKIP      Ohne Carry, Sprung zur Anweisung mit der Mar-
                        ke SKIP
        INC   $110A,X   Mit Carry, erhöhe den Inhalt von 110A + X um
                        eins.
SKIP    INX             Erhöhe X um 2, da Ergebnis 2 Byte lang
        INX             Erhöhe Y um eins, um das nächste Paar zu la-
        INY             den und zu addieren
        CPY   #5        Prüfe, ob Y = 5
        BNE   LOOP      Ist Y ≠ 5, springe zu der mit LOOP gekennzeich-
                        neten (bzw. „markierten") Anweisung zurück,
                        um das nächste Zahlenpaar zu verarbeiten.
END                     Ende des Programms
```

ABSOLUTES INDIZIERTES ADRESSIEREN

Die Befehle zum Laden, Addieren, Register erhöhen und Speichern werden alle im absoluten indizierten Adressierungs-Modus verwendet. In diesem Modus wird eine Basis-Adresse spezifiziert. Der im X- oder Y-Register befindliche Wert wird zur Basis-Adresse addiert, um die momentan akute Adresse zu erhalten.

Beispiele:
Wir nehmen an, das Y-Register enthalte den Wert 3, das X-Register 6, und die folgenden Werte seien im Speicher.

Speicher	*Wert*
1100	12
1101	34
1102	56
1103	78
1104	9A
1105	BC
1106	DE
1107	F0
1108	01
1109	23

Wir sehen uns nun an, was bei Ausführung der folgenden Befehle geschieht.

LDA $1100,Y lädt den Akkumulator mit dem Inhalt von 1100 + 3, oder 1103

AKKUMULATOR [78]

ADC $1105,Y addiert den Wert 78 zu dem in 1105 + 3 oder 1108 enthaltenen Wert (Wert ist 01)

AKKUMULATOR [79]

STA $110B,X speichert den Wert aus dem Akkumulator in 110B + 6 oder 1111

SPEICHER 1111 [79]

In das X- und Y-Register wird anfänglich der Wert 0 geladen. Der Y-Wert wird bei jedem Lauf durch die Schleife um 1 erhöht, während der X-Wert dabei zweimal erhöht wird. Es wird demnach ein neuer Wert geladen, ein neuer Wert addiert und das Ergebnis in einem neuen Paar von Speicherplätzen abgelegt.

Y=	*X=*	*Geladener Wert aus*	*Addierter Wert aus*	*Ergebnis gespeichert in*
0	0	1100	1105	110A und 110B
1	2	1101	1106	110C und 110D
2	4	1102	1107	110E und 110F
3	6	1103	1108	1110 und 1111
4	8	1104	1109	1112 und 1113
5	A	END OF PROGRAM		

Abb. 8-4 Speicherverwendung

Sie können nun den Rechner mit dem Assembler Modul benutzen.

ANWENDUNG DES PROGRAMMS ZUR ADDITION VON FÜNF ZAHLENPAAREN

Wenn Sie den Assembler Modul im linken Schlitz des Atari haben, können Sie zunächst über den Edit Modus das Programm eingeben.

```
.
.
.
EDIT
10      *=$1000
20      LDY #0
30      LDX #0
40    LOOP CLC
50      LDA $1100,Y
60      ADC $1105,Y
70      STA $110B,X
80      BCC SKIP
90      INC $110A,X
100     SKIP INX
110       INX
120       INY
130       CPY #5
140       BNE LOOP
150     END
■
```

Dann assemblieren Sie das Programm.

```
.
.
150 END
ASM
0000              10          *=     $1000

1000  A000         20          LDY    $0

1002  A200         30          LDX    #0

1004  18           40 LOOP     CLC

1005  B90011      50          LDA    $1100,Y

1008  790511      60          ADC    $1105,Y
```



```
100B  9D0B11      70          STA    $110B,X

100E  9003        80          BCC    SKIP

1010  FE0A11      90          INC    $110A,X

1013  E8          100 SKIP    INX

1014  E8          110         INX

1015  C8          120         INY

1016  C005        130         CPY #5

1018  D0EA        140         BNE LOOP

                  150 END

EDIT
■
```

Als nächstes müssen Sie Daten in die entsprechenden Tabellen bringen, insbesondere Nullen in die Ergebnistabelle. Dies letztere ist notwendig, um sicher zu gehen, daß die höherwertigen Bytes des Ergebnisses urspünglich wirklich Nullen enthalten. Da Sie das Carry-Bit mit dem INC-Befehl in das höherwertige Byte des Ergebnisses bringen wollen, müssen Sie sich vergewissern, daß sich anfangs eine Null im Speicher befindet. Zur Eingabe von Daten müssen Sie im Debug Modus sein.

```
.
.
140     BNE LOOP
150   END
BUG

DEBUG
■
```

Der DEBUG-Befehl zur Änderung von Werten im Speicher lautet:

CXXXX<yy Die X repräsentieren dabei die hexadezimalen Stellen der zu ändernden Adresse. Die y repräsentieren die ein- oder zweistellige Hex-Zahl, die in die angegebene Speicherzelle gebracht werden soll.

Um jeweils eine Zahl in aufeinanderfolgenden Speicherzellen zu ändern, wird der Befehl in folgender Form benutzt.

CXXXX<aa,bb,cc,dd wobei XXXX die erste zu ändernde Adresse darstellt. aa, bb... etc. sind die Hex-Werte, die von XXXX an in aufeinanderfolgende Zellen gebracht werden sollen. Es können bis zu 16 Datenwerte zusammen angegeben werden. Danach ist eine neue Startadresse nötig.

```
DEBUG
C1100<12,34,56,78,9A,BC,DE,F0,1,23,0,0 <- Tippen Sie 16 Codes
,0,0,0,0                                <- Geben Sie dann RETURN

DEBUG
C1110<0,0,0,0                           <- Geben Sie noch einmal RETURN, nachdem Sie vier weitere Nullen eingegeben haben

DEBUG
■
```

Nun sind die Daten alle da, wo sie hingehören, und das Programm ist assembliert. Zur abschließenden Überprüfung schauen Sie sich das Programm mit dem DISPLAY-Befehl im Debug Modus nochmals an.
Programmausgabe auf dem Bildschirm.

```
.
.
DEBUG
D1000,1019                              <- Programm

1000 A0 00 A2 00 18 B9 00 11            <- Ausgabe von acht Werten pro Zeile
1008 79 05 11 9D 0B 11 90 03
1010 FE 0A 11 E8 E8 C8 C0 05
1018 D0 EA

DEBUG
■
```

Vergleichen Sie die Hex-Codes mit denen auf Seite 148. Bei Übereinstimmung überprüfen Sie die Daten.


```
.
.
DEBUG
D1100,1113                              <- Datentabellen

1100 12 34 56 78 9A BC DE F0
1108 01 23 00 00 00 00 00 00
1100 00 00 00 00

DEBUG
■
```

OK, Sie können das Programm laufen lassen.

```
.
.
DEBUG
G1000
101A                A=BD X=0A Y=05 P=33 S=00
DEBUG
```

Prüfen Sie schließlich die Ergebnisse anhand der Ausgabe der Ergebnistabelle auf dem Schirm.

```
.
.
DEBUG
D110A,110B

110A 00 CE      <-      12+BC = 00CE
DEBUG
D110C,110D

110C 01 12      <-      34+DE = 0112
DEBUG
D110E,110F

110E 01 46      <-      56+F0 = 0146
DEBUG
D1110,1111

1110 00 79      <-      78+01 = 0079
DEBUG
D1112,1113

1112 00 BD      <-      9A+23 = BD
DEBUG
■
```

Unser Muster-Problem soll sich nun folgendermaßen ändern. Anstelle der paarweisen Addition der zehn Zahlen wollen wir deren Gesamtsumme haben.

Muster-Problem 2: Addiere zehn zweistellige Zahlen und speichere das Ergebnis

Wieder entwickeln wir Block B der beiden Funktionsblöcke aus Abb. 8-2. Diesmal wollen wir:

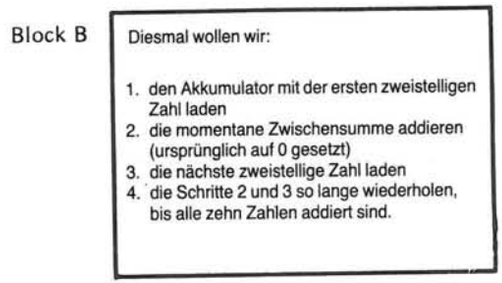


Abb. 8-5 Bestandteile von Block B - Additionsprogramm

Achten Sie auf den Unterschied zwischen beiden Lösungen für unser Muster-Problem 1 und 2.

Lösung zu Muster-Problem 2

```
        LDY   #0
LOOP    CLC
        LDA   $1100,Y
        ADC   $110B
        STA   $110B
        BCC   SKIP
        INC   $110A
SKIP    INY
        CPY   #0A
        BNE   LOOP
END
```

Lösung zu Musterproblem 1

```
        LDY   #0
        LDX   #0
LOOP    CLC
        LDA   $1100,Y
        ADC   $1105,Y
        STA   $110B,X
        BCC   SKIP
        INC   $110A,X
SKIP    INX
        INX
        INY
        CPY   #05
        BNE   LOOP
END
```

Zur Lösung des Problems 2 werden, wie Sie sehen, weniger Befehle benötigt. Die Zwischen- und die Endsumme werden in den Zellen 110A und 110B gespeichert. Die Zwischensumme wird bei jedem Lauf durch die Schleife zur zweistelligen Zahl addiert und diese Summe in dieselbe Speicherzelle zurückgebracht. Daher wird das Register X bei der Lösung von Problem 2 nicht benötigt.

Im folgenden ist die Speicherverwendung bei der jeweiligen Addition einer zweistelligen Zahl dargestellt.

Schleifen nummer	*Akkumulator geladen*	*Zwischensumme*	*110A*	*110B*
			Gespeichert	
1	12	00+12	00	12
2	34	12+34	00	46
3	56	46+56	00	9C
4	78	9C+78	01	14
5	9A	14+9A	01	AE
6	BC	AE+BC	02	6A
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

etc.

Abb. 8-6 Speicherverwendung im Programm „Addiere 10 Zahlen"

ANWENDUNG DES PROGRAMMS ZUR ADDITION VON ZEHN ZAHLEN

Aus den vorangegangenen Beispielen wissen Sie inzwischen, wie man das Programm zum Laufen bringt. Wir wollen deshalb auf die detaillierte Angabe der einzelnen Schritte, die Sie dazu benötigen, verzichten und stattdessen die Ergebnisse zeigen.

Im Edit Modus:

1. Eingabe des Programms
2. Assemblieren des Programms

Im Debug Modus:

3. Daten laden
4. Programmlauf
5. Ergebnisausgabe

1. Eingabe des Programms.

```
10     *=$1000
20      LDY #0
30    LOOP CLC
40      LDA $1100,Y
50      ADC $110B
60      STA $110B
70      BCC SKIP
80      INC $110A
90     SKIP INY
100     CPY #10
110     BNE LOOP
120    END
```

Beachten Sie, daß man die Anzahl der Schritte (10) folgendermaßen eingeben kann:
#10 für Dezimalwerte oder
#$10 für Hexadezimalwerte

Dies zeigt Hex an

2. Assemblieren des Programms.

```
ASM

0000              10          *=    $1000

1000  A000         20          LDY   #0

1002  18           30 LOOP     CLC

1003  B90011       40          LDA   $1100,Y

1006  6D0B11       50          ADC   $110B

1009  8D0B11       60          STA   $110B

100C  9003         70          BCC   SKIP

100E  EE0A11       80          INC   $110A

1011  C8           90 SKIP     INY

1012  C00A         0100        CPY   #10

1014  D0EC         0110        BNE   LOOP

                   0120 END
```

#10 dezimal

Assembliert als Hex-Wert, OA

3. Daten laden im Debug Modus

```
.
.
BUG

DEBUG
C1100<12,34,56,78,9A,BC,DE,F0,1,23,0,0,
```

Ergebnisse

Daten

4. Programmlauf

```
DEBUG
G1000
1016          A=5C X=00 Y=0A P=33 S=00
```

5. Ergebnisausgabe

```
DEBUG
D110A,110B
110A 04 5C
```

Ergebnis in 110A und 110B

EINE VARIATION DES PROGRAMMS

In aller Regel gibt es nicht nur eine Möglichkeit, um die Lösung eines Problems zu programmieren, weshalb ein guter Programmierer immer wieder überlegt, wie er die Effizienz seines Programms verbessern kann.
Schauen wir uns doch einmal ein gegenüber dem ursprünglichen nur leicht geändertes Programm an, das eine 2-Byte-Anweisung weniger benötigt.

DIE BEIDEN ADDITIONS-PROGRAMME

Erstes Programm

```
10     *=$1000
20      LDY   #0      <---***--->
30    LOOP    CLC
40      LDA   $1100,Y
50      ADC   $110B
60      STA   $110B
70      BCC   SKIP
80      INC   $110A
90     SKIP   INY     <---***--->
100     CPY   #10     <---***--->
110     BNE   LOOP
120    END
```

weggelassen

Zweites Programm

```
10     *=$1000
20      LDY    #10
30    LOOP     CLC
40      LDA    $1100,Y
50      ADC    $110C
60      STA    $110C
70      BCC    SKIP
80      INC    $110B
90     SKIP    DEY
100     BNE    LOOP
110    END
```

*** bezeichnet Programmänderungen

Beim ständig wiederholten Schleifendurchlauf zählt das zweite Programm rückwärts von 10 bis 0. Daher sind die Daten- und Ergebnistabellen im Speicher etwas anders angelegt. Jeder Wert wird um einen Platz weitergeschoben.

Die Anweisung CPY # 10 ist im zweiten Programm weggelassen. Dies ist möglich, da das Zero-Bit des Status-Registers auf 1 gesetzt wird, wenn das Y-Register beim letzten Lauf durch die Schleife den Wert 0 erreicht. Daher kann die BNE-Anweisung ohne irgendeinen Vergleich nach dem DEY-Befehl gegeben werden. Da Sie im ersten Programm von 0 bis 10 an vorwärts gezählt haben, mußte der Wert im Y-Register mit 10 verglichen werden, ehe die BNE-Anweisung verwendet werden konnte.

Ein Vergleich beider Programme zeigt, daß das zweite Programm zwei Bytes kürzer ist. Es endet in Zelle 1013, das erste dagegen erst in Zelle 1015.

Der Zugriff zu den Datentabellen ist in beiden Programmen ähnlich; die Adressen sind wegen der Rückwärtszählung im Register Y im zweiten Programm lediglich um 1 verschoben.

Erstes Programm

```
0000                10        *=    $1000
1000  A000           20        LDY   #0
1002  18             30 LOOP   CLC
1003  B90011         40        LDA   $1100,Y
1006  6D0B11         50        ADC   $110B
1009  8D0B11         60        STA   $110B
100C  9003           70        BCC   SKIP
100E  EE0A11         80        INC   $110A
1011  C8             90 SKIP   INY
1012  C00A           0100      CPY   #10   <-- Diese Anweisung wird im zweiten Programm nicht benötigt
1014  D0ED           0110      BNE   LOOP
                     0120 END
```


Zweites Programm

```
0000                10        *=    $1000
1000  A00A           20        LDY   #10   <-- Anfangswert 10 statt Null
1002  18             30 LOOP   CLC
1003  B90011         40        LDA   $1100,Y
1006  6D0C11         50        ADC   $110C   <-- Verschiedene Speicherzellen verwendet
1009  8D0C11         60        STA   $110C
100C  9003           70        BCC   SKIP
100E  EE0B11         80        INC   $110B
1011  88             90 SKIP   DEY   <-- Rückwärtszählen
1012  D0EE           0100      BNE LOOP
                     0110 END
```

Erstes Programm			*Zweites Programm*
	Daten	Speicher	Daten
Datenzugriff in dieser Reihenfolge ↓	12	1100	nicht verwendet
	34	1101	12
	56	1102	34
	78	1103	56
	9A	1104	78
	BC	1105	9A
	DE	1106	BC
	F0	1107	DE
	01	1108	F0
	23	1109	01
	Niedriges-Byte-Ergebnis	110A	23
	Hohes-Byte-Ergebnis	110B	Niedriges-Byte-Ergebnis
	Nicht benutzt	110C	Hohes-Byte-Ergebnis

↑ Datenzugriff in umgekehrter Reihenfolge

Abb. 8-7 Datentabellen

Natürlich liefern beide Programme das gleiche Ergebnis, obwohl das zweite Programm in umgekehrter Reihenfolge addiert.
Das zweite Programm enthält einen neuen Befehl.

DEY (DEcrement the Y register) Verminderung des Y-Registers
Op Code 88

Es handelt sich um eine nur im implizierten Adressierungs-Modus zu verwendende 1-Byte-Anweisung. Sie bewirkt gerade das Gegenteil von INY (Increment Y). Bei Ausführung des Befehls wird der Wert im Y-Register um 1 vermindert.

Manche Programmierer haben vielleicht etwas gegen die Verschiebung der Datentabellen. Diese Verschiebung wurde aber gerade so vorgenommen, daß der Sprungbefehl die durch das Zero-Bit gegebene Bedingung nach der DEY-Anweisung verwenden kann. Ohne diese Verschiebung wäre beim letzten Lauf durch die Schleife folgendes geschehen.

Altes Y	*Ausgeführte Anweisung*	*Summe*	*Letzter Zugriff zum Speicher*	*Neues Y*
1	SKIP DEY	04 4A	1101	0
0	BNE LOOP	04 4A		

Setzt das Zero Bit im Status-Register

Da das Zero Bit gesetzt ist, würde kein Rücksprung zur Addition der letzten Zahl erfolgen. Das Ergebnis (044A) wäre daher falsch.

VARIATION 2

Sie können noch einen anderen Weg wählen, bei dem die Datentabellen ihre ursprüngliche Position behalten. Das erfordert allerdings eine Änderung der Befehlsfolge und den Wiedereinbau des CPY-Befehls in das Programm.

Drittes Programm

```
10    *=$1000
20     LDY  #10
30    LOOP  DEY   <-- Bringe DEY hierher
40     CLC
50     LDA  $1100,Y <-- Für die Y-Werte 9,8,7,
60     ADC  $110B      6,5,4,3,2,1,0
70     STA  $110B
80     BCC  SKIP
90     INC  $110A  <-- Wieder auf den ursprünglichen Werten
100   SKIP  CPY #0
110    BNE  LOOP  <-- CPY wieder da
120   END
```

Wie Sie hier sehen, haben Sie beim Entwurf eines Programms ein beträchtliches Maß an Freiheit. Jedes Programm, das die ihm gestellte Aufgabe löst, ist ein *gutes* Programm. Meist können gute Programme im Hinblick auf Effizienz und Geschwindigkeit noch verbessert werden. Programmieren bietet großartige Möglichkeiten zu individueller Entfaltung und Kreativität. Viel Spaß dabei!

ZUSAMMENFASSUNG

- In diesem Kapitel haben Sie gelernt, wie Sie ein Assembler Programm zur Lösung eines bestimmten Problems planen. Dazu wurden folgende Schritte empfohlen:

1. Analysieren Sie das Problem und entscheiden Sie dann, wie die Lösung allgemein erreicht werden soll.
2. Legen Sie den für die Daten und Ergebnisse nötigen Speicherplatz fest.
3. Durchdenken Sie Ihr Problem in Form von Funktionsblöcken.
4. Gliedern Sie jeden Block in Schritte auf, die der Computer ausführen kann.
5. Schreiben Sie, den aufgegliederten Funktionsblöcken entsprechend, das Programm.

- Sie haben die Handhabung von Daten mittels Tabellen gelernt, auf die zum Laden, Addieren und Speichern von Daten mit dem absolut indizierten Adressierungs-Modus zugegriffen wird. Bei den in diesem Modus verwendeten Befehlen handelt es sich um:
  1. LDA $1100,Y Lade Daten aus dem Speicher in den Akkumulator (Speicherzelle 1100 + Wert im Y-Register)
  2. ADC $1105,Y Addiere Daten aus dem Speicher zu den im Akkumulator befindlichen Daten (Speicherzelle 1105 + Wert im Y-Register)

3. STA $110B,X Bringe den Inhalt des Akkumulators in den Speicher (Zelle 110B + Wert im X-Register)
4. INC $110A,X Erhöhe den Inhalt im Speicher, Zelle 110A + Wert im Register X.

- Sie haben die Eingabe von Daten im Debug Modus mit dem Befehl zur Speicheränderung gelernt.

Beispiele:

```
C1100<12,34,56,78,9A,BC,DE,F0,1,23,0,0
,0,0,0,0
```

16 Datensätze werden hintereinander eingegeben, wobei mit Zelle 1100 angefangen wird

```
C1110<0,0,0,0
```

4 weitere Datensätze werden bei 1100 beginnend hintereinander eingegeben

- Sie haben gelernt, daß ein Index-Register ebensogut rückwärts wie vorwärts zählen kann.

Rückwärts (down) → DEY Vermindere den Inhalt des Y-Registers
Vorwärts (up) → INY Erhöhe den Inhalt des Y-Registers

- Sie haben gelernt, wie man ein Programm zur Lösung eines neuen Problems abändern und Variationen von vorhandenen Programmen herstellen kann.

- Sie haben gelernt, daß es im allgemeinen verschiedene Wege gibt, um die Lösung eines Problems zu programmieren.

ÜBUNGEN

1. In welchem Modus (EDIT, ASM oder DEBUG) wurden die Dateneingaben für die Programme in diesem Kapitel in den Speicher geladen? _____
2. Welcher Adressierungs-Modus wird für den folgenden Befehl verwendet?

   STA $1500,Y

   _____
3. In welche Speicherzelle gelangt der Wert 10 bei Ausführung des Befehls aus Übung 2, wenn der Akkumulator den Wert 10 und das Register Y den Wert 5 enthalten?

   _____

4. Die folgenden Werte befinden sich, wie hier gezeigt, im Computer.

Akkumulator	*Speicher*	*Daten*	*Y-Register*
12	1100	D5	1
	1101	33	
	1102	22	
	1103	24	

Danach wird die hier angegebene Befehlsfolge ausgeführt. Tragen Sie die richtigen Daten in die entsprechenden Leerstellen ein.

```
LDA $1100,Y
ADC $1101,Y
INY
STA $1100,Y
```

Akkumulator	*Speicher*	*Daten*	*Y-Register*
	1100		
	1101		
	1102		
	1103		

5. Welche Werte würden Sie erhalten, wenn die Befehle aus Übung 4 unter Verwendung der dort erzielten Ergebnisse nochmals ausgeführt würden?

Akkumulator	*Speicher*	*Daten*	*Y-Register*
	1100		
	1101		
	1102		
	1103		

6. Was würde sich nach der Ausführung des Befehls
   C1100<A4,F3,C5,19
   in den folgenden Speicherzellen befinden?

Speicher	*Daten*
1100	
1101	
1102	
1103	
1104	

Hinweis: Eine Speicherzelle bleibt unverändert. Schreiben Sie dort XX hinein.

7. Geben Sie einen DEBUG Befehl zur Ausgabe der vier Speicherzellen-Inhalte von Aufgabe 5. _____
8. Versehen Sie die Leerstellen der unten angegebenen Tabelle mit den dem folgenden Programm entsprechenden Werten.

```
10    *=$1000
20    LDY #0
30  LOOP CLC
40    LDA $1101,Y
50    ADC $1100,Y
60    INY
70    STA $1100,Y
80    CPY #4
90    BNE LOOP
100   END
```

Ende der Schleife	*Y-Register*	*Akkumulator*	*Speicher*				
			1100	*1101*	*1102*	*1103*	*1104*
0	0	00	01	02	03	04	05
1							
2							
3							
4							

9. Geben Sie die hexadezimalen und die dezimalen Werte folgender Assembler Notationen an

#24 _____hex = _____ dezimal
#$24_____hex = _____ dezimal

ANTWORTEN

1. DEBUG
2. Absoluter indizierter Adressierungs Modus
3. 1505 (hex)
4. Akkumulator

55

Speicher	Daten
1101	D5
1101	33
1102	55
1103	24

Y-Register

2

5. Akkumulator

79

Speicher	Daten
1100	D5
1101	33
1102	55
1103	79

Y-Register

3

6.

Speicher	Daten
1100	A4
1101	F3
1102	C5
1103	19
1104	XX

← (weiß nicht)

7. D1100,1103
8.

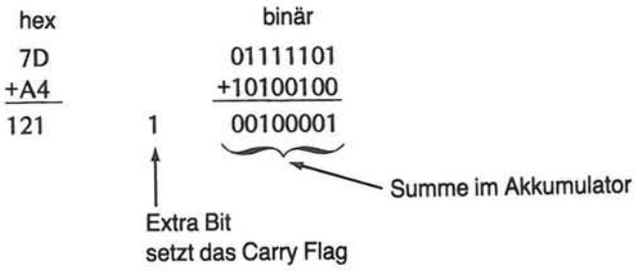
Am Ende der Schleife #	*Y-Register*	*Akkumulator*	*Speicher*				
			1100	*1101*	*1102*	*1103*	*1104*
0	0	00	01	02	03	04	05
1	1	03	01	03	03	04	05
2	2	06	01	03	06	04	05
3	3	0A	01	03	06	0A	05
4	4	0F	01	03	06	0A	0F

9. #24 18 hex = 24 dezimal
   #$24 24 hex = 36 dezimal

Addition und Subtraktion

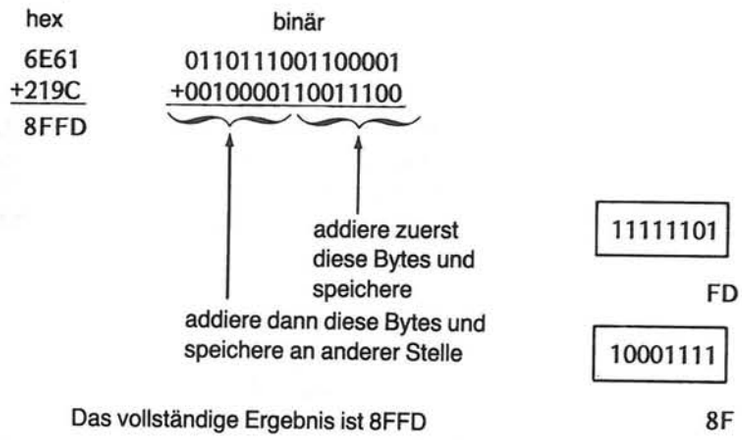
Sie sind nun damit vertraut, wie Computer Zahlen addieren und subtrahieren. Da die Größe von Registern und Speicherzellen nur 8 Binärstellen beträgt, muß die Addition und Subtraktion mit Datenteilen gleicher Größe durchgeführt werden. Wie Sie wissen, ist 255 die größte Dezimalzahl, die mit 8 Bit (einem Byte) dargestellt werden kann.
In Kapitel 6 haben Sie gelernt das Carry Bit abzufragen, um zu sehen, ob die Summe zweier 8-Bit Zahlen für den Akkumulator zu groß war.

Beispiel:



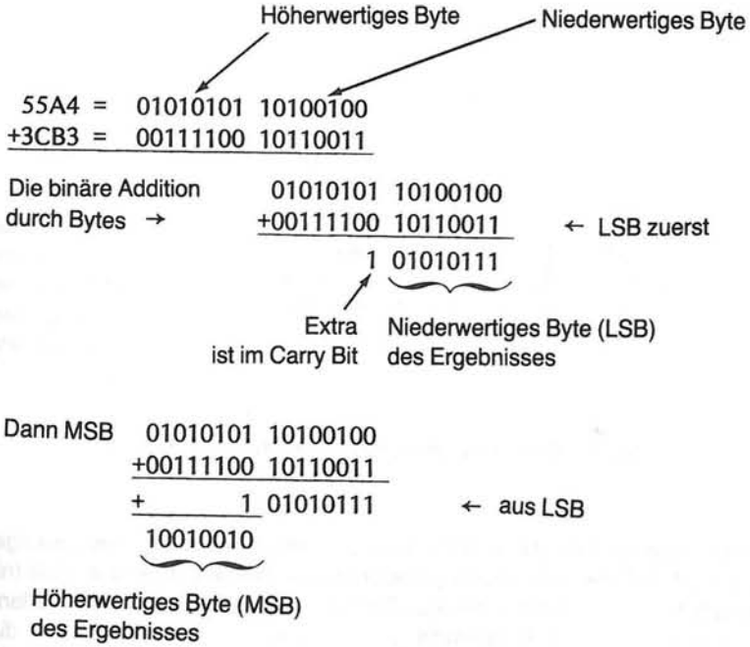
Sie können so feststellen, wann die Summe von zwei 8-Bit Zahlen zu groß ist, um in ein Register oder eine Speicherzelle zu passen. Dennoch möchte man gelegentlich Zahlen addieren, die größer sind als 255. Um dies zu bewerkstelligen, muß der Computer zur Handhabung dieser Zahl mehr als ein Byte verwenden. Der Computer kann je ein Byte von zwei Zahlen addieren, das Ergebnis speichern und sodann die nächsten zwei Bytes der beiden Zahlen addieren. Sie können dann die beiden Teile als eine vollständige Zahl ausgeben.

Beispiel:



ZWEI-BYTE ADDITION

Die schriftliche Addition von Zwei-Byte Zahlen wird uns helfen, für diese Rechenoperation ein Computer-Programm zu schreiben. Angenommen, wir wollen die beiden folgenden hexadezimalen Zahlen addieren.



```
                MSB       LSB
Endergebnis = 10010010 01010111
                 9 2      5 7      HEX
```

In diesem Beispiel liefert die Addition der niederwertigen Bytes ein Carry. Die ADC-Anweisung (Addiere mit Carry) addiert dieses Carry Bit automatisch zur Summe der beiden höherwertigen Bytes. Demnach erfolgt die Summation von Zwei-Byte Zahlen durch Addition der niederwertigen Bytes im ersten und Addition der höherwertigen Bytes im zweiten Schritt.

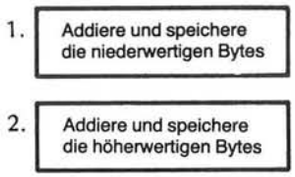


Abb. 9-1 Funktionsblöcke zur Addition

Es sieht ganz nach einem sehr geradlinigen Programm aus. Nehmen Sie sich einen Speicherblock her, um die Bytes, die addiert werden sollen, und die Bytes für das Ergebnis der Addition abzuspeichern. Da für jede Zahl zwei Bytes benötigt werden, reservieren Sie 6 Bytes.

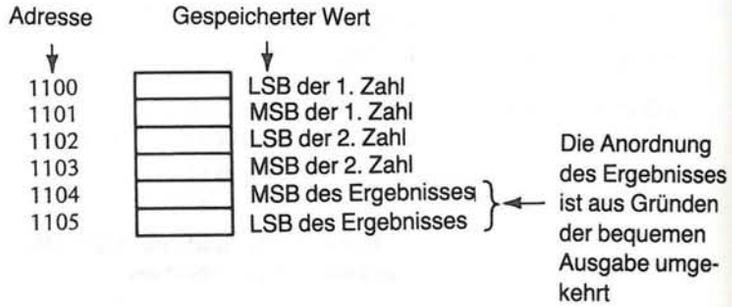


Abb. 9-2 Speicheranordnung für die Zwei-Byte Addition

Die erste Zwei-Byte Zahl, die addiert werden soll, wird in Zelle 1100 (niederwertiges Byte) und 1101 (höherwertiges Byte) gespeichert. Die zweite Zahl wird in 1102 (niederwertiges Byte) und 1103 (höherwertiges Byte) gespeichert. Diese Speicherzellen müssen *vor* Ausführung des Programms geladen werden. Sie wissen ja, wie dies im Debug-Modus gemacht wird.

Untergliedern Sie die Funktionsblöcke in Einzelschritte und verwenden Sie die Buchstaben-Codes, mit denen Sie inzwischen vertraut sind.

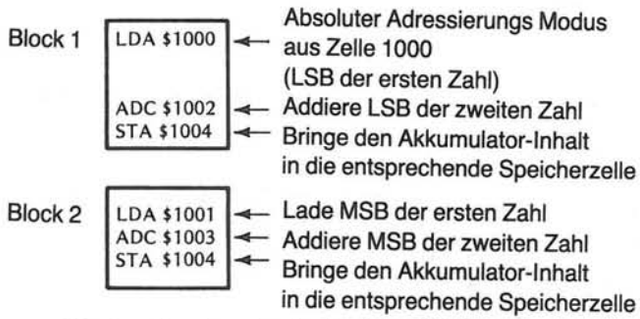


Abb. 9-3 Einzelschritte der Funktionsblöcke

Bei der Anwendung des ADC-Befehls müssen Sie daran denken, daß das Carry-Bit, falls vorhanden, zur Summe hinzuaddiert wird. Bei der Addition der niederwertigen Bytes müssen Sie daher sicher gehen, daß das Carry-Bit nicht vorhanden ist (auf Null zurückgesetzt). Demnach wird ein CLC (Lösche Carry-Bit) vor der Addition der niederwertigen Bytes benötigt.
Wenn Sie zunächst die Operationen, die ausgeführt werden müssen, in einem Ablaufprogramm (Flußdiagramm) aufzeichnen, haben Sie es leichter, das Programm Schritt für Schritt niederzuschreiben.

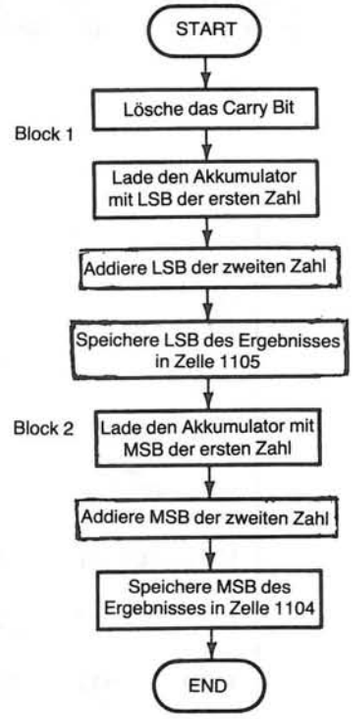


Abb. 9-4 Flußdiagramm der Zwei-Byte Addition

Wenn, wie in diesem Fall, ein Programm kurz ist und Sie ein detailliertes Flußdiagramm wie in Abb. 9-4 vor sich haben, können Sie das Programm direkt im Writer/Editor Modus eingeben.

```
EDIT
NUM 100,10                  <-- Zeilennummern ab 100 in Abständen von 10

100   *=$1000
110   CLC                   <-- Lösche Carry zur Addition von LSB
120   LDA $1100
130   ADC $1102
140   STA $1105
150   LDA $1101
160   ADC $1103             <-- Ein Carry kann zu MSB addiert werden
170   STA $1104
180   END                   <-- Drücke RETURN, um das Programm zu verlassen
190   ■
```

Beachten Sie, daß diesmal LDA, ADC und STA alle im absoluten Adressierungsmodus verwendet werden. Sie können diesen Modus verwenden, da Sie von jedem Datenanteil genau wissen, wo er sich im Speicher befindet oder wo Sie ihn abspeichern wollen. Nach der Eingabe assemblieren Sie Ihr Programm.

```
.
.
.
180  END
190
ASM                         <-- Tippe: ASM

0000                 100    *=    $1000

1000  18             110    CLC

1001  AD0011         120    LDA   $1100

1004  6D0211         130    ADC   $1102

1007  8D0511         140    STA   $1105

100A  AD0111         150    LDA   $1101

100D  6D0311         160    ADC   $1103

1010  8D0411         170    STA   $1104

                     180 END

EDIT
■
```


Nun ist das Programm assembliert. Es befindet sich in den Zellen 1000 bis 1012. Vergessen Sie die Eingabedaten nicht. Verwenden Sie diejenigen aus dem schriftlich gerechneten Beispiel, damit Sie die Ergebnisse überprüfen können:

```
  55A4
+ 3CB3
------
  9257
```

Plazieren Sie alle vier Anteile in aufeinanderfolgenden Speicherzellen.

Speicher	*Datum*	
1100	A4	LSB 1. Zahl
1101	55	MSB 1. Zahl
1102	B3	LSB 2. Zahl
1103	3C	MSB 2. Zahl

Abb. 9-5 Im Beispiel verwendete Daten

Sie haben nun alles vorbereitet, um die Daten im Debugger Modus in die in Abb. 9-5 gezeigten Speicherzellen zu bringen. Wissen Sie noch, wie dies mit der Change Memory Anweisung gemacht wird?

```
.
.
.
EDIT
BUG

DEBUG
C1100<A4,55,B3,3C           Alle vier Werte in aufeinanderfolgenden Speicherzellen
■
```

Führen Sie das Programm aus und geben Sie den Inhalt der Zellen 1104 und 1005 aus, um die Summe zu sehen.

```
.
.
.
DEBUG
C1100<A4,55,B3,3C
G1000                    <-- Ausführung

1013        A=92 X=00 Y=00 P=32 S=00
DEBUG
D1104,1105               <-- Ausgabe des Ergebnisses

1104  92  57              <-- Korrekt

DEBUG
■
```

Versuchen Sie nun, ehe Sie das Zwei-Byte Additions-Programm verlassen, das in Abb. 9-6 gegebene Zahlenpaar zu addieren. Benutzen Sie die Change Memory-Anweisung zum Speichern der Zahlen. Schreiben Sie Ihre Antworten in die entsprechenden Kästchen in Abb. 9-6. Vergleichen Sie Ihre Ergebnisse mit den Ergebnissen der zu diesem Kapitel gehörenden Übungen.

Erste Zahl	*Zweite Zahl*	*Summe*
000A	000B	
13C5	0F24	
6666	333E	
37AB	A09D	
E111	2000	

Abb. 9-6 Übungen zur Zwei-Byte Addition

Beachten Sie vor allem die letzte Aufgabe in Abb. 9-6. Können Sie das Ergebnis vorhersagen, das nach Ausführung des Programms ausgegeben wird?

WIR SPEICHERN ZWEI PROGRAMME

Nehmen wir einmal an, Sie möchten ein Additions-Programm und gleichzeitig ein Subtraktions-Programm im Rechner haben. Um unser derzeit im Rechner befindliches Additions-Programm in ein Zwei-Byte Subtraktions-Programm umzubauen, sind lediglich 3 Änderungen nötig. Es wäre Zeitverschwendung, das ganze Programm neu zu schreiben; stattdessen können Sie das Programm mit dem Move-Befehl des DEBUGGER

Modus in einen anderen Speicherteil hineinkopieren und es dort in ein Subtraktions-Programm umbauen.
Es ist Ihnen sicher aufgefallen, daß am Ende jedes Programms eine BRK-Anweisung (Op Code 00) beim Assemblieren eingeführt wird. Die END-Anweisung des Quell-Programms erzeugt den BRK-Befehl. Die BRK-Anweisung sollte bei der Verschiebung des ursprünglichen Programms vorhanden sein.

.	.	
.	.	
1000	18	Ursprüngliches Additions-Programm
1001	AD	
.	.	
.	.	
.	.	
1012	11	
1013	00	
.	.	
.	.	
.	.	
1020	18	Verschiebe das ursprüngliche Programm in diese Zellen
1021	AD	
.	.	
.	.	
.	.	
1032	11	
1033	00	
.	.	
.	.	

Wenn Sie nicht mehr wissen, wie man mit dem Move-Befehl umgeht, informieren Sie sich in Kapitel 6 oder verfahren Sie folgendermaßen:

```
.
.
.
1104  92  57              <- Antwort aus der Addition

DEBUG
M1020<1000,1013           <- Verschiebung

DEBUG
■
```

Überzeugen Sie sich von der Gleichheit beider Programme ehe Sie weitermachen, um sicher zu gehen, daß die Verschiebung korrekt ausgeführt ist.

```
.
.
.
M1020<1000,1013

DEBUG
V1020<1000,1013

DEBUG                    <-- Es wird kein Unterschied
■                            ausgegeben, d.h. kein Fehler
```

Sie haben nun alles vorbereitet, um das ursprüngliche Programm in ein Subtraktions-Programm umzubauen.

ZWEI-BYTE SUBTRAKTION

Die Subtraktion geschieht in ganz ähnlicher Weise. Das **Zwei-Byte Additions-Programm** kann durch drei geringfügige Änderungen umgebaut werden:

- In Zelle 1000:
  Ändere 18 (CLC) in 38 (SEC) ← Setze das Carry Bit anstatt es zu löschen
- In Zelle 1004 und 100D:
  Ändere 6D (ADC) in ED (SBC) ← Subtrahiere mit Borrow anstatt addiere mit Carry

Die Änderung in Zelle 1000 setzt, in völliger Analogie zum **Ein-Byte Subtraktions-Programm** in Kapitel 4, das Carry Bit zur Vorbereitung der Subtraktion. Die „Addiere mit Carry"-Anweisung (ADC) wird durch die „Subtrahiere mit Borrow" (borgen)-Anweisung (SBC) ersetzt. Die Subtraktions-Anweisung wird im absoluten Adressierungs-Modus verwendet. Die Adresse, unter der die abzuziehende Zahl gespeichert ist, folgt unmittelbar auf den Op Code, ebenso wie dies bei der ADC-Anweisung im Additions-Programm geschah. Demnach befindet sich das niederwertige Byte des zu subtrahierenden Wertes in Zelle 1102, das dazugehörende höherwertige Byte in Zelle 1103.
Änderung:

```
.
.
V1020<1000,1013

DEBUG
C1000<38
C1004<ED
C100D<ED
■
```


Sie haben nun zwei Maschinen-Programme im Rechner. Das assemblierte **Zwei-Byte Additions-Programm** begann in Zelle 1000 und wurde dann nach Zelle 1020 bis 1032 verschoben. Das **Zwei-Byte Subtraktions-Programm** befindet sich jetzt in den Zellen 1000 bis 1013. Sie können beide Programme wahlweise verwenden, indem Sie die entsprechende Adresse für den G-Befehl spezifizieren, der das Programm, welches bei dieser Adresse anfängt, ausführt.
Ehe Sie eines der Programme ausführen, bringen Sie die ursprünglichen Zahlen (55A4 und 3CB3) wieder in die Zellen 1100 bis 1103.

```
.
.
.
C1100<A4,55,B3,3C        ← Achten Sie auf die
■                          richtige Reihenfolge
```

Sie können nun beide Programme verwenden

```
.
.
.
G1000                    ← Zuerst die Subtraktion

1013        A=18 X=00 Y=00 P=32 S=00
DEBUG
D1104,1105

1104  18  F1              ← Antwort: 18F1 für Subtraktion

DEBUG
■
```

```
1104  18  F1

DEBUG
G1020                    ← Nun die Addition

1033        A=92 X=00 Y=00 P=32 S=00
DEBUG
D1104,1105               ← Ausgabe des Ergebnisses

1104  92  57              <-- Antwort: 9257 für Addition

DEBUG
■
```

Überzeugen Sie sich durch schriftliches Nachrechnen von der Richtigkeit der Subtraktion. Hoffentlich macht Ihnen das binäre Subtrahieren Spaß. Hier können Sie sich eine Menge „borgen".

```
 55A4 =  0101 0101 1010 0100
-3CB3 =  0011 1100 1011 0011
         0001 1000 1111 0001      binär
      =  1    8    F    1        hex        Ja, es stimmt
```

Versuchen Sie sich an den Übungen in Abb. 9-7. Achten Sie besonders auf die Ergebnisse der beiden letzten Aufgaben in der Tabelle. Können Sie diese erklären?

Erste Zahl	*Letzte Zahl*	*Differenz*
FFFF	0112	
76A3	6DCB	
590A	3A1B	
2222	3333	
0000	0004	

Abb. 9-7 Übungen zur Zwei-Byte Subtraktion

Das in den beiden letzten Übungen von Abb. 9-7 auftretende Problem resultiert aus der Tatsache, daß Sie versuchen, eine positive Zahl von einer kleineren positiven Zahl abzuziehen.

```
 2222        und    0000
-3333               -0004
```

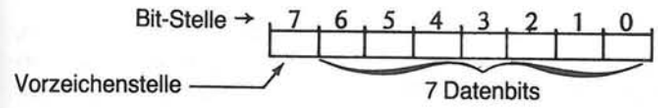
Wer ein wenig Algebra kann, weiß, daß die Ergebnisse beider Aufgaben negative Zahlen sind. Tatsächlich erschien auf dem Bildschirm folgende Ausgabe.

EEEF und FFFC

Das heißt, daß eine Methode vorhanden sein muß, mit der man entscheiden kann, ob eine Zahl positiv oder negativ ist.

NEGATIVE ZAHLEN

Man kann die Darstellung von Daten im Computer in verschiedener Weise betrachten. Wenn Zahlen mit Vorzeichen (sie sind entweder positiv, negativ oder Null) dargestellt werden sollen, so muß sie der Computer irgendwie unterscheiden können.
Betrachten wir einmal einen 8-Bit Datenblock so, als hätte er ein Vorzeichen-Bit und sieben Daten-Bits.



1. Wenn sich in der Vorzeichenstelle eine Null befindet, werden die Daten als positive Zahl angesehen.

Beispiele:

```
0 1 1 1 1 1 1 1 = +127 (64+32+16+8+4+2+1)
0 1 1 1 1 1 1 0 = +126 (64+32+16+8+4+2   )
0 1 1 1 1 1 0 1 = +125 (64+32+16+8+4   +1)
0 1 1 1 1 1 0 0 = +124 (64+32+16+8+4     )
          .            .         .
          .            .         .
          .            .         .
          .            .         .
          .            .         .
0 0 0 0 0 0 1 1 = +3   (             +2+1)
0 0 0 0 0 0 1 0 = +2   (             +2   )
0 0 0 0 0 0 0 1 = +1   (               +1 )
0 0 0 0 0 0 0 0 = +0   (                  )
```

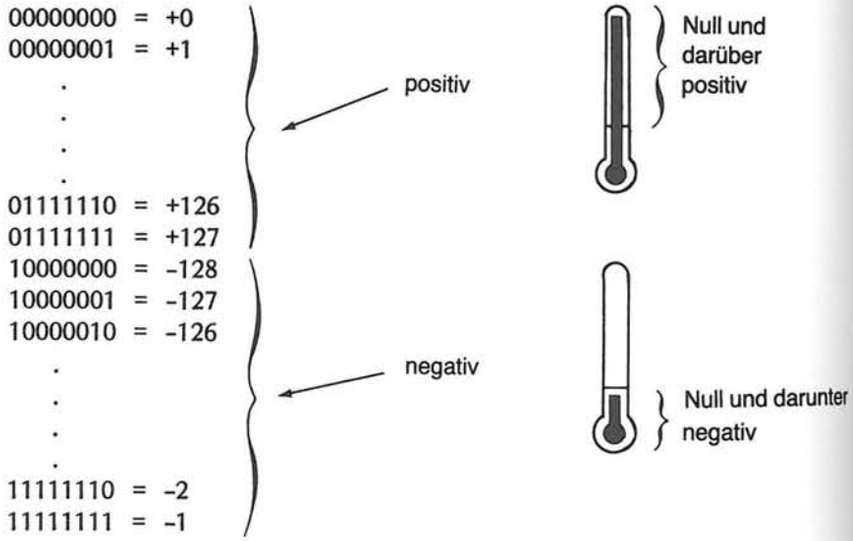
Null wird von Sprung-Anweisungen als positive Zahl angesehen

2. Wenn sich in der Vorzeichenstelle eine Eins (1) befindet, werden die Daten als negative Zahl angesehen.

Beispiele:

```
1 0 0 0 0 0 0 0 = -128
1 0 0 0 0 0 0 1 = -127
1 0 0 0 0 0 1 0 = -126
1 0 0 0 0 0 1 1 = -125
1 0 0 0 0 1 0 0 = -124
          .
          .
          .
1 1 1 1 1 0 1 1 = -5
1 1 1 1 1 1 0 0 = -4
1 1 1 1 1 1 0 1 = -3
1 1 1 1 1 1 1 0 = -2
1 1 1 1 1 1 1 1 = -1
```

Wir haben gelernt, positive Binärzahlen als positive Dezimalzahlen zu interpretieren, was aber sollen diese negativen Zeitgenossen? Sie sehen nicht gerade vertrauenerweckend aus. Man erkennt aber leicht, daß man mit einem solchen 8-Bit Code alle ganzen Zahlen von -128 bis +127 darstellen könnte.



Lassen Sie uns doch mal die negativen Zahlen auf irgendeine sinnvolle Beziehung zu ihren positiven Gegenstücken hin untersuchen.

Betrachten Sie die positive Zahl 126 = 01111110
ihr Einerkomplement = 10000001
ihr Zweierkomplement = 10000010
Vergleichen Sie das letztere mit -126 = 10000010

Die binäre Darstellung einer negativen Zahl (-1 bis -127) ist gleich dem Zweierkomplement (positives Gegenstück) ihres Betrags.

-127 = das Zweierkomplement von +127
-126 = das Zweierkomplement von +126
-125 = das Zweierkomplement von +125

-2 = das Zweierkomplement von +2
-1 = das Zweierkomplement von +1

Sie können daher die Darstellung der Ergebnisse der letzten beiden Beispiele aus Abb. 9-7 durch die folgende Convertierung der Schirmbild-Ausgabe interpretieren:

1. 2222
   -3333

EEEF hex =	1110	1110	1110	1111	binär
Nehmen Sie das Einerkomplement und addieren Sie	0001	0001	0001	0000	
Eins für das Zweierkomplement	0001	0001	0001	0001	

Deshalb, EEEF hex = negativ 1111 hex

2. 0000
   -0004

FFFC hex =	1111	1111	1111	1100	binär
Einerkomplement	0000	0000	0000	0011	
Zweierkomplement	0000	0000	0000	0100	

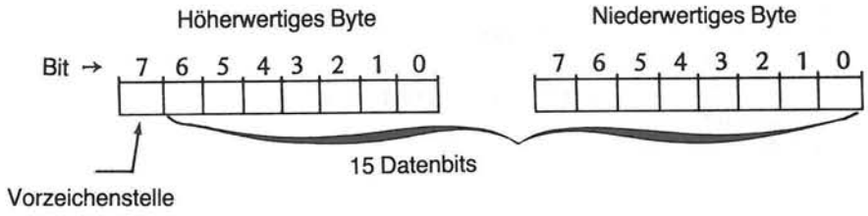
Deshalb, FFFC hex = negativ 0004 hex

Betrachten wir Daten als zwei Byte lange Hex-Zahlen mit Vorzeichen, werden alle Zahlen von 8000 bis FFFF als negative Zahlen angesehen. Die Zwei-Byte Hex-Zahlen von 0000 bis 7FFF werden als positiv angesehen.

Ohne Vorzeichen	*Mit Vorzeichen*	
0001	+0001	
0002	+0002	
0003	+0003	
.	.	
.	.	
.	.	Positiv
.	.	
.	.	
7FFD	+7FFD	
7FFE	+7FFE	
7FFF	+7FFF	
8000	-8000	
8001	-7FFF	
8002	-7FFE	
8003	-7FFD	
.	.	
.	.	
.	.	Negativ
.	.	
.	.	
FFFD	-0003	
FFFE	-0002	
FFFF	-0001	

Abb. 9-8 Zwei Byte lange Hex-Zahlen mit Vorzeichen

Bei zwei Byte langen Zahlen wird die *höchstwertige Stelle* (höchstwertiges Bit) des *höherwertigen Bytes* als Vorzeichenstelle behandelt.



Zahlen wie:

0000000000010000 = 0010 hex
0010011111100110 = 27E6 hex
0110111110101001 = 6FA9 hex

würden als positive Zahlen angesehen, hingegen Zahlen wie:

1000000000000000 = 8000 hex
1010011111100101 = A7E5 hex
1111011010111100 = F6BC hex

würden als negative Zahlen angesehen.

Eine vollständige Diskussion der Arithmetik mit vorzeichenbehafteten Zahlen geht über den Rahmen dieses Buches hinaus. Eine weitergehende Behandlung dieses Problems findet man im MOS Programming Manual, das im Fachhandel bzw. bei MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401, USA, erhältlich ist.

Für unsere Zwecke genügt die Kenntnis, daß gewisse bedingte Anweisungen die Ergebnisse auf ihr Vorzeichen hin testen. Das Testergebnis hängt davon ab, ob das Negative Flag-Bit des Status-Registers auf 1 gesetzt ist oder nicht. Dieses Bit wird auf 1 gesetzt, wenn der Rechner die Ergebnisse gewisser Anweisungen als negative Zahlen interpretiert (eine 1 in Stelle 7 als Folge der Ausführung eines Befehls). In Kapitel 5 finden Sie eine Tabelle, die die Auswirkung jeder einzelnen Anweisung auf die verschiedenen Flags des Status-Registers angibt.

MULTI-BYTE ADDITION UND SUBTRAKTION

Um Zahlen, die mehr als zwei Bytes benötigen, zu addieren bzw. zu subtrahieren, kann man die für zwei Bytes verwendete Verfahrensweise erweitern. Die Operation wird immer vom niederwertigsten Byte aus vorwärts durchgeführt (von rechts nach links).

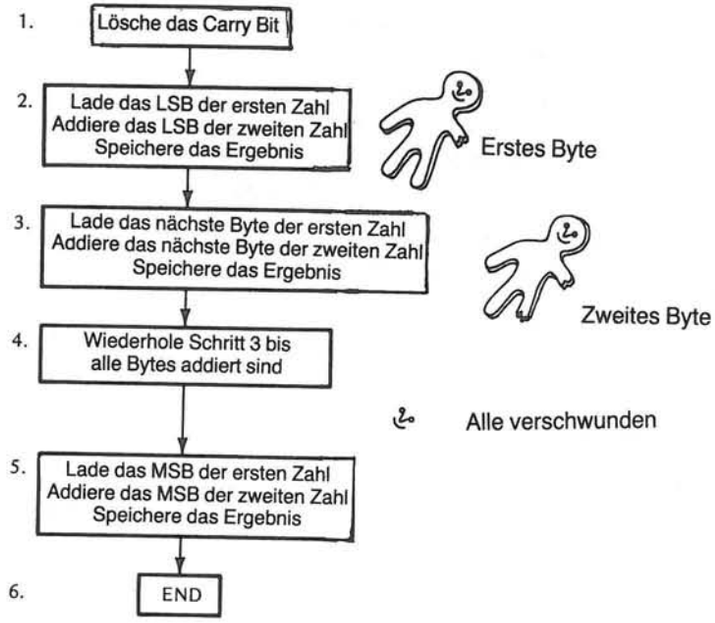


Abb. 9-9 Flußdiagramm der Multi-Byte Addition

DEZIMALARITHMETIK

Sind Sie das Convertieren von binär in hex bzw. dezimal leid? Für diesen Fall enthält die Befehlsliste 6502 eine Anweisung, die Ihnen aushilft. Wenn Sie die Werte, die Sie addieren oder subtrahieren wollen, sorgfältig als *binär-codierte Dezimalzahlen* (d.h. im BCD-Code) ausdrücken, kann der Atari diese Zahlen addieren oder subtrahieren und das Ergebnis als Dezimalzahl ausdrücken. Was sind nun binär-codierte Dezimalzahlen? Es ist die Bezeichnung für eine Binärzahl, die in zwei 4-Bit Blöcke aufgeteilt ist. Diese Teile werden dann als dezimale Ziffern interpretiert.

Beispiele:

Binärzahl	*Binärcodierte Dezimalzahl*	*Dezimalzahl*
01011000	0101 1000	58
10010011	1001 0011	93
00010110	0001 0110	16

Da je vier Bits als dezimale Ziffer interpretiert werden, muß die binäre Eingabe mit großer Sorgfalt durchgeführt werden.

11001001 = 1100 1001
KEIN BCD-Wert

10101011 = 1010 1011
KEINE BCD-Werte

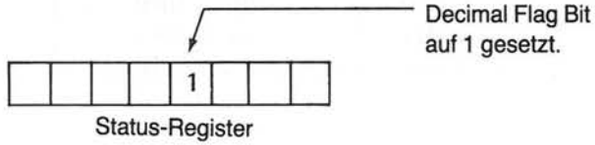
Nur folgende 4-Bit Blöcke sind möglich:

BCD	*DEZIMAL*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Der Befehl, der den Computer zur dezimalen Addition bzw. Subtraktion veranlaßt, lautet:

SED (SEt Decimal mode) Setze Dezimal Modus
Op Code = F8
Implizierter Adressierungs-Modus
Ein Byte lang
Betroffene Status Flag Bits : D

Diese Anweisung setzt das Decimal Flag Bit im Status-Register auf 1.



Ist diese Anweisung erst einmal gegeben, werden alle Additions- und Subtraktionsbefehle als dezimale Operationen ausgeführt, da das Decimal Flag-Bit auf 1 gesetzt ist. Die Durchführung irgendwelcher anderer Anweisungen bleibt davon *unberührt*. Ist in einem Programm die SED-Anweisung ausgeführt und man möchte zur binären Addition oder Subtraktion zurückkehren, muß der Rechner zuvor eine „Lösche den Dezimal-Modus" (Clear Decimal mode -CLD) -Anweisung ausführen.

CLD (Clear Decimal Mode) Lösche den Dezimal-Modus
Op Code = D8
Implizierter Adressierungs-Modus
Ein Byte lang
Betroffene Status Flag Bits: D

Diese Anweisung setzt das Decimal Flag Bit im Status-Register auf Null.
Wenn Sie die Dezimalzahlen 18 und 23 addieren möchten, können Sie das folgende Programm verwenden.

ADDIERE ZWEI DEZIMALZAHLEN

```
SED    Setze den Dezimal-Modus
CLC    Lösche das Carry Bit
LDA $1000   Lade den Akkumulator aus Zelle $1000
ADC $1001   Addiere den Wert aus Zelle $1001
STA $1100   Speichere die Summe
```

Eingeben und assemblieren:

```
.
.
EDIT
NUM 100,10

100   *=$1000              ← Eingeben
110   SED
120   CLC
130   LDA   $1100
140   ADC   $1101
150   STA   $1002
160   END
170
ASM                        ← Assemblieren

0000              100   *=$1000
```

```
1000  F8        110   SED
1001  18        120   CLC
1002  AD0011    130   LDA  $1000
1005  6D0111    140   ADC  $1001
1008  8D0211    150   STA  $1002
                160 END

EDIT
■
```

Gehen Sie zum DEBUGGER, lassen Sie das Programm laufen und geben Sie das Ergebnis aus. Vorher allerdings müssen Sie die Daten nach 1100 und 1101 bringen.

```
.
.
EDIT
BUG

DEBUG
G1000                                  ← Ausführung
100B          A=41 X=00 Y=00 P=38 S=00
DEBUG
D1102,1102                             ← Ausgabe

1102 41                                ← Ergebnis
DEBUG
■
```

Das dezimale Ergebnis von 18 + 23 ist 41. Hätten wir stattdessen die hexadezimalen Werte von 18 und 23 addiert, würde das Ergebnis 3B lauten. Wie Sie wissen, repräsentieren 18 und 23 in hexadezimaler Notation andere Werte als in dezimaler Notation.

3B hex und 41 dezimal sind *keine* äquivalenten Werte

Lassen wir nun unser Programm bei Zelle 1001 beginnend laufen (ohne SED-Anweisung), und schauen wir uns das Ergebnis an.


```
.
.
DEBUG
G1001
100B          A=41 X=00 Y=00 P=38 S=00
DEBUG
D1102,1102

1102 41                                ← Gleiches Ergebnis
DEBUG
■
```

Ist die SED-Anweisung erst einmal gegeben, werden alle Additionen im Programm im Dezimal-Modus durchgeführt. Um zur binären Addition zurückzukehren, muß die CLD-Anweisung (Lösche Dezimal-Modus) ausgeführt werden.

Der Vorteil der dezimalen Addition liegt darin, daß er Ihnen das Convertieren einer Zahl von einer Basis zur anderen zwecks Interpretation abnimmt. Sie können Dezimalzahlen (wie 23 und 18) eingeben und erhalten solche als Ergebnis (41).
Probieren Sie das Additions-Programm für zwei Dezimalzahlen mit anderen Dezimalzahl-Paaren. Ersetzen Sie die ursprünglich in 1100 und 1101 gebrachten Werte durch die neuen Zahlen. Verwenden Sie den Change Memory Befehl, um die Werte in Abb. 9-10 einzugeben.
Der erste Änderungsbefehl wäre:

```
.
.
C1100<51,23
■
```

Erste Zahl	*Zweite Zahl*	*Ausgegebene Summe*
51	23	
68	14	
29	17	
33	99	
72	45	

Abb. 9-10 Übungen zur Dezimal-Addition

Was ist mit den beiden Übungen von Abb. 9-10 los? Sie müssen in diesem Programm absichern, daß die Summe der zwei Zahlen kleiner als 100 ist. Jede Summe, die 99 überschreitet, paßt weder in den Akkumulator noch in irgendeine ander Speicherzelle. Wir haben es mit einem 8-Bit Rechner zu tun. Wenn Sie nun unbedingt weiter experimentieren wollen (das soll eine diskrete Aufforderung sein), passiert folgendes:

Letztes Übungsbeispiel:

```
.
.
C1100<72,45                    <-- Lade Daten
G1000                          <-- Ausführung
100B        A=17 X=00 P=F9 S=00
DEBUG
D1102,1102                     <-- Ergebnis-Ausgabe

1102 17                        <-- Sie wissen, es müßte 117 sein,
DEBUG                              dafür ist aber kein Platz da
■
```

Da der 6502 Mikroprozessor Datenblöcke von der Größe 8 Bit (1 Byte) verarbeitet, muß man für den Fall, in dem eine Summe so groß ist, daß sie nicht mehr in ein Byte paßt, vorsorglich etwas unternehmen. Eine entsprechende Vorgehensweise wurde in früheren Kapiteln anhand der Hexadezimalarithmetik erläutert. Sie läßt sich ohne weiteres auf das Addieren und Subtrahieren von Dezimalzahlen übertragen.

ZUSAMMENFASSUNG

In diesem Kapitel haben Sie gelernt:

- Große Zahlen in mehr als ein Byte aufzuteilen, um Zwei-Byte Arithmetik zu betreiben.
- Einzelne Bytes von Zwei-Byte Zahlen im absoluten Adressierungs-Modus aufzufinden, zu addieren (oder subtrahieren) und zu speichern.
- Wie das Carry-Bit bei der Zwei-Byte Addition und -Subtraktion verwendet wird.
- Wie Datenblöcke aus einer Speicherregion in eine andere geschafft werden.
- Wie vorzeichenbehaftete Zahlen interpretiert werden können:
  negativ, wenn das höchstwertige Bit auf 1 gesetzt ist
  positiv, wenn das höchstwertige Bit auf 0 gesetzt ist.
- Die Bedeutung der Status-Flags (Zustandsbits) im Status-Register bei Verzweigungs-Anweisungen.
- Daß Addition und Subtraktion auch bei Zahlen möglich ist, die größer als zwei Bytes sind.

- Zahlen im BCD-Code zu schreiben, so daß der Rechner dezimal addieren und subtrahieren kann.
- Wie man die SED – und die dazu passende Lösch-Anweisung benutzt, um entweder binär oder dezimal zu rechnen.

ÜBUNGEN

1. Wird die Summe der beiden Zwei-Byte Hex-Zahlen 9B und 66 zu groß, um noch in den Akkumulator zu passen? _____
2. Beschreiben Sie, wie der Rechner die Summe aus Aufgabe 1 handhabt.
   (a) Zahl im Akkumulator _____
   (b) Carry-Flag _____
   (Null, Eins)
3. Die folgenden Daten sind in den angegebenen Zellen gespeichert (oder sollen gespeichert werden).

Adresse	*Daten*
1100	3F
1111	A3
1112	

← Summe hier eintragen

   Nennen Sie die Assembler-Befehle, um die Zahl aus Zelle 1110 zu laden, die Zahl aus Zelle 1111 zu addieren und das Ergebnis zu speichern; benutzen Sie den absoluten Adressierungs-Modus.
   (a) _____
   (b) _____
   (c) _____
4. Bei der Addition oder Subtraktion von niederwertigen Bytes muß das Carry-Bit geeignet gesetzt oder gelöscht werden. Der Assembler-Befehl lautet:
   (a) für die Addition _____
   (b) für die Subtraktion _____

5. Wo würde die folgende Anweisung die Daten hinbringen?
   C1120<15,A4,32,CC

Adresse	*Daten*
1120	
1121	
1122	
1123	

6. Schreiben Sie den DEBUGGER-Befehl auf, der die Daten von ihrem derzeitigen Speicherplatz in Übung 5 zum Speicherplatz 1105 bis 1108 bringt _____

_____

7. Entscheiden Sie, ob die folgenden vorzeichenbehafteten Zahlen als negativ oder positiv interpretiert werden.
   (a) 10100111 binär _____
   (b) 7F hexadezimal _____
   (c) 01011111 binär _____
   (d) A3 hexadezimal _____

8. Zwei-Byte Zahlen werden angesehen als:
   (a) positiv von _____ bis _____
   (b) negativ von _____ bis _____

ANTWORTEN

Abb. 9-6

Erste Zahl	*Zweite Zahl*	*Summe*
000A	000B	0015
13C5	0F24	22E9
6666	333E	99A4
37AB	A09D	D848
E111	2000	0111

Das höherwertige Bit fehlt

Abb. 9-7

Erste Zahl	*Letzte Zahl*	*Differenz*
FFFF	0112	FEED
76A3	6DCB	08D8
590A	3A1B	1EEF
2222	3333	EEEF
0000	0004	FFFC

Die letzten beiden Ergebnisse sind negativ

Abb. 9-10

Erste Zahl	*Zweite Zahl*	*Ausgegebene Summe*
51	23	74
68	14	82
29	17	46
33	99	32
72	45	17

In den beiden letzten Antworten fehlt das höherwertige Bit

1. Ja
2. (a) 01
   (b) Eins (richtige Antwort ist 101)
3. (a) LDA $1110
   (b) ADC $1111
   (c) STA $1112
4. (a) CLC
   (b) SEC
5.

Adresse	*Daten*
1120	15
1121	A4
1122	32
1123	CC

6. M 1105‹1120,1123
7. (a) negativ
   (b) positiv
   (c) positiv
   (d) negativ
8. (a) 0000 bis 7FFF
   (b) 8000 bis FFFF

Verschieben und Rotieren

In Kapitel 9 haben Sie gelernt, wie man addiert und subtrahiert. Diese beiden Operationen stellen aber nur die Hälfte der vier arithmetischen Grundoperationen dar. Wie steht es mit dem Multiplizieren und dem Dividieren? Der Mikroprozessor 6502 hat für die Multiplikation und die Division keine direkten Anweisungen. Denken Sie aber einmal darüber nach, wie Sie diese Operationen beim schriftlichen Rechnen durchführen. Sie addieren, subtrahieren und verschieben mehrmals, um dem jeweiligen Teilergebnis den richtigen Stellenwert zuzuordnen.

Beispiele:

a) Multiplikation

```
  23
 x32
  46  <-- 1) Zwei 23er = 23 + 23 = 46
 69   <-- 2) Um eine Stelle nach links verschieben
      <-- 3) Drei 23er = 23 + 23 + 23 = 69
 ---
 736      4) Addiere 46 + 69 = 736
```

b) Division

```
    32
 23|736
   69  <-- 1) Drei 23er = 23 + 23 + 23 = 69
   ---
       <-- 2) Subtrahiere
   46  <-- 3) Verschiebe nach rechts
   46  <-- 4) Zwei 23er = 23 + 23 = 46
   ---
    0  <-- 5) Subtrahiere
```

Sie wissen, daß der Rechner addieren und subtrahieren kann. Ehe Sie sich jedoch an die Multiplikation und Division begeben, müssen Sie zunächst die Handhabung von Zahlen und Befehlen erlernen, die Zahlen nach links oder rechts verschieben.
Werfen wir doch noch einmal einen Blick auf die Struktur der 8-Bit Binärzahlen; Sie werden dann die Bedeutung einer Stellenänderung noch besser verstehen.

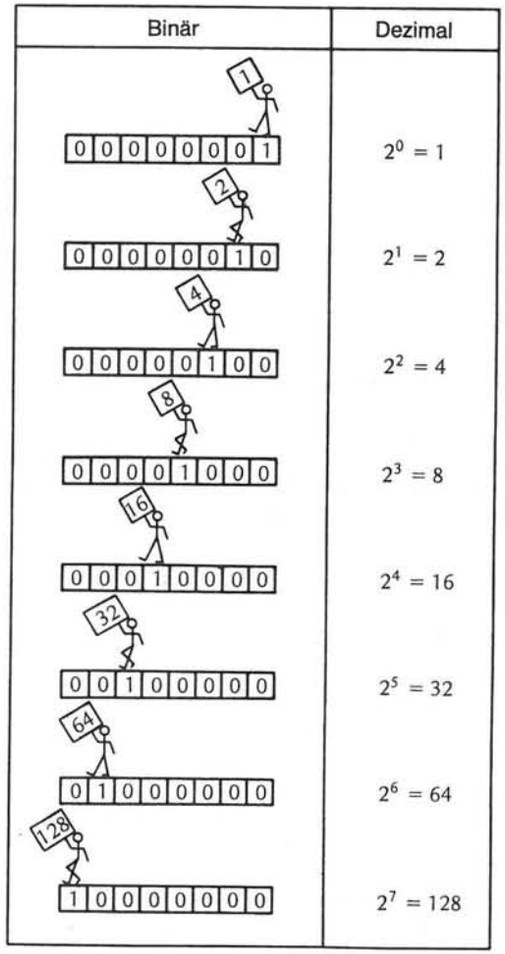


Abb. 10-1 8 Bit Binärstellenwerte

Abb. 10−1 zeigt, daß jeder Stellenwert das Doppelte des unmittelbar zur Rechten vorangegangenen Stellenwertes ist. Anders ausgedrückt, verdoppelt die Verschiebung einer Binärziffer um eins nach links den Stellenwert. Schauen Sie sich die folgenden Beispiele an, die die Wirkung einer Linksverschiebung um eine Stelle für eine Binärziffer demonstrieren.

1.	Binär	Hex	Dezimal
	00000101	5	5
	↓ ↓		
Verschoben	00001010	A	10

2.	Binär	Hex	Dezimal
	00000111	7	7
	↓↓↓		
Verschoben	00001110	E	14
3.	Binär	Hex	Dezimal
	00010010	12	18
	↓ ↓		
Verschoben	00100100	24	36
4.	Binär	Hex	Dezimal
	01101001	69	105
	↓↓ ↓ ↓		
Verschoben	11010010	D2	210

Die vorangegangenen Beispiele haben Ihnen gezeigt, daß die Linksverschiebung um eine Stelle den Wert einer 8-Bit Zahl verdoppelt. In den schriftlich gerechneten Beispielen zur dezimalen Multiplikation und Division fanden Verschiebungen statt, die den Stellenwert um Zehnerpotenzen veränderten.

```
  23
 x32
  46 <------ 2 x 23
 69  <------ Verschiebe um eine
---          Zehnerpotenz nach links,
 736         da 3 x 23 eigentlich 30 x 23 ist
```

Bei binärer Multiplikation:

```
  101
 x11
 101 <------ 1 x 101
101          Verschiebe um eine Zweierpotenz nach links
----         1 x 101 ist tatsächlich 10 x 101
1111
```

Sie tun dies eigentlich in zwei Teilen

```
  101        101
 x  1       x10
 ---        ---
  101   +  1010   =  1111
```

Die Verschiebung ist lediglich die Zusammenfassung mehrerer Einzeloperationen zu einer Einheit.

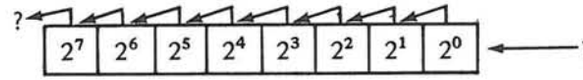
In diesem Kapitel wollen wir die vier Verschiebe- und Rotations-Anweisungen der 6502-Befehlsliste beschreiben. Wir werden auch Unterprogramme verwenden, kurze Mini-Programme, die von verschiedenen Speicherzellen aus aufgerufen werden können.
Wir werden zunächst zeigen, wie die Verschiebe- und Rotations-Anweisungen arbeiten, um danach einige Anwendungen vorzuführen.

Die vier Befehle lauten:

1. ASL (Arithmetische Linksverschiebung),
2. LSR (Logische Rechtsverschiebung),
3. ROL (Rotiere nach links),
4. ROR (Rotiere nach rechts).

ARITHMETISCHE LINKSVERSCHIEBUNG

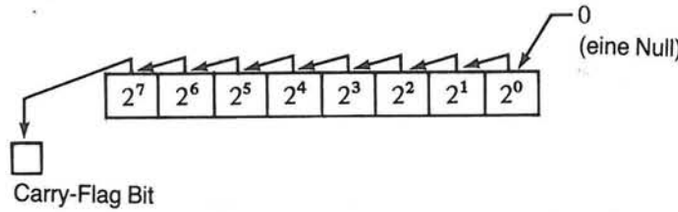
Als erstes sehen wir uns den ASL (Arithmetic Shift Left)-Befehl an. Wie wir in den vorangegangenen Beispielen sahen, verschiebt diese Anweisung jedes Bit eines bestimmten Bytes um eine Stelle nach links.
Einen Punkt hatten wir dabei noch nicht erörtert. Wenn alle Bits nach links verschoben werden, was passiert dann an den beiden Enden?



2^0 wird 2^1
2^1 wird 2^2
2^2 wird 2^3
2^3 wird 2^4
2^4 wird 2^5
2^5 wird 2^6
2^6 wird 2^7

Wohin aber geht 2^7, und woher kommt das 2^0 Bit?

Bei jeder Ausführung der ASL-Anweisung gelangt der Wert, der sich zuvor in der Linksaußenposition (2^7) befand, in die Carry-Flag-Position des Status-Registers. Alle übrigen Bits werden um eine Stelle nach links verschoben und in die Rechtsaußenposition (2^0) wird eine Null gebracht.



Das anschließende Beispiel zeigt den Inhalt des Akkumulators vor, während und nach der Ausführung der Anweisung. Vor der Ausführung befindet sich im Akkumulator der Wert B 1.

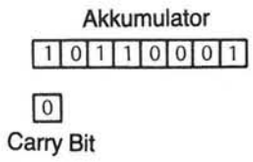


Abb. 10-2 (a) Vor der ASL-Anweisung

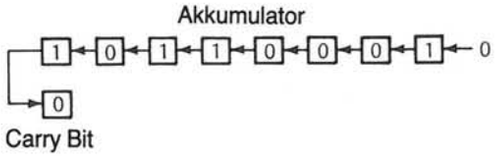


Abb. 10-2 (b) Der Ablauf der Operation

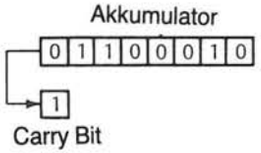


Abb. 10-2 (c) Nach Ausführung der ASL-Anweisung

Geben Sie das folgende kurze Programm ein, und Sie können sehen, wie der Rechner den Wert bei Ausführung der ASL-Anweisung im Akkumulator um eine Stelle nach links verschiebt. Das Programm bringt eine 1 in den Akkumulator und verschiebt sie um eine Stelle nach links.

```
.
.
EDIT
NUM 100,10

100   *=$1000
110   CLC
120   LDA #1      <-- Lade eine 1 (00000001 binär)
130   ASL A       <-- Verschiebe um eine Stelle nach links
140   END
150 ■
```


Assemblieren Sie das Programm.

```
.
.
150
ASM

0000          100       *=    $1000
1000  18      110       CLC
1001  A901    120       LDA   #$01
1003  0A      130       ASL   A
              140 END

EDIT
■
```

Geben Sie den DEBUGGER ein und setzen Sie mittels des Charge-Register-Befehls (Lade-Register) den Akkumulator auf Null.

```
.
.
EDIT
BUG

DEBUG
CR<0
■
```

Verfolgen Sie den Programmablauf durch Eingabe von T1000 und Drücken der RETURN-Taste.

```
.
.
T1000
1000  18          CLC                    Akkumulator
    A=00 X=00 Y=00 P=32 S=00             00000000
1001  A9 01       LDA  #$01
    A=01 X=00 Y=00 P=30 S=00             00000001
1003  0A          ASL  A
    A=02 X=00 Y=00 P=30 S=00             00000010
1004  00          BRK
    A=02 X=00 Y=00 P=30 S=00

DEBUG
■
```

Beachte den Akkumulator

Hier verschoben

Ehe Sie an das nächste Programm gehen, setzen Sie den Akkumulator wieder auf Null.

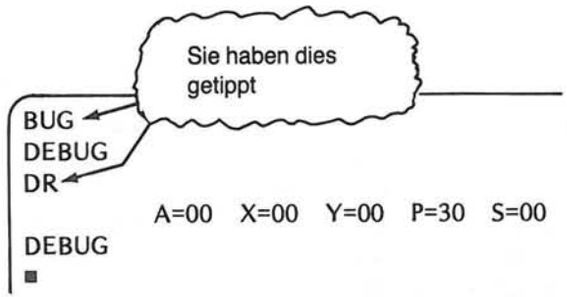
```
DEBUG
CR<0
```

Im vorangegangenen Programm haben Sie nur einmal nach links verschoben. Was geschieht aber, wenn Sie dies achtmal hintereinander tun? Bleibt irgend etwas im Akkumulator übrig? Wie sieht das Carry-Bit aus? Probieren Sie es aus.
Geben Sie das folgende Programm ein und assemblieren Sie es.

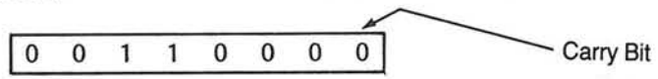
```
010   *=$1000
020   CLC
030   LDA #1
040   ASL A
050   ASL A
060   ASL A
070   ASL A
080   ASL A
090   ASL A
100   ASL A
110   ASL A
120   END
```

8 Linksverschiebungen

Dieses Programm setzt das Carry-Bit auf Null, bringt eine Eins in den Akkumulator und verschiebt dessen Inhalt achtmal nach links. Nach Ausführung des Programms sollte der Akkumulatorinhalt Null und das Carry-Bit gesetzt sein, da das die Eins im Akkumulator darstellende Bit durch jede Stelle nach links verschoben und dann aus dem Akkumulator hinaus in die Carry-Position gebracht sein müßte. Schauen Sie sich das selbst an, geben Sie „BUG" und drücken Sie RETURN. Geben Sie dann „DR" und drücken Sie nochmals RETURN. Die Register müßten folgendermaßen aussehen.



Der Akkumulator enthält Null und das Carry-Bit ist, wie Sie sehen, im Status-Register nicht gesetzt.

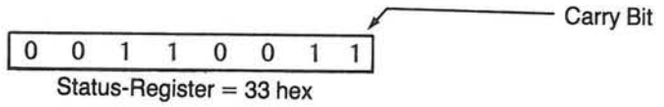


Status-Register = 30 hex

Nun führen Sie das Programm durch Eingabe von G1000 und Drücken der RETURN-Taste aus. Folgendes sollte dann erscheinen:

```
DEBUG
G1000
100B          A=00 X=00 Y=00 P=33 S=00
DEBUG
■
```

Das Status-Register ist geändert und das Carry-Bit (Bit 0) ist gesetzt.



Status-Register = 33 hex

Probieren Sie es jetzt mit der Eingabe „T1000", um den Programmablauf zu verfolgen. Sie können dann die Änderung der Register bei jedem Programmschritt sehen. Vergessen Sie nicht, nach „T1000" die RETURN-Taste zu betätigen.

```
DEBUG
T1000
1000    18          CLC
  A=00 X=00 Y=00 P=32 S=00
1001    A9 01       LDA   #$01
  A=01 X=00 Y=00 P=30 S=00
1003    0A          ASL   A
  A=02 X=00 Y=00 P=30 S=00
1004    0A          ASL   A
  A=04 X=00 Y=00 P=30 S=00
1005    0A          ASL   A
  A=08 X=00 Y=00 P=30 S=00
1006    0A          ASL   A
  A=10 X=00 Y=00 P=30 S=00
1007    0A          ASL   A
  A=20 X=00 Y=00 P=30 S=00
1008    0A          ASL   A
  A=40 X=00 Y=00 P=30 S=00
1009    0A          ASL   A
  A=80 X=00 Y=00 P=B0 S=00
100A    0A          ASL   A
  A=00 X=00 Y=00 P=33 S=00
100B    00          BRK
  A=00 X=00 Y=00 P=33 S=00
```

Carry gelöscht
Null gesetzt

Null gelöscht

Negativ gesetzt

Negativ gelöscht
Null gesetzt
Carry gesetzt

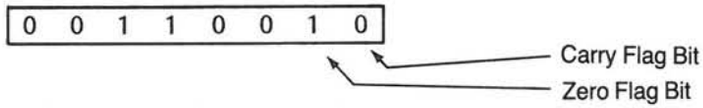
Akkumulator

00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000
00000000

1
Carry

Abb. 10-3 Verschiebungsprogramm

Nach Ausführung der ersten Programm-Anweisung (CLC) zeigt das Status-Register (Register P) den Wert 32, d. h. daß das Carry-Bit auf Null gesetzt ist. Wäre das Carry-Bit gesetzt, würde das Status-Register den Wert 33 enthalten. Das Zero-Flag Bit ist gesetzt (das zweite von rechts, bzw. Bit 1 des Registers P).



P-Register nach der CLC-Anweisung

Nach dem nächsten Schritt (LDA#$01), der eine 1 in den Akkumulator lädt, enthält das A-Register 01 und das P-Register 30, was zeigt, daß beim Laden der 1 (eines Wertes ≠ 0) das Zero-Flag Bit auf 0 gesetzt ist.

A-Register P-Register

- Beachten Sie nach jeder Linksverschiebung das 01-Muster im Akkumulator. Bei Verschiebung der 1 aus der ursprünglichen Position wird der Wert im Akkumulator verdoppelt.
- Beachten Sie auch, daß das Register P seinen Wert von 30 in BO änderte, nachdem 80 im Akkumulator auftrat. Erzeugt eine Verschiebung einen Wert, der größer ist als 7F, wird das Negative-Flag Bit gesetzt (Bit 7 im Register P). Wie Sie dem Text bereits früher entnommen haben, betrachtet der Rechner jede Zahl, deren Linksaußenposition (Bit 7) mit 1 besetzt ist (alle Hex-Zahlen zwischen 80 und FF), zu Vergleichszwecken als negativ (wie etwa bei Verzweigungen).

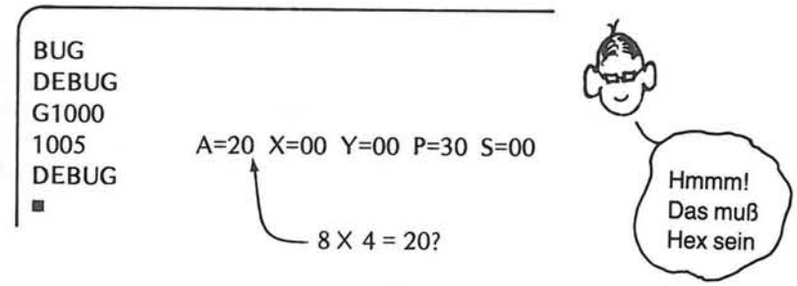
Nach Ausführung der letzten Verschiebungs-Anweisung nimmt das Register A den Wert 0 an, und das den Wert 80 repräsentierende Bit gelangt in die Carry-Position des Status-Registers. Das Negative-Flag Bit ist gelöscht (da sich im Register A der Wert 0 befindet). Im Status-Register befinden sich Einsen im Carry- und Zero-Bit.

A-Register P-Register

Wie wir bereits früher erwähnten, bedeutet die Linksverschiebung einer Zahl deren Verdoppelung. Dies kann man für ein einfaches Multiplikationsprogramm verwenden. Freilich kann dieses Programm eine Zahl nur mit einem Vielfachen von 2 multiplizieren. Wollen Sie eine Zahl mit 2 multiplizieren, dann verschieben Sie sie einmal. Sie verschieben um 2, wenn Sie mit 4 multiplizieren, um 3, wenn Sie mit 8 multiplizieren möchten, etc. Das folgende Programm ist ein Beispiel für die Multiplikation von 8 mit 4. Geben Sie das Programm ein und assemblieren Sie es.


```
010     *=$1000
020     CLC
030     LDA   #8
040     ASL   A      <-- verdoppelt
050     ASL   A      <-- nochmals verdoppelt
060     END
```

Geben Sie DEBUG ein und lassen Sie das Programm laufen



Im Akkumulator befindet sich nun die Hex-Zahl 20, was dezimal 32 bedeutet, d. h. 8x4=32, oder?
Man muß jedoch bei dieser Art der Multiplikation von zwei Zahlen vorsichtig sein. Hat man nämlich eine Zahl, wie vielleicht 80 (Hex), und versucht, diese durch eine zweifache Linksverschiebung mit 4 zu multiplizieren, erhält man sowohl im Akkumulator als auch im Carry-Bit den Wert 0.

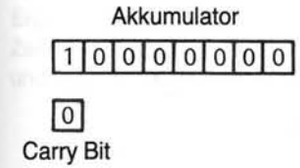


Abb. 10-4 (a) Ursprünglicher Wert

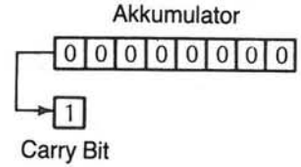
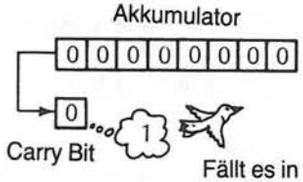


Abb. 10-4 (b) Nach einer Verschiebung



Fällt es in den Graben, dann fressen es die Raben

Abb. 10-4 (c) Nach zwei Verschiebungen

Wir haben die ASL-Anweisung im Akkumulator-Adressierungs-Modus verwendet. Diese Anweisung kann ebenso wie die übrigen Verschiebungs- und Rotations-Anweisungen in diesem Modus oder dem Zero-Page-, dem indizierten Zero-Page-, dem absoluten oder dem absoluten indizierten Modus, wie in den Tabellen in Kapitel 5 angegeben, verwendet werden.
Die ASL-Anweisung, wie auch die übrigen Verschiebungs- und Rotations-Anweisungen, beeinflußt bei ihrer Ausführung das Negative-, Zero- und Carry-Flag Bit im Status-Register.

LOGISCHE RECHTSVERSCHIEBUNG

Die LSR (Logical Shift Right)-Anweisung bewirkt letzten Endes das gleiche wie die ASL-Anweisung, nur in umgekehrter Richtung. Bei jeder Ausführung wird in die Linksaußenposition eine Null gebracht, die übrigen Bits werden um eine Stelle nach rechts verschoben, und der zuvor in der Rechtsaußenposition befindliche Wert wird in die Carry-Position gebracht.

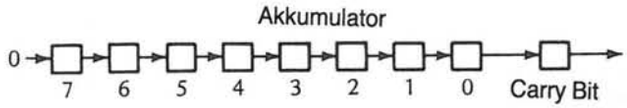


Abb. 10-5 Arbeitsweise des LSR-Befehls

Wir werden diese Anweisung anhand eines Wertes vorführen, der verschoben werden soll, indem wir ihn aus einer Speicherzelle in den Akkumulator holen und dann eine festgesetzte Anzahl von Rechtsverschiebungen durchführen. Nach jeder Verschiebung wird der im Akkumulator befindliche Wert in einer anderen Speicherzelle abgelegt. Jede weitere, nach der ersten ausgeführte Verschiebung, speichert den Wert unter der nächsthöheren Adresse.

Adresse	*Daten*
1100	Ursprünglicher Wert
1101	Zahl der Verschiebungen
1102	Ergebnis der 1. Verschiebung
1103	Ergebnis der 2. Verschiebung
1104	Ergebnis der 3. Verschiebung
1105	usw.

Abb. 10-6 Speicherverwendung bei der Rechtsverschiebung

Hier nun das Programm im einzelnen.

Anweisungen	*Kommentar*
*=$1000	Starte das Programm bei Zelle 1000
CLC	Lösche das Carry Bit
LDX #0	Setze den Speicherzeiger auf Null
LDA $1100	Hole den zu verschiebenden Wert
LDY $1101	Lade die zuvor festgesetzte Anzahl der auszuführenden Verschiebungen in Register Y
LOOP LSR A	Verschiebe einmal
STA $1102,X	Bringe es in die nächste Zelle
INX	Zeige auf die nächste Zelle
DEY	Wurde schon oft genug verschoben?
BNE LOOP	Wenn nicht, gehe zurück und verschiebe erneut

Dieses Programm holt sich die Hex-Zahl, die Sie zuvor in Zelle 1100 gebracht haben und verschiebt sie so oft nach rechts, wie Sie es in Zelle 1101 angegeben haben. Die Ergebnisse jeder Verschiebung werden, beginnend mit 1102, in die darauffolgenden Zellen gebracht, pro Verschiebung eine Zelle. Geben Sie das folgende Programm ein und assemblieren Sie es.

PROGRAMM ZUR RECHTSVERSCHIEBUNG

```
010     *=$1000
020     CLC
030     LDX   #0
040     LDA   $1100
050     LDY   $1101
060   LOOP LSR A
070     STA   $1102,X
080     INX
090     DEY
100     BNE   LOOP
110   END
```

Geben Sie nun DEBUG ein und bringen Sie einen beliebigen Wert, den Sie verschieben wollen, nach Zelle 1100. Nehmen wir einmal an, es wäre die Hex-Zahl 80. Ihre Eingabe:

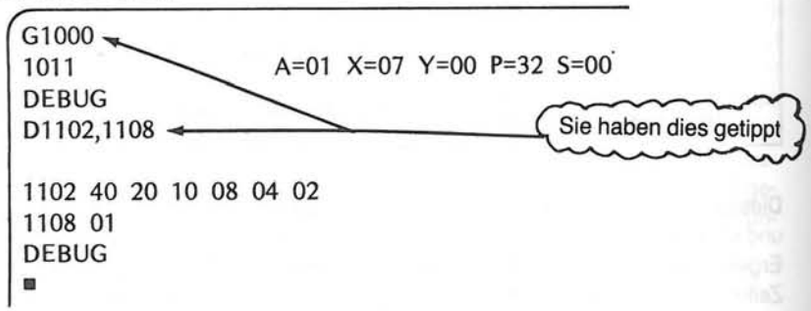
Geben Sie nun die gewünschte Anzahl von Verschiebungen in Zelle 1101. Handelt es sich z. B. um 7, tippen Sie:

```
C1101<07
```

Sie haben jetzt:

Adresse	*Daten*
1100	80
1101	07

Sie können nun das Programm laufen lassen. Tippen Sie G1000 und drücken Sie RETURN. Um die Ergebnisse anzuschauen, tippen Sie D1102, 1108. Sie können so den Inhalt der sieben Zellen sehen, die Sie zum Speichern der sieben verschobenen Werte verwendet haben.



Und da sind sie! Achten Sie auch auf die Werte, die sich nach Programmende in den Registern befinden:

- Der Akkumulator enthält das zuletzt verschobene Ergebnis.
- Das Register X enthält die Anzahl der im Akkumulator vorgenommenen Verschiebungen.
- Das Register Y enthält O, d. h. alle sieben Verschiebungen sind tatsächlich durchgeführt.
- Im Status-Register P ist das Carry-Bit nicht auf 1 gesetzt. Das erklärt sich dadurch, daß nicht genügend Verschiebungen ausgeführt wurden, um die 1 in das Carry-Bit hineinzuschubsen.

Wenn Sie sich die Daten in den Speicherplätzen 1102 bis 1108 anschauen könnten, würden die einzelnen Verschiebungen offensichtlich.

Adresse	*Binärdaten*
1102	01000000
1103	00100000
1104	00010000
1105	00001000
1106	00000100
1107	00000010
1108	00000001

Versuchen Sie nun, 8mal zu verschieben, um zu sehen, was dann passiert. Ändern Sie den Inhalt von 1101 entsprechend und lassen Sie das Programm erneut laufen. Lassen Sie diesmal die Inhalte von 1102 bis 1109 ausgeben.

```
C1101<8
G1000
1011          A=00 X=08 Y=00 P=33 S=00
D1102,1109

1102 40 20 10 08 04 02
1108 01 00
DEBUG
■
```

Sie sehen, daß das Status-Register nach dem Programmlauf 33 enthält. Durch die letzte Verschiebung wurde das Carry-Bit auf 1 gesetzt.
Geben Sie die Werte an, die sich nach Ausführung des Programms mit den folgenden Speicherwerten in den vier Registern befinden würden.

Adresse	*Daten*
1100	80
1101	0A

A= _____ X= _____ Y=_____ P=_____

Versuchen Sie, die Werte in Abb. 10–7 in 1100 und 1101 einzugeben. Tragen Sie dann in diese Tabelle die nach Programmablauf im Speicher befindlichen Werte ein.

(a)

Adresse	Daten
1100	91
1101	03
1102	
1103	
1104	

(b)

Adresse	Daten
1100	33
1101	04
1102	
1103	
1104	
1105	

(c)

Adresse	Daten
1100	E7
1101	06
1102	
1103	
1104	
1105	
1106	
1107	

Abb. 10-7 LSR-Übungen

Die zwei Rotations-Befehle (ROL, ROR) bewirken, bis auf einen einzigen wesentlichen Unterschied, im Grunde das gleiche. Während bei der Verschiebung das Carry-Bit herausgeworfen und an einem Ende des Byte eine Null nachgeschoben wird, gelangt hier das Bit wieder in das ursprüngliche Byte zurück.

LINKSROTATION

Der Befehl zur Linksrotation (ROL) veranlaßt den Rechner, sich die Bit-Positionen in kreisförmiger statt linearer Anordnung vorzustellen.

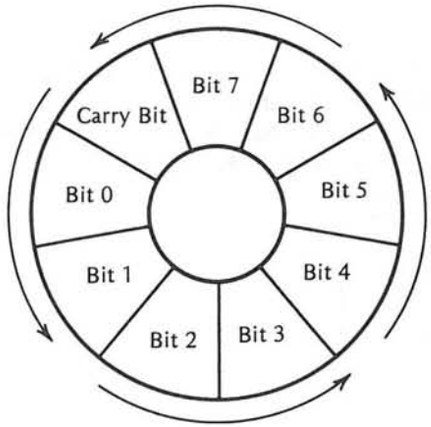
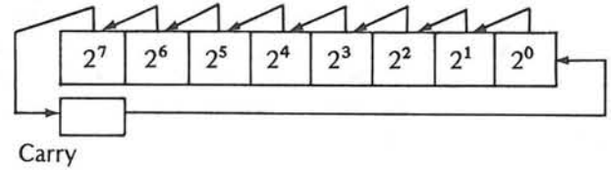


Abb. 10-8 Linksdrehung des Rades

Oder in konventioneller Darstellung:

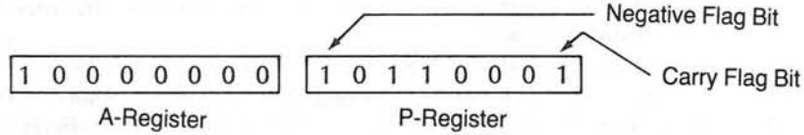


Auch bei einer großen Anzahl hintereinander ausgeführter Drehungen geht kein Bit verloren. Sie wandern stets nur im Kreis an eine jeweils neue Stelle. Es ist etwa so wie bei dem Spiel „Die Reise nach Jerusalem", nur wird niemandem der Stuhl weggenommen. Für jedes Bit ist immer ein Platz da, auf den es gelangt.
Das nächste Programm setzt das Carry-Bit auf 1 und lädt den Akkumulator mit 1. Dann wird der Akkumulator 8mal linksrotiert bzw. linksherum gedreht. Bei der letzten Drehung wandert die durch den ganzen Akkumulator gelaufene 1 in die Carry-Position. Im Akkumulator steht danach 80, da die ursprüngliche 1 aus der Carry-Position bis zu dieser Stelle den Akkumulator durchlaufen hat.
Geben Sie das Programm ein und assemblieren Sie es.

```
010     *=$1000
020     SEC
030     LDA   #1
040     ROL   A
050     ROL   A
060     ROL   A
070     ROL   A
080     ROL   A
090     ROL   A
100     ROL   A
110     ROL   A
```

Lassen Sie nun das Programm laufen, indem Sie den Debugger-Modus eingeben und G1000 tippen (RETURN nicht vergessen). Folgendes sollte erscheinen:

```
.
.
G1000
100B          A=80 X=00 Y=00 P=B1 S=00
DEBUG
■
```



Wie Sie sehen, ist das Status-Register P in der Carry-Flag- und in der Negative-Flag-Position jeweils 1. Die 1 in der Carry-Position hat zuvor den ganzen Akkumulator durchlaufen, um ihn am Ende zu verlassen und in Carry zu landen. Das Negative-Bit wurde auf 1 gesetzt, als der Akkumulator durch Drehung den Wert C0 erreichte (und blieb bis 80 auf 1).
Untersuchen Sie den Ablauf des folgenden Programms.

```
DEBUG
T1000
1000      38              SEC
      A=00 X=00 Y=00 P=31 S=00
1001      A9 01           LDA  #$01
      A=01 X=00 Y=00 P=31 S=00
1003      2A              ROL  A
      A=03 X=00 Y=00 P=30 S=00
1004      2A              ROL  A
      A=06 X=00 Y=00 P=30 S=00
1005      2A              ROL  A
      A=0C X=00 Y=00 P=30 S=00
1006      2A              ROL  A
      A=18 X=00 Y=00 P=30 S=00
1007      2A              ROL  A
      A=30 X=00 Y=00 P=30 S=00
1008      2A              ROL  A
      A=60 X=00 Y=00 P=30 S=00
1009      2A              ROL  A
      A=C0 X=00 Y=00 P=B0 S=00
100A      2A              ROL  A
      A=80 X=00 Y=00 P=B1 S=00
100B      00              BRK
      A=80 X=00 Y=00 P=B1 S=00
DEBUG
■
```

Im ersten Schritt wird Carry durch die SEC-Anweisung auf 1 gesetzt. Dann wird der Akkumulator mit 1 geladen. Nach der ersten ROL-Anweisung wandert die Null aus der Linksaußenposition des Akkumulators in Carry, löscht also Carry (wie P=30 zeigt). Der alte Carry-Wert 1 wurde in die Rechtsaußenposition des Akkumulators gebracht. Alle übrigen Bits im Akkumulator wurden um eine Stelle nach links verschoben, womit der Akkumulator den Wert 03 erhält.
Abb. 10–9 zeigt die Binär-Resultate jedes einzelnen Programmschrittes.
Nach Ausführung der 7. Rotations-Anweisung enthält der Akkumulator den Wert C0. Damit gelangt eine 1 in die Negative-Flag-Position, womit das Status-Register P den Wert B0 aufweist (P=B0).

Schritt	Anweisung	Carry	Akkumulator	
1000	SEC	1	00000000	
1001	LDA #01	1	00000001	So wird rotiert
1003	ROL A	0	00000011	
1004	ROL A	0	00000110	
1005	ROL A	0	00001100	
1006	ROL A	0	00011000	
1007	ROL A	0	00110000	
1008	ROL A	0	01100000	
1009	ROL A	0	11000000	Letzte Umdrehung
100A	ROL A	1	10000000	

Abb. 10-9 Akkumulator und Carry rotieren

Nach der nächsten Drehung ist die 1 aus der Linksaußenposition des Akkumulators in Carry gelangt und bringt damit das Register P auf B1 (P=B1). Im Akkumulator bleibt der Wert 80.

RECHTSROTATION

Der Befehl zur Rechtsrotation (ROR) bewirkt das Gegenteil zur Linksrotation. Bei wiederholter Rechtsdrehung wandern die Bits im Akkumulator und in Carry im Uhrzeigersinn immer weiter.

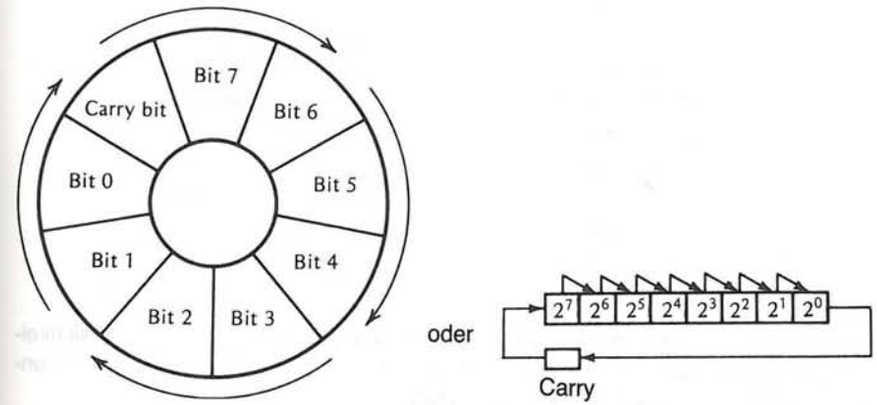
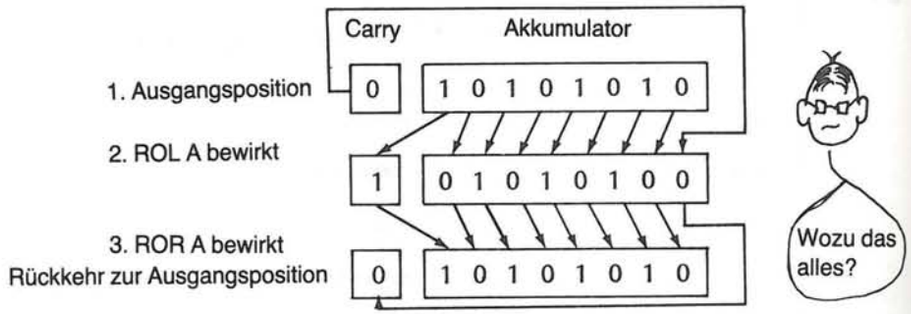


Abb. 10-10 Rechtsdrehung des Rades

Führt man nach einer Linksdrehung eine Rechtsdrehung durch, so befinden sich danach alle Bits wieder in ihrer Ausgangslage.

Beispiel:



Ich kann mir zwar nicht vorstellen, daß irgendjemand so etwas vorhat, aber vielleicht fällt Ihnen ein Grund ein.
Um die ROR-Anweisung zu demonstrieren, nehmen Sie sich noch einmal das **Programm zur Rechtsverschiebung** vor und schauen Sie, ob es sich in ein **Programm zur Rechtsrotation** umwandeln läßt. Nur dies eine Mal sorgen Sie für folgende Anfangswerte:

- Carry-Bit = 1
- Zelle 1100 = 0
- Zelle 1101 = 0
- Drehe (rotiere) Zelle 1100 achtmal nach rechts
- Speichere das Ergebnis jeder Drehung in aufeinanderfolgende Zellen von 1111 bis 1118

Sie können das folgendermaßen erreichen:

```
0100     *=$1000
0110     SEC
0120     LDY   $1101
0130   LOOP ROR $1100
0140     LDA   $1100
0150     STA   $1111,Y
0160     INY
0170     CPY   #8
0180     BNE   LOOP
0190   END
```

Verschieben und Rotieren kann, wie Sie wissen, im absoluten Adressierungs-Modus geschehen

Bringe die Ergebnisse nach 1111 bis 1118

Wir haben das Programm etwas geändert, um zu zeigen, daß es mehrere Möglichkeiten gibt, um das gleiche Resultat zu erzielen. Das Y-Register dient nun als Schleifenzähler und als Index zur Speicherung des Ergebnisses jedes „Drehs".

Geben Sie das Programm ein und assemblieren Sie es. Geben Sie dann über den DEBUGGER die Anfangswerte in den Speicher.

```
.
.
BUG

DEBUG
C1100<0,0
■
```

Sowohl 1100 als auch 1101 werden hierdurch mit Null geladen

Verfolgen Sie den Programmablauf in Einzelschritten und beobachten Sie die Änderung des aus 1100 geladenen Akkumulator-Wertes bei jeder Ausführung des Befehls unter 1007. Achten Sie auch auf die Änderung des Registers Y bei 100D.

```
.
.
C1100<0,0
S1000
1000     38            SEC
   A=00 X=00 Y=00 P=33 S=00
DEBUG
S
1001     AC 01 11      LDY      $1101
   A=00 X=00 Y=00 P=33 S=00
DEBUG
S
1004     6E 00 11      ROR      $1100
   A=00 X=00 Y=000 P=B0 S=00
DEBUG
S
1007     AD 00 11      LDA      $1100
   A=80 X=00 Y=00 P=B0 S=00
DEBUG
S
100A     99 11 11      STA      $1111,Y
   A=80 X=00 Y=00 P=B0 S=00
DEBUG
S
100D     4C            INY
   A=80 X=00 Y=01 P=30 S=00
```

Sie sehen nicht wie er es macht, aber...

da ist es

10000000 Akkumulator

Y geändert

```
.
.
DEBUG
S
100E     C0 08          CPY       #$08
   A=80 X=00 Y=01 P=30 S=00
DEBUG
S
1010     D0 F2          BNE       $1004
   A=80 X=00 Y=01 P=30 S=00
DEBUG
S
1004     6E 00 11       ROR       $1100      Rotiere nochmals
   A=80 X=00 Y=01 P=30 S=00
DEBUG
S
1007     AD 00 11       LDA       $1100
   A=40 X=00 Y=01 P=30 S=00      Da ist es
DEBUG                            01000000 Akkumulator
  .
  .
  .
```

Setzen Sie das Einzelschrittverfahren bis zum Programmende fort, und geben Sie dann den Inhalt der Zellen von 1111 bis 1118 aus.

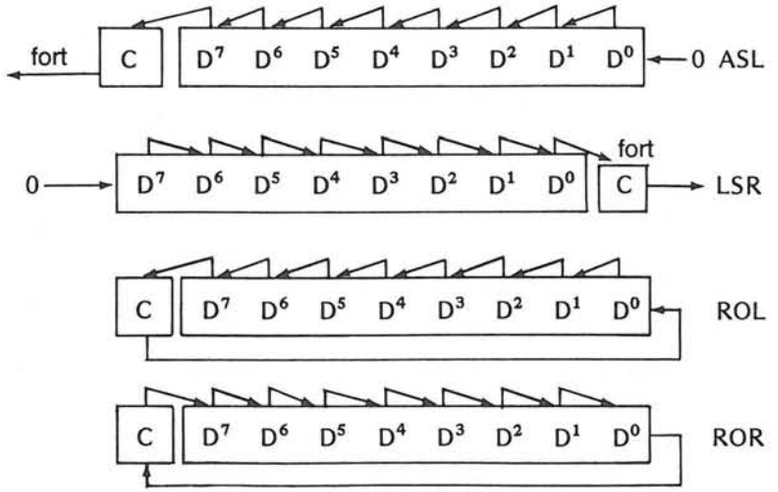
```
.
.
DEBUG
D1111,1118

1111  80 40 20 10 08 04 02
1118  01

DEBUG
■
```

Sie können nun Daten auf vielerlei Weise manipulieren. Gehen Sie an die Zusammenfassung und dann an die Aufgaben!

ZUSAMMENFASSUNG

In diesem Kapitel wurden Verschiebungs- und Rotations-Anweisungen behandelt, um Sie auf die Multiplikation und Division im nächsten Kapitel vorzubereiten. Sie haben gelernt, daß

- das Multiplizieren und Dividieren durch eine Reihe von Additionen, Subtraktionen und Stellenverschiebungen bewerkstelligt wird,
- die Linksverschiebung einer Binärstelle um einen Platz deren Stellenwert verdoppelt,
- die Linksverschiebung aller Bits einer Binärzahl um eine Stelle den Wert der Binärzahl verdoppelt,
- Verschiebungs- und Rotations-Anweisungen in den folgenden Adressierungs-Modi verwendet werden können: Akkumulator, Zero Page, Zero Page indiziert, absolut und absolut indiziert (wobei jedoch nur Register X zum Indizieren benutzt werden kann),
- die folgenden Programme die Verschiebungs- und Rotations-Anweisungen darstellen:



ÜBUNGEN

1. Wie würde, als Dezimalzahl geschrieben, das Ergebnis aussehen, wenn die folgende Binärzahl zweimal nach links verschoben würde?
   00010110 (Binärzahl vor den Verschiebungen)
   _____ Dezimalzahl nach zwei Verschiebungen

In den Aufgaben 2 bis 6 wird davon ausgegangen, daß sich die folgenden Binärziffern im Akkumulator bzw. im Status-Register P befinden. Geben Sie deren Inhalt jeweils nach Ausführung der unten spezifizierten Anweisungen wieder.

Akkumulator	0 1 0 1 0 0 1 1	Zu Beginn *jeder* Aufgabe
P-Register	0 0 1 1 0 0 0 0	
	N V B D I Z C	

2. Nach : ASL A
ASL A
ASL A

Akkumulator | | | | | | | | |

P-Register | | 0 | 1 | 1 | 0 | 0 | | |

3. Nach : LSR A
LSR A

Akkumulator | | | | | | | | |

P-Register | | 0 | 1 | 1 | 0 | 0 | | |

4. Nach : ROL A
ROL A
ROL A

Akkumulator | | | | | | | | |

P-Register | | 0 | 1 | 1 | 0 | 0 | | |

5. Nach : ROR A
ROR A

Akkumulator | | | | | | | | |

P-Register | | 0 | 1 | 1 | 0 | 0 | | |

6. Nach : ROR A
LSR A
ROL A
ASL A

Akkumulator | | | | | | | | |

P-Register | | 0 | 1 | 1 | 0 | 0 | | |

7. Nehmen Sie an, Sie verwenden das Rechtsverschiebungs-Programm mit folgender Befehls-Sequenz:

```
DEBUG
C1100< FF,08
G1000
```

(a) Welche Bits befinden sich am Ende im Akkumulator?

(b) das Carry Bit ist _____
(1,0)

8. Geben Sie in Aufgabe 7 den Inhalt der folgenden Speicherzellen in hexadezimaler Notation an.

1102	
1103	
1104	
1105	
1106	
1107	
1108	
1109	

ANTWORTEN

Aus Abb. 10-7

A=00 X=0A Y=00 P=32

(a)

Adresse	*Daten*
1100	91
1101	03
1102	48
1103	24
1104	12

(b)

Adresse	*Daten*
1100	33
1101	04
1102	19
1103	0C
1104	06
1105	03

(c)

Adresse	*Daten*
1100	E7
1101	06
1102	73
1103	39
1104	1C
1105	0E
1106	07
1107	03

1. 88 dezimal (64 + 16 + 8)

2. Akkumulator | 1 0 0 1 1 0 0 0 |
P-Register | 1 0 1 1 0 0 0 0 | N=1, Z=0, C=0

3. Akkumulator | 0 0 0 1 0 1 0 0 |
P-Register | 0 0 1 1 0 0 0 1 | N=0, Z=0, C=1

4. Akkumulator | 1 0 0 1 1 0 0 1 |
P-Register | 1 0 1 1 0 0 0 0 | N=1, Z=0, C=0

5. Akkumulator | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

   P-Register | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | N=1, Z=0, C=1

6. Akkumulator | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |

   P-Register | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | N=0, Z=0, C=0

7. (a) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

   (b) gesetzt

8.

1102	7F
1103	3F
1104	1F
1105	0F
1106	07
1107	03
1108	01
1109	00

Multiplikation, Division und Unterprogramme

Da der Befehlsvorrat des 6502 keine Anweisungen zur Multiplikation und Division enthält, müssen Sie die Durchführung dieser Operationen selber entwickeln. Die Multiplikation kann man sich als mehrfache Addition vorstellen.

Beispiel:

3 x 12 = 12 + 12 + 12

und

5 x 16 = 16 + 16 + 16 + 16 + 16

Es ist sehr leicht, ein Programm zu schreiben, indem Sie mittels einer Schleife eine Zahl n-mal addieren, um das gleiche Resultat zu erhalten, als wenn Sie die Zahl mit n multipliziert hätten.

MULTIPLIKATION DURCH ADDITION

```
      CLC
      LDX #5          Lade X mit einer Zahl
      LDA #0
LOOP  ADC $1100       Addiere in jeder Schleife
      DEX             eine zweite Zahl
      BNE  LOOP
      END
```

Wird eine zweite Zahl (z. B. 10) in die Zelle 1100 geladen und das Programm assembliert und ausgeführt, enthält danach der Akkumulator das gewünschte Ergebnis.

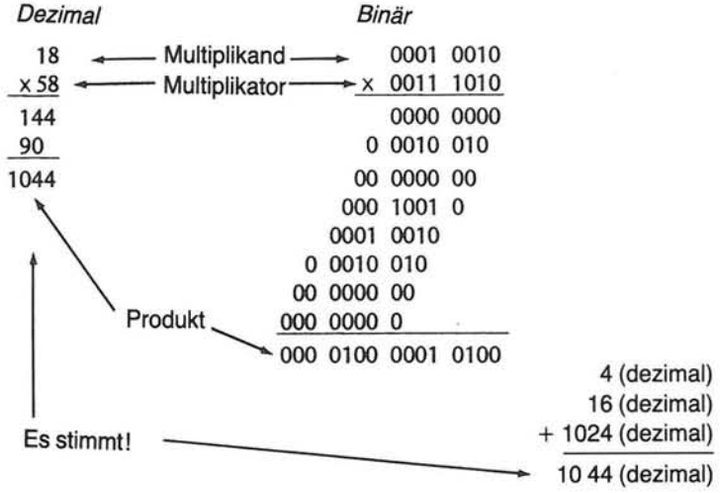
Man sieht unmittelbar, daß ein Programm für diese Methode leicht und schnell geschrieben werden kann, dies bei der Multiplikation großer Zahlen aber sehr zeitaufwendig ist. Die Schleife müßte sehr oft durchlaufen werden. Das Programm funktioniert bei 8-Bit Zahlen, müßte jedoch bei Zahlen, für deren Darstellung 8 Bit nicht ausreichen, neu entworfen werden. Seiner mangelnden Effizienz wegen wird es allerdings selten benutzt.

In ganz ähnlicher Weise kann man dividieren. Man subtrahiert einfach mit einer Schleife den Teiler, was freilich die gleichen Nachteile wie die besprochene Multiplikationsmethode hat.

Da Sie in Kapitel 10 alles über das Verschieben und Rotieren kennengelernt haben, lassen Sie uns doch versuchen, dies bei der Multiplikation und Division zu nutzen.

ACHT-BIT MULTIPLIKATION

Wir wollen uns vorderhand auf 8-Bit Zahlen beschränken und als erstes die Multiplikation ansehen. Da der Rechner seine Arithmetik bekanntlich mit Hilfe von Binärzahlen bewältigt, lassen Sie uns zunächst einmal eine schriftliche Binär-Multiplikation anschauen. Multiplizieren Sie 18 (dezimal) und 58 (dezimal) dezimal und binär.

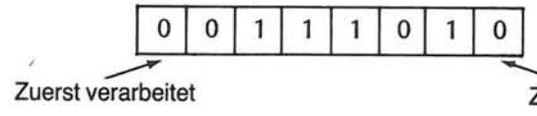


Beachten Sie, daß bei der Multiplikation der Multiplikand immer dann addiert wird, wenn im Multiplikator eine 1 auftritt. Natürlich wird jedesmal, wenn ein neues Bit des Multiplikators verarbeitet wird, einmal nach links verschoben, gerade so, wie es beim dezimalen Multiplizieren geschieht. Auch beim nacheinander Abarbeiten der Bits, sozusagen beim „Aufbrauchen", gehen Sie von rechts nach links vor.
Unser Multiplikationsprogramm wird fast das gleiche tun. Im ersten Teil des Programms werden zunächst entsprechende Werte gespeichert, die wir in folgende Zellen bringen:

Speicher-Adresse	*Inhalt*
1100	Höherwertiges Byte des Produkts
1101	Niederwertiges Byte des Produkts
1102	Multiplikand (12)
1103	Multiplikator (3A)

Abb. 11-1 Speicherverwendung bei der Multiplikation

Während des Programmablaufs wird der Akkumulator zeitweise das niederwertige Byte des Produktes enthalten. Sie werden bemerken, daß der Multiplikator (im Gegensatz zur schriftlichen Rechnung) zur Vereinfachung von links nach rechts abgearbeitet wird.



Der Multiplikator wird beim Multiplizieren nach links verschoben

Das Flußdiagramm in Abb. 11–2 erklärt die Arbeitsweise des Programms.

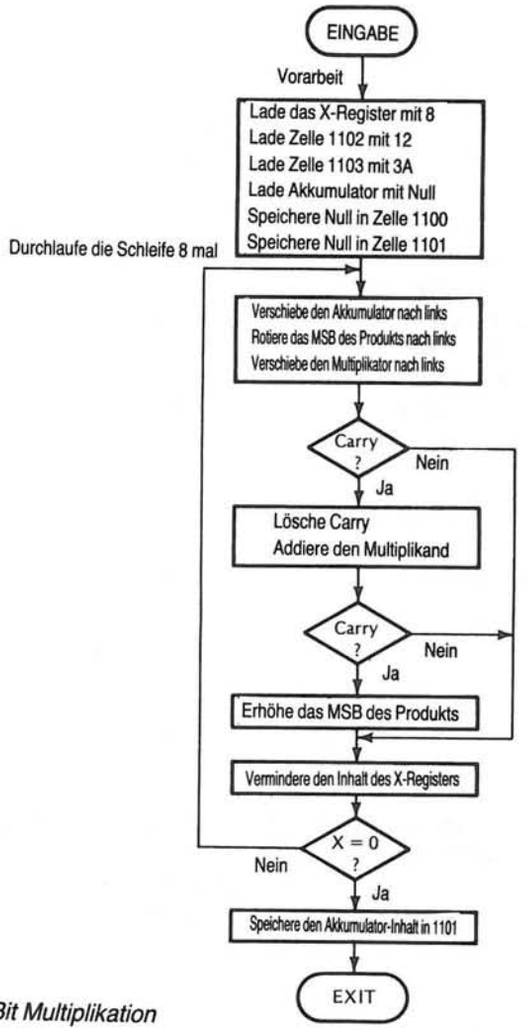


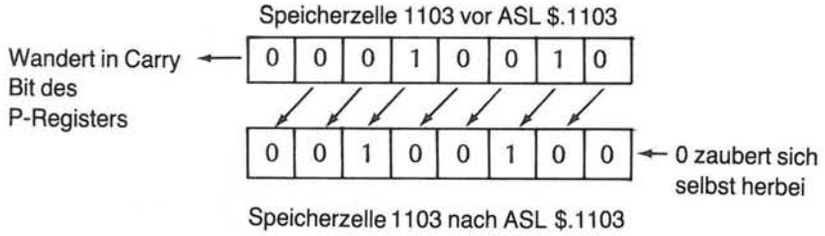
Abb. 11-2 Flußdiagramm der 8-Bit Multiplikation

Die kleinen Vorarbeiten am Anfang unseres Programms bereiten Ihnen keine Schwierigkeiten. Jede der dort vorkommenden Anweisungen haben Sie bereits früher verwendet. Es handelt sich ausschließlich um Lade- bzw. Speicher-Befehle in verschiedenen Adressierungs-Modi.
Der entsprechende Teil des Assembler-Programms sieht wie folgt aus:

```
EDIT
NUM 100,10

100    *=$1000
110    LDX     #8
120    LDA     #$12
130    STA     $1102
140    LDA     #$3A
150    STA     $1103
160    LDA     #0
170    STA     $1100
180    STA     $1101
```

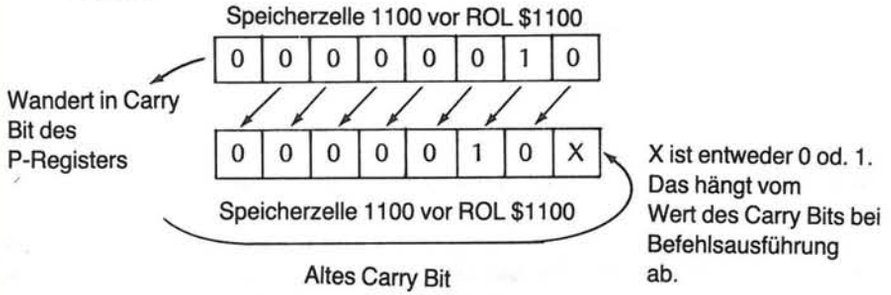
Der die Schleife enthaltende Teil des Programms führt die Multiplikation aus. Es kommen zwei Befehle vor, die Sie im letzten Kapitel kennenlernten. Einer von ihnen ist der ASL-Befehl (arithmetische Linksverschiebung). Er wird in Anweisung 190 zur Verschiebung des Akkumulator-Inhalts verwendet. In Anweisung 210 wird der Befehl im absoluten Adressierungs-Modus benutzt. In diesem Modus werden die Bits der spezifizierten Speicherzelle (1103) nach links verschoben.

Beispiel:

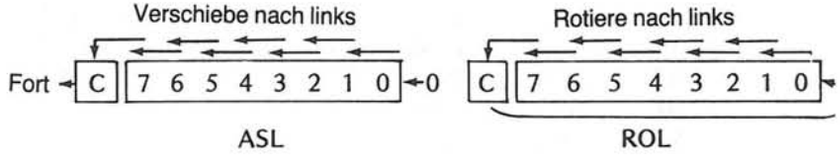


Wir werden uns zunutze machen, daß das höchstwertige Bit bei der Verschiebung in das Carry-Bit des Status-Registers P gelangt.
In Anweisung 200 wird der ROL-Befehl (Linksverschiebung) im absoluten Adressierungs-Modus benutzt. In Kapitel 10 haben Sie diesen Befehl auf den Akkumulator angewendet. Diesmal werden die Bits in Zelle 1100 „rotiert".

Beispiel:



Bei der Ausführung der einzelnen Anweisungen (ASL bzw. ROL) gelangt das höchstwertige Bit in die Carry-Stelle des Status-Registers. Bei Ausführung von ASL wird eine Null in die Rechtsaußenstelle gebracht. Bei Ausführung von ROL wird der zuvor in Carry befindliche Wert dorthin gebracht (1 oder 0). Die Namen der Befehle leiten sich aus ihrer Wirkung ab.



Hier nun das Herzstück des Programms, die Schleife.

```
190   LOOP   ASL  A
200     ROL  $1100
210     ASL  $1103
220     BCC  SKIP
230     CLC
240     ADC  $1102
250     BCC  SKIP
260     INC  $1100
270   SKIP   DEX
280     BNE  LOOP
```

Ist die Schleife 8mal durchlaufen worden, d. h. pro Bit des Multiplikators je einmal, enthält der Akkumulator das niederwertige Byte des Resultats. Es wird im Schlußteil des Programms zwecks späterer Ausgabe in Zelle 1101 abgespeichert. Damit ist das Programm zu Ende.
Das höherwertige Byte befindet sich nun in 1100, das niederwertige in 1101.

Schlußteil

```
290   STA   $1101
300   END
```

ANWENDUNG DES 8-BIT MULTIPLIKATIONS-PROGRAMMS

Geben Sie das Programm im Writer/Editor Modus ein und assemblieren Sie es. Wir haben beim Assemblieren des **Multiplikations-Programms** den Atari 820 Drucker benutzt, um das ganze Programm auf einmal dazuhaben. Es ist ein langes Programm und kann daher nicht im Ganzen auf den Bildschirm gebracht werden. Unser assembliertes Programm ergab folgenden Ausdruck:

```
0000                0100          *=     $1000
1000  A208          0110          LDX    #8
1002  A912          0120          LDA    #$12
1004  8D0211        0130          STA    $1102
1007  A93A          0140          LDA    #$3A
1009  8D0311        0150          STA    $1103
100C  A900          0160          LDA    #0
100E  8D0011        0170          STA    $1100
1011  8D0111        0180          STA    $1101
1014  0A            0190 LOOP     ASL    A
1015  2E0011        0200          ROL    $1100
1018  0E0311        0210          ASL    $1103
101B  9009          0220          BCC    SKIP
101D  18            0230          CLC
101E  6D0211        0240          ADC    $1102
1021  9003          0250          BCC    SKIP
1023  EE0011        0260          INC    $1100
1026  CA            0270 SKIP     DEX
1027  D0EB          0280          BNE    LOOP
1029  8D0111        0290          STA    $1101
                    0300 END
```

Abb. 11-3 Ausdruck des assemblierten Multiplikations-Programms

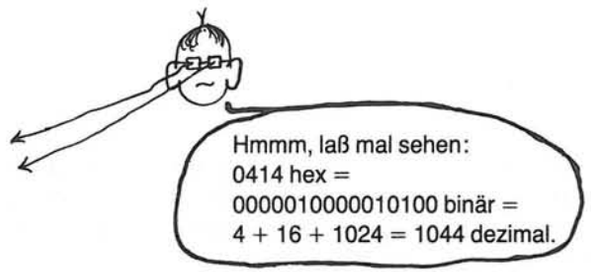
Jetzt können Sie das Programm ausführen. Sie brauchen diesmal keinerlei Daten zu laden, da Ihnen das Programm diese Arbeit abnimmt. Gehen Sie in den DEBUGGER und lassen Sie das Programm laufen.


```
EDIT
BUG

DEBUG
G1000
102C              A=14  X=00  Y=00  P=32  S=00
DEBUG
■
```

Das Ergebnis der Multiplikation befindet sich in 1100 (höherwertiges Byte) und 1101 (niederwertiges Byte). Sie können beide auf einmal ausgeben, indem Sie tippen:

```
D1100,1101
```

```
.
.
DEBUG
D1100,1101

1100  04  14
DEBUG
■
```



Es stimmt mit unserem zuvor schriftlich erzielten Ergebnis überein. Wenn Sie im Debugger Modus sind, ändern Sie Multiplikator und Multiplikand, die sich in Zelle 1003 bzw. 1008 befinden. Versuchen Sie es mit folgenden Paaren von Hex-Zahlen und tragen Sie Ihre Ergebnisse in die entsprechenden Felder von Abb. 11–4.

Multiplikator (1003)	*Multiplikand (1008)*	*Ergebnis (1100 und 1010)*
03	09	
1A	E4	
3C	D8	
A4	C2	
FF	FF	

Abb. 11-4 Multiplikationsübungen

Denken Sie an den Change-Memory-Befehl im Debugger-Modus (Speicherände-rungsbefehl). Hier sind zwei Möglichkeiten, um die erste Modifikation zu bewerkstelli-gen.

```
DEBUG
C1003<03
DEBUG
C1008<09
DEBUG
■
```

In zwei Schritten

oder

```
DEBUG
C1003<03 , , , , , 09
DEBUG
■
```

In einem Schritt

Wohl jeder Programmierer, der mit der Maschinensprache arbeitet, wird bestätigen, daß einem der Assembler eine Menge Plackerei erspart. Hätten wir das obige Multipli-kations-Programm direkt in Maschinensprache schreiben wollen, so hätten wir uns zu-sätzlich um zahlreiche Details kümmern müssen.

- Jedem Befehl und dem zugehörigen Operanden hätten individuelle Adressen zu-geordnet werden müssen.
- Sprungadressen hätten berechnet werden müssen.
- Einzelne Operationscodes hätten wir in der Befehlsliste suchen müssen.
- Wir hätten eine Convertierung zwischen dezimal und hexadezimal benötigt.

Die Programme in Atari Assembler machen das Leben leichter. Das **Writer/Editor/ Programm** kann freilich kein Programm direkt ausführen, aber es übernimmt die Zu-ordnung von Op-Codes und Operanden für das **Objekt-Programm.**
Der Assembler selbst verläßt sich hinsichtlich der Ausführung des assemblierten Pro-gramms und der Ergebnissuche im Speicher auf den **Debugger.** Die Dreier-Mann-schaft aus Editor, Assembler und Debugger ist schwer zu schlagen. Der Assembler er-ledigt die ganze Detailarbeit, und der Debugger bietet neben der Programmausführung etliche Korrekturmöglichkeiten. Schließlich erlaubt Ihnen der Editor, Ihre Wünsche dem Assembler mitzuteilen.
Nach dieser kurzen Einführung in die Multiplikation lassen Sie uns einen Blick auf die Division werfen.

ACHT-BIT DIVISION

Zunächst wollen wir uns wiederum die schriftliche Division anschauen, ehe wir uns mit dem Verfahren des Rechners befassen. Da die Division die zur Multiplikation inverse Operation ist, wollen wir, bis auf eine Ausnahme, die gleichen Zahlen verwenden, die im Beispiel für die Multiplikation vorkamen. Wir werden den Dividenden so wählen, daß unser Beispiel für die Division nicht aufgeht, d. h. ein Rest übrig bleibt.

Beispiel:

1046 (dezimal) ÷ 58 (dezimal)

Dezimal

```
      18  Quotient
58 ) 1046
     58
    ---
     466
     464
    ---
       2 Rest
```

Binär

```
                1  0010  Quotient
0011 1010 ) 0100 0001 0110
            0011 1010
            ---------
                 111 011
                 111 010
                 -------
                      10 Rest
```

Es stimmt → { 1 0010 = 12 HEX = 18 dez.
10 = 2 HEX = 2 dez.

Auch hier finden, wie bei der Multiplikation, beim Dividieren Verschiebungen statt. Der Stellenwert spielt bei diesem Vorgang eine entscheidende Rolle. Es wird nur dann sub-trahiert, wenn der Divisor (Teiler) kleiner ist als der Anteil des Dividenden, der gerade geteilt wird. In diesem Fall erhält der Quotient eine Eins. Ist der Divisor jedoch größer, bekommt der Quotient eine Null.
Unser Assembler-Programm für die Division wird fast genauso aussehen wie unser Multiplikations-Programm. Der erste Teil des Programms bringt geeignete Werte in die entsprechenden Speicherplätze, wie dies Abb. 11–5 zeigt.

Speicheradresse	*Inhalt*	
	Am Anfang	*Am Ende*
Akkumulator	Höherwertiges Byte des Dividenden	Rest
1100	Divisor	Quotient
1101	Niederwertiges Byte des Dividenden	–
1102	–	Rest

Abb. 11-5 Speicherverwendung bei der Division

Abb. 11–6 zeigt das Flußdiagramm des Programms. Zu Beginn enthalten der Akkumulator und die Zelle 1101 den ursprünglichen Dividenden. Bei jedem Lauf durch die Schleife wird der Inhalt von 1101 nach links verschoben und der Akkumulator nach links rotiert. Verursacht die Verschiebung von 1101 nach links einen Übertrag (1 nach Carry), so erscheint er bei der Rotation des Akkumulators als 1 in der Rechtsaußenposition, d. h. in der Stelle mit dem niedrigsten Stellenwert. Es wird also in jeder Schleife der Dividend um eine Stelle aus 1101 in den Akkumulator geschoben. Der Rechner vergleicht so den Divisor mit dem höherwertigen Teil des Dividenden, um gegebenenfalls zu dividieren.

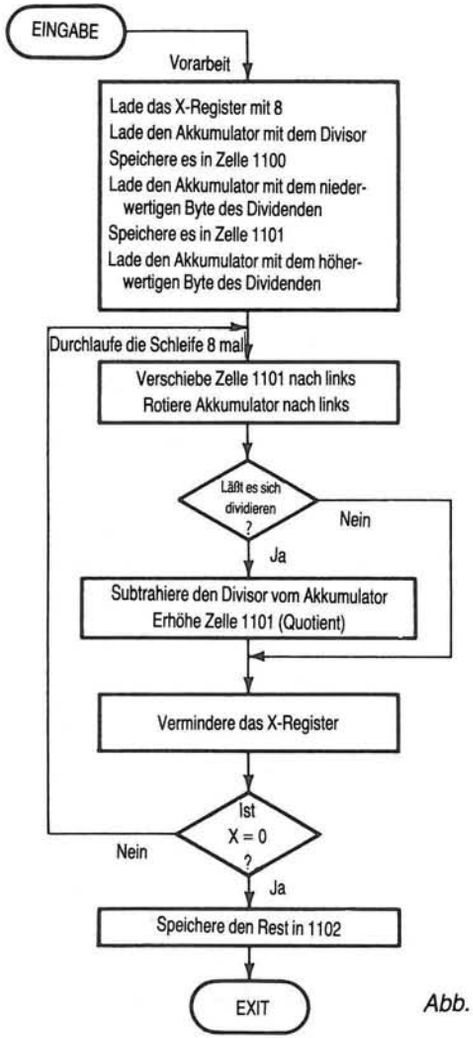


Abb. 11-6 Flußdiagramm der Division

Die Division wird durch den Vergleich des Divisors mit dem Akkumulator bewerkstelligt. Jedesmal, wenn der Divisor kleiner oder gleich dem Akkumulator-Inhalt ist, wird der Divisor vom Akkumulator-Inhalt abgezogen und der Inhalt von 1101 um 1 erhöht, d. h. im Quotienten erscheint eine 1.
Ist der Divisor größer als der Akkumulator, so wird nicht mehr subtrahiert und 1101 nicht mehr erhöht, d. h. im Quotienten erscheint eine 0.
Während die Bits im Akkumulator und in 1101 nach links wandern, gelangt von rechts der Quotient in die Zelle 1101 (0, wenn der Divisor „nicht in den Dividenden ging", andernfalls 1).
Nach Abarbeiten der Schleife (8maliger Durchlauf) wird der Wert nach Zelle 1102 gebracht. Nach Programmende finden Sie den 8-Bit Quotienten in 1101 und den Rest in 1102.
Die Vorbereitungsarbeiten gleichen denen des **Multiplikationsprogramms.**

```
10    *=$1000
20    LDX   #8
30    LDA   #$3A
40    STA   $1100
50    LDA   #$16
60    STA   $1101
70    LDA   #4
```

Als nächstes kommt der Hauptteil des Programms. In diesem Programm erscheint der Rotations-Befehl in der Schleife in anderer Form. Zur Rotation des Akkumulators verwenden wir ROL A. Der Befehl arbeitet genauso, wie bei der Rotation eines Speicherzellen-Inhalts. Diesmal jedoch wird der Akkumulator-Inhalt rotiert. Wieder wird die Schleife 8mal durchlaufen, wobei das Register X als Zähler dient.

```
80    LOOP   ASL $1101
90           ROL  A
100          CMP  $1100
110          BCC  BRANCH
120          SBC  $1100
130          INC  $1101
140   BRANCH DEX
150          BNE  LOOP
```

Ist die Schleife 8mal durchlaufen, wird der Akkumulator-Inhalt (der Rest) zwecks Vereinfachung der Ausgabe nach 1102 gebracht.

```
160    STA   $1102
170    END
```

Geben Sie nun das Programm ein und assemblieren Sie es. Das assemblierte Programm sollte folgendermaßen aussehen:

```
0000                10          *=    $1000
1000  A208        20          LDX   #8
1002  A93A        30          LDA   #$3A
1004  8D0011      40          STA   $1100
1007  A916        50          LDA   #$16
1009  8D0111      60          STA   $1101
100C  A904        70          LDA   #4
100E  0E0111      80 LOOP     ASL   $1101
1011  2A          90          ROL   A
1012  CD0011      0100        CMP   $1100
1015  9006        0110        BCC   BRANCH
1017  ED0011      0120        SBC   $1100
101A  EE0111      0130        INC   $1101
101D  CA          0140 BRANCH DEX
101E  D0EE        0150        BNE   LOOP
1020  8D0211      0160        STA   $1102
                  0170 END
```

Abb. 11-7 Ausdruck des assemblierten Divisions-Programms

Geben Sie den Debugger-Modus ein, lassen Sie das Programm laufen und geben Sie die Ergebnisse aus.

```
EDIT
BUG

DEBUG
G1000
1023        A=22 X=00 Y=00 P=32 S=00
DEBUG
D1101,1102
                    <-- Quotient
1101 12 02
                    <-- Rest
DEBUG
■
```

Dies stimmt mit unserer schriftlichen Rechnung überein. Versuchen Sie, die zusätzlichen Divisionsaufgaben aus Abb. 11–8 zu lösen. Die Lösungen finden Sie am Ende der Übungen zu diesem Kapitel.

Dividend		*Divisor*	*Ergebnis*	
MSB (100D)	*LSB (1008)*	*(1003)*	*Quotient (1101)*	*Rest (1102)*
01	00	10		
0A	BC	DE		
0A	05	9F		
05	AA	83		
7F	FF	FF		

Abb. 11-8 Divisionsübungen

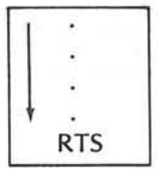
Wenn Sie irgendwelche andere Beispiele ausprobieren wollen, achten Sie darauf, daß die Quotienten nicht größer als FF werden. Das Programm versagt bei Quotienten, die ein Byte überschreiten.

UNTERPROGRAMME

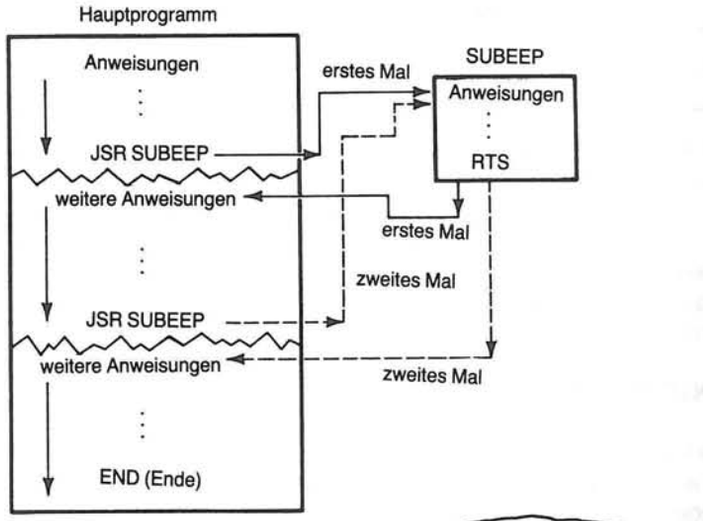
Es kommt nicht selten vor, daß eine bestimmte Befehlsfolge an verschiedenen Stellen eines Programms ausgeführt werden soll. Anstatt an jeder Stelle, an der diese Befehlsfolge verwendet wird, alle Befehle neuerlich aufzuschreiben, ist es effizienter, sich diese Arbeit einmal zu machen und dies als Unterprogramm zum Hauptprogramm zu verwenden. Jedesmal, wenn die Befehlsfolge ausgeführt werden soll, wird einfach die Jump to SubRoutine Anweisung (Springe ins Unterprogramm) gegeben.

Beispiel:
Liegt ein Unterprogramm mit der Marke SUBEEP vor, so hat die JSR-Anweisung die Form:
JSR SUBEEP
Haben Sie das Unterprogramm ausgeführt, müssen Sie an die entsprechende Stelle des Hauptprogramms zurückkehren können, und zwar genau zu der Anweisung, die auf den JSR-Befehl, mit dem Sie in das Unterprogramm gesprungen sind, folgt. Dies wird dadurch erreicht, daß am Ende des Unterprogramms eine RTS-Anweisung (Re-Turn from Subroutine) gegeben wird.

Unterprogramm-Anweisungen



Man sagt auch häufig, das Unterprogramm wird durch die ISR SUBEEP Anweisung „aufgerufen". Das Unterprogramm wird ausgeführt und die Rückkehr zum Hauptprogramm geschieht folgendermaßen:



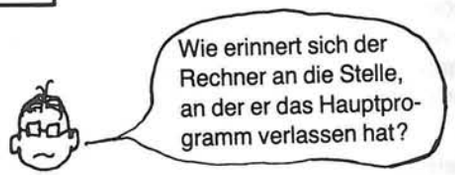
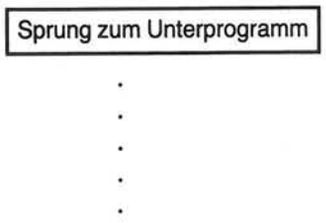
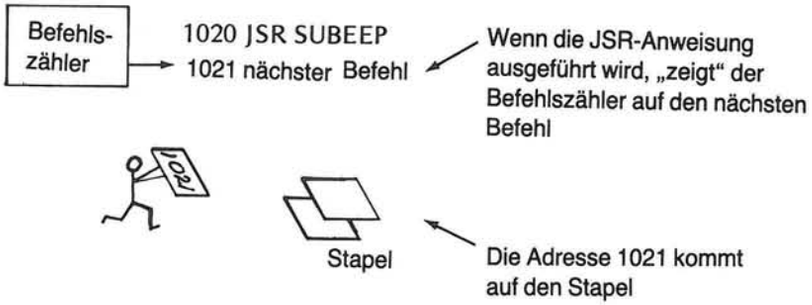


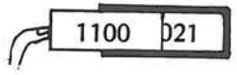
Abb. 11-9 Flußdiagramm des Unterprogramms

Erinnern Sie sich noch an den in Kapitel 1 und 5 behandelten Stapelspeicher? Wenn eine JSR-Anweisung ausgeführt wird, bringt der Rechner automatisch die im Befehlszähler enthaltene Speicheradresse auf den Stapel. Wird dann am Ende des Unterprogramms die RTS-Anweisung ausgeführt, holt sich der Rechner die Adresse vom Stapel und bringt sie wieder in den Befehlszähler. Von dieser Stelle an läuft dann das Hauptprogramm weiter.

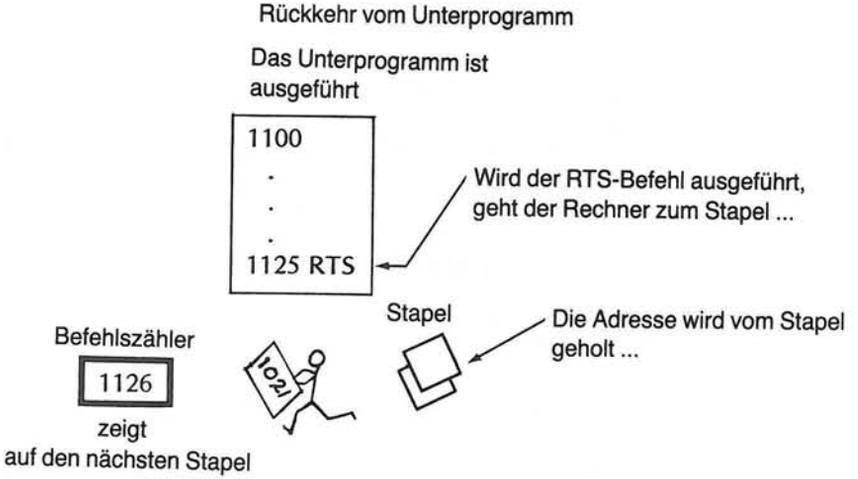




Die Adresse des Unterprogramms kommt in den Befehlszähler.

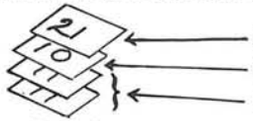


Der Befehlszähler zwingt nun den Rechner, die bei 1100 beginnende Befehlsfolge (Unterprogramm) auszuführen.



Dies veranlaßt den Rechner zur Ausführung der bei 1021 beginnenden Befehlsfolge.

Sicher erinnern Sie sich inzwischen wieder, daß auch der Stapelspeicher aus 8-Bit Zellen besteht. Adressen jedoch können bis zu 16 Bits lang sein. Daher ist der Vorgang etwas komplizierter, als aus unserer eben gegebenen, vereinfachten Beschreibung hervorgeht. Tatsächlich wird die Adresse in Form von zwei Bytes gestapelt.



Bei Ausführung der RTS-Anweisung wird die Rücksprung-Adresse ebenfalls in zwei getrennten Bytes vom Stapel geholt. Anders gesagt, macht die RTS-Anweisung den Effekt der JSR-Anweisung rückgängig. Beide arbeiten genauso zusammen, wie in BASIC die GOSUB- und RETURN-Anweisungen.
Ebenso wie in BASIC können Unterprogramme auch in Assembler ineinandergeschachtelt sein.

ANWENDUNG EINES UNTERPROGRAMMS

Kehren wir zu dem zuvor in diesem Kapitel benutzten 8-Bit-Multiplikations-Programm zurück und fügen ein Ton-Unterprogramm ein, das jedesmal, wenn ein Bit des Multiplikators verwendet wird, einen anderen Ton produziert. Dazu müssen Sie eine ganze Anzahl neuer Anweisungen in das ursprüngliche Programm einbauen. Sie werden einige Anweisungen, die den Stapelspeicher zum Ablegen und Wiederholen von Informationen benutzen, kennenlernen.

HAUPTPROGRAMM (unveränderter Teil)

```
100   *=$100
110   LDX   #8
120   LDA   #$12
130   STA   $1102
140   LDA   #$3A
150   STA   $1103
160   LDA   #0
170   STA   $1100
180   STA   $1101
190   LOOP  ASL A
200   ROL   $1100
210   ASL   $1103
220   BCC   SKIP
230   CLC
240   ADC   $1102
250   BCC   SKIP
260   INC   $1100
```

Datenaufbereitung

Multiplikationsschleife

HAUPTPROGRAMM (geänderter Teil)

```
270   SKIP PHA
280    PHP
290    TXA
300    PHA
310    JSR SUBEEP  <-- Hier gehen Sie ins Unterprogramm (GOSUB)
320    PLA         <-- Hier kommen Sie zurück
330    TAX
340    PLP
350    PLA
360    DEX
370    BNE LOOP
380    STA $1101
390    BRK         <-- Programm endet hier
```

Unterprogramm SUBEEP

```
1024
400   SUBEEP LDA #$C8
410    STA $CE
420    LDA #A0
430    STA $D201
440    LDA $1103,X
450    STA $D200
460    LDA #$AF
470   ALOOP STA $D201
480    LDA $CE
1051
490    JSR DELAY   <-- Springe zum geschachtelten Unterprogramm
500    SEC         <-- Kehre hierhin zurück
510    SBC #$01
520    CMP #$9F
530    BNE ALOOP
540    RTS         <-- Ende von SUBEEP
1059
550   DELAY LDY #$13  <-- Anfang des geschachtelten Unterprogramms
560   DELAY2 DEY
Geschachteltes Unterprogramm
570    BNE DELAY2
580    DEX
590    BNE DELAY
600    RTS         <-- Ende des geschachtelten Unterprogramms
1061
610   END
```

Das Programm wird in der in Abb. 11-10 gezeigten Reihenfolge durchlaufen. Das Unterprogramm SUBEEP wird vom Hauptprogramm achtmal aufgerufen. Das mit der Marke DELAY gekennzeichnete Unterprogramm ist in SUBEEP geschachtelt.

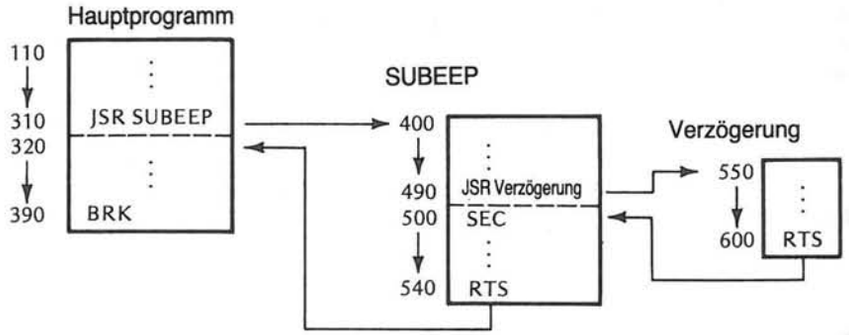
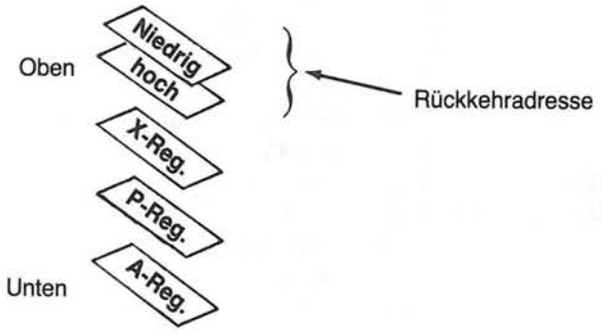


Abb. 11-10 Flußdiagramm des Ton-Unterprogramms

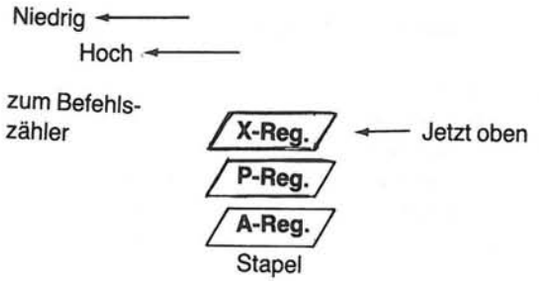
Die Tonfrequenz wird durch den in D200 von der Datentabelle aus 1103+X gespeicherten Wert gesteuert. Das Akustik-Steuer-Register in D201 steuert die Lautstärke des Tons.
Da der 6502 Mikroprozessor nur zwei Indexregister (X und Y) besitzt, wird in den Zeilen 270-300 und 320-350 eine neue Programmiertechnik vorgeführt. Das Hauptprogramm *und* die Unterprogramme nutzen das Register X und den Akkumulator zu verschiedenen Zwecken. Es werden daher *vor* dem Aufruf der Unterprogramme alle Werte in den Registern X, A und P gestapelt, damit diese Register den Unterprogrammen für eigene Vorhaben zur Verfügung stehen.

PHA Bringe Akkumulator-Inhalt auf Stapel (Push)
PHP Bringe P-Inhalt auf Stapel
TXA und PHA Bringe X nach Z, dann Akkumulator-Inhalt auf Stapel

Bei Ausführung von JSR gelangt die Information folgendermaßen auf den Stapel.



Wird die zu SUBEEP gehörende RTS-Anweisung ausgeführt, holt sich der Rechner die Rückkehr-Adresse vom Stapel, auf dem sich oben dann noch folgende Daten befinden.



Der Rechner kehrt in das Hauptprogramm zurück und erledigt nacheinander:

PLA, TAX ← Stapelkopf wegnehmen und in das Register X bringen

PLP ← Stapelkopf wegnehmen und in das Register P bringen

PLA ← Stapelkopf wegnehmen und in das Register A bringen

Natürlich werden die Werte, wie Sie sehen, in umgekehrter Reihenfolge vom Stapel heruntergeholt, wie Sie vorher dort abgelegt wurden. Damit kommt wieder alles dahin, wo es hingehört.
Geben Sie das Programm, wie es dasteht, ein und assemblieren Sie es. Gehen Sie dann zu DEBUG und geben Sie die Daten aus der Tabelle von 1104 bis 110B ein.

Adresse	*Daten*
1104	48
1105	60
1106	78
1107	90
1108	A8
1109	C0
110A	D8
110B	F0

Abb. 11-11 Daten für das Ton-Programm

Lassen Sie nun das Programm laufen und schauen Sie, ob Ihnen die vom Unterprogramm SUBEEP erzeugten Noten zusagen. Wenn Sie sie ändern wollen, verändern Sie die Daten in Abb. 11-11. Die Frequenzen werden in umgekehrter Reihenfolge von 11OB bis 1104 entnommen.

ZUSAMMENFASSUNG

Sie haben nun gesehen, wie zwei Operationen, für die der Rechner keine speziellen Befehle kennt, durch Bildung von zusammengesetzten Anweisungen, die aus Gruppen einzelner Befehle bestehen, durchgeführt werden können. Obwohl der Mikroprozessor 6502 keine Multiplikations- oder Divisions-Befehle kennt, können diese Operationen dennoch mittels einer Reihe von Additionen, Subtraktionen, Verschiebungen oder Rotationen durchgeführt werden. In diesem Kapitel haben Sie gelernt:

- Verschiebungs- und Rotations-Anweisungen in Programmen anzuwenden, um Zahlen zu multiplizieren und zu dividieren;
- ein Demonstrations-Programm anzuwenden, das zwei 8-Bit Zahlen miteinander multipliziert und dabei eine 16-Bit Zahl als Ergebnis erzeugt;
- ein Demonstrations-Programm anzuwenden, das einen 16-Bit Dividenden durch einen 8-Bit Divisor teilt und dabei ein 8-Bit Ergebnis und einen 8-Bit Rest erzeugt;
- mittels einer JSR-Anweisung (Jump to SubRoutine – Springe ins Unterprogramm) ein Unterprogramm aufzurufen und mittels der RTS-Anweisung (ReTurn from Subroutine – Kehre vom Unterprogramm zurück) in das Hauptprogramm zurückzukehren;
- den Stapelspeicher zum Aufheben und Zurückholen von Informationen zu verwenden, falls Register oder Zähler verschiedene Aufgaben wahrnehmen sollen;
- die folgenden Befehle zu verwenden:
  - PHA Bringe den Akkumulator-Inhalt auf den Stapel
  - PHP Bringe den Inhalt des P-Registers auf den Stapel
  - PLA Hole den obersten Wert vom Stapel und bringe ihn in den Akkumulator
  - PLP Hole den obersten Wert vom Stapel und bringe ihn in das P-Register
- Ein Töne erzeugendes Unterprogramm zu verwenden

ÜBUNGEN

1. Multiplizieren Sie die Dezimalzahlen 28 und 37.
   (a) in dezimaler Notation

   ```
   28
   x37
   ```

   (b) in binärer Notation

   ```
   28=
   x37= _____
   ```

   (c) Prüfen Sie das binäre Ergebnis

   ```
   _____
   + _____
   + _____
   ```


2. Welches Bit des Multiplikators wird bzgl. des Ablaufdiagramms in Abb. 11–2 zuerst verarbeitet? _____
   höchster, niedrigster Stellenwert

   Die Aufgaben 3 bis 7 beziehen sich auf die im folgenden dargestellten Situationen im Akkumulator und im Carry, jeweils vor und nach der Ausführung einer Verschiebungs- oder Rotations-Anweisung.

   (a)

Carry	Akkumulator	
1	01010111	vorher
0	10101110	nachher

   (b)

Carry	Akkumulator	
1	01010111	vorher
0	10101111	nachher

   (c)

Carry	Akkumulator	
1	01010111	vorher
1	00101011	nachher

   (d)

Carry	Akkumulator	
1	01010111	vorher
1	10101011	nachher

3. Nennen Sie die Verschiebungs- bzw. Rotations-Anweisung, die in (a) ausgeführt wurde _____
4. Nennen Sie die Verschiebungs- bzw. Rotations-Anweisung, die in (b) ausgeführt wurde _____
5. Nennen Sie die Verschiebungs- bzw. Rotations-Anweisung, die in (c) ausgeführt wurde _____
6. Nennen Sie die Verschiebungs- bzw. Rotations-Anweisung, die in (d) ausgeführt wurde _____
7. Tragen Sie unten die Ergebnisse ein, die aus den nacheinander ausgeführten Operationen, die Ihren Antworten aus den Aufgaben 3, 4, 5 und 6 entsprechen, resultieren würden.

   Ausgangssituation:

Carry	Akkumulator
1	01010111

   Die Anweisungen aus den Antworten auf 3, dann 4, dann 5 und 6 werden ausgeführt.

   Endsituation:

Carry	Akkumulator

8. Das 8-Bit Multiplikations-Programm aus Abb. 11–3 soll zur Multiplikation der Dezimalzahlen 120 und 73 verwendet werden.
   (a) Welche zwei hexadezimalen Werte müssen in welche beiden Speicherplätze gebracht werden?

Adresse	*Daten*

(b) Zeigen Sie, wie man mit dem DEBUGGER beide Werte mit einem Befehl lädt.

```
.
.
DEBUG

_____
```

9. Der Assembler-Befehl zum Aufruf eines Unterprogramms mit der Marke SUBBY lautet:

   _____ _____

10. Der letzte in einem Unterprogramm ausgeführte Befehl muß lauten:

    _____

ANTWORTEN

Antworten zu Abb. 11-4

Multiplikator	*Multiplikand*	*Ergebnis*
03	09	00 1B
1A	E4	17 28
3C	D8	32 A0
A4	C2	7C 48
FF	FF	FE 01

Antworten zu Abb. 11-8

Dividend	*Divisor*	*Ergebnis*	
		Quotient	*Rest*
01 00	10	10	00
0A BC	DE	0C	54
0A 05	9F	10	15
05 AA	83	0B	09
7F FF	FF	80	7F

1. (a) In dezimaler Notation

```
  28
 x37
 196
 84
1036
```


(b) In binärer Notation

```
 18=        00011100
x37=        00100101
            --------
            00011100
           00000000
          00011100
         00000000
        00000000
       00011100
       --------
       0010000001100
```

(c)

```
   4
  +8

 +1024
 ----
 1036
```

2. Höherwertiges Bit
3. ASL A
4. ROL A
5. LSR A
6. ROR A
7. Endsituation

Carry	Akkumulator
0	00010111

8. (a)

Adresse	*Daten*
1003	49
1008	78

die Daten können auch vertauscht sein

(b)

```
.
.
DEBUG
C1003<49 , , , , , 78
```

9. JSR SUBBY
10. RTS

Programmierpraxis

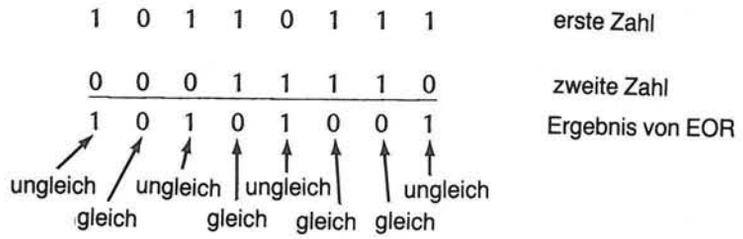
Man kann eine Menge über das Programmieren lernen, wenn man von anderen geschriebene Programme anwendet, analysiert und abändert. Inzwischen sind Sie mit dem Assembler Modul vertraut und sollten sich bei seinem Gebrauch richtig wohlfühlen. Wir haben daher dies letzte Kapitel Programmen gewidmet, die Sie benutzen, aber auch für Ihre eigenen Zwecke umbauen können. Wir machen einige Änderungsvorschläge, und zweifellos werden Ihnen selbst solche einfallen.
Wir haben den Assembler Modul nicht vollständig behandelt, sondern uns auf die Teile beschränkt, die Sie nach unserer Meinung am häufigsten brauchen werden.
Programmieren erfordert viel Übung. Sie sollten daher auch andere Programme ausprobieren, um alle Aspekte der 6502-Befehlsliste und der Assembler-Modul-Befehle bzw. -Anweisungen zu untersuchen. Nehmen Sie sich die Befehle, die Sie noch nicht kennen, vor, studieren Sie die Beschreibungen und schreiben Sie kurze Programme, um sie anzuwenden.
Sie können den Assembler auch verwenden, um Unterprogramme in Maschinensprache zu entwickeln, die die Leistungsfähigkeit und Schnelligkeit Ihrer BASIC-Programme erhöhen. Mannigfache Anwendungen, die Klang-, Graphik- und Trickfilm-Effekte sowie spezielle Funktionen verwenden, lassen sich im Gegensatz zu BASIC in Maschinensprache programmieren.
Bereits früher haben wir – mit erhobenem Zeigefinger – darauf hingewiesen, daß Ihre Maschinenprogramme eine eigene Speicherregion benötigen. Wenn Sie also assemblierte Unterprogramme von BASIC aufrufen, müssen Sie sicher sein, daß diese Maschinenprogramme in einer von Ihrem BASIC-Programm nicht verwendeten Speicherregion residieren. Die Speicherplätze von 0600 (dezimal 1536) bis 06FF (dezimal 1791) können für diesen Zweck eingesetzt werden. Diese 256 zur Verfügung stehenden Bytes werden für die meisten Ihrer Maschinen-Unterprogramme ausreichen. Alle Programme dieses Kapitels benutzen diesen Teil des Speichers. Der BASIC Modul reserviert überdies 7 Bytes des Zero Page Speichers (die Speicherplätze CB bis D1 oder dezimal von 203 bis 209). Zusätzlich werden die Zellen D4 und D5 des Zero Page Speichers zur Übergabe von Parametern (numerischen Werten) vom Maschinen-Unterprogramm mit der USR-Funktion an BASIC verwendet.
Das Assembler Benutzer Manual enthält einige Unterhaltungsprogramme. Sie finden dort auch eine Methode zur Erzeugung eines assemblierten Unterprogramms und ein BASIC-Programm, das dieses Unterprogramm aufruft. Wir werden uns in diesem Kapi-

tel mit ähnlichen Programmen, die allerdings nur den Assembler Modul benutzen, beschäftigen. Wenn Sie wollen, können Sie die Programme so neu schreiben, daß sie von BASIC aus aufgerufen werden. Unser erstes Programm zeigt beide Methoden der Verwendung eines assemblierten Programms.
In BASIC können logische Funktionen mit Hilfe von NOT, AND und OR formuliert werden. Die Befehlsliste des 6502 enthält zusätzlich das Exclusive OR (ausschließliches oder) als logische Funktion. Unser erstes Programm zeigt seine Verwendung.

ANWENDUNG EINER LOGISCHEN FUNKTION

Der Mikroprozessor 6502 besitzt Befehle, welche die in BASIC vorhandenen logischen Funktionen NOT, AND und OR erweitern. Einer dieser Befehle berechnet die logische Funktion des Exclusive OR (Assembler-Abkürzung EOR). Der Befehl vergleicht zwei Binärzahlen Bit-weise. Sind zwei Bits gleich, steht in der entsprechenden Bit-Position des Ergebnisses eine Null. Ist eines der beiden Bits, die miteinander verglichen werden, 1 (Eins) und das andere 0 (Null), dann steht in der entsprechenden Bit-Position des Ergebnisses eine 1 (Eins).

Beispiel:



Sie sehen, daß in einer gegebenen Bit-Position des Ergebnisses eine 1 steht, wenn *entweder* in der einen *oder* in der anderen Zahl an der entsprechenden Stelle eine Eins vorkommt, *nicht* jedoch dann, wenn *beide* Zahlen dort eine Eins haben.
Die Anwendung des EOR-Befehls auf die beiden hexadezimalen Zahlen B7 und 1E liefert, wie das obige Beispiel zeigt, das Ergebnis A9.
Wir wollen nun ein Assembler-Programm schreiben, das unser Beispiel auf dem Rechner durchführt.

EOR QUELL-PROGRAMM

```
100     *=$0600          Programm-Anfang bei 0600
110     LDA   #$B7
120     EOR   #$1E       EOR B7 und E1
130     STA   $1000
140     END
```

Geben Sie das Programm ein und assemblieren Sie es.

ASSEMBLIERTE AUSGABE

```
.
.
130   STA    $1000
140   END
ASM
0000                  100       *=     $0600

0600  A9B7             110       LDA    #$B7

0602  491E             120       EOR    #$1E

0604  8D0010           130       STA    $1000

                      140  END

EDIT
■
```

Geben Sie den DEBUGGER ein und führen Sie das Programm aus.

```
.
.
EDIT
BUG

DEBUG
G0600
0607        A=A9  X=00  Y=00  P=B0  S=00
DEBUG
■
```


```
.
.
DEBUG
D1000,1000

1000  A9
DEBUG
■
```


Es klappt, sehr schnell und kurz. Lassen Sie uns sehen, wie wir ein ähnliches Programm von BASIC aus erreichen können. Die Werte, auf die das Exclusive OR angewendet werden soll, werden vom BASIC-Unterprogramm an das Maschinen-Unterprogramm übergeben. Das Unterprogramm holt sich die Werte vom Stapel, unterzieht sie dem Vergleich und liefert das Ergebnis an das BASIC-Programm ab. Abb. 12–1 zeigt das Ablaufdiagramm für das BASIC-Programm und das Unterprogramm.

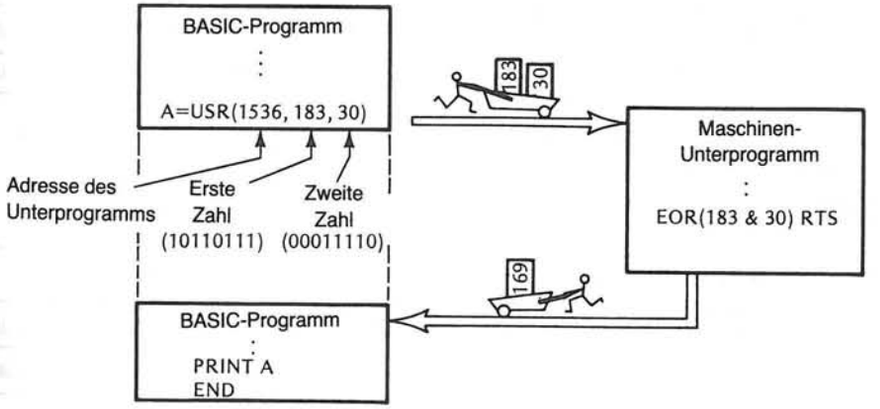
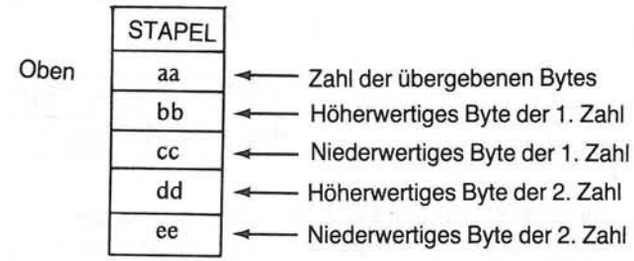


Abb. 12-1 Flußdiagramm zum logischen Programmablauf

Das BASIC-Programm ist sehr kurz. Geben Sie es noch nicht in den Rechner, denken Sie vielmehr an den früher erwähnten Vorgang.

```
100   REM ** EOR UNTERPROGRAMM **
110   GR. 0
120   A = USR (1536, 183, 30)
130   PRINT "183 EXCLUSIVES ODER MIT 30 IST": A
140   END
```

Beachten Sie die USR-Funktion in Zeile 120. Die in Klammern hinter der Adresse 1536 stehenden Daten werden für die spätere Verwendung durch das Unterprogramm wie folgt gestapelt:



wobei aa, bb, cc, dd und ee
hexadezimale Darstellungen sind.

Das Maschinen-Unterprogramm nimmt das oberste Byte vom Stapel. Es wird nicht gebraucht. Danach wird das höherwertige Byte der ersten Zahl vom Stapel geholt und in Zelle $CC gebracht, dann das niederwertige in Zelle $CD. Sie haben damit folgende Situation vor sich:

STAPEL	
dd	← höherwertiges Byte – 2. Zahl
ee	← niederwertiges Byte – 2. Zahl

Oben

SPEICHER	
00CC	höherwertiges Byte – 1. Zahl
00CD	niederwertiges Byte – 1. Zahl

Nun holt das Unterprogramm das höherwertige Byte der zweiten Zahl vom Stapel und vergleicht es (Exlusive OR) mit dem höherwertigen Byte der ersten Zahl in $CC. Das Ergebnis wird in Zelle $D5 gespeichert. Man hat jetzt:

STAPEL
ee

Oben — niederwertiges Byte – 2. Zahl

SPEICHER	
00CC	höherwertiges Byte – 1. Zahl
00CD	niederwertiges Byte – 1. Zahl
.	.
.	.
.	.
00D5	höherwertiges Byte – Ergebnis

Als nächstes holt das Unterprogramm das niederwertige Byte vom Stapel und vergleicht es (Exclusive OR) mit dem niederwertigen Byte aus $CD. Das Ergebnis wird in $D4 gespeichert, und man hat jetzt:

STAPEL
??

Oben — Alle Variablen, die von USR übergeben wurden, sind fort

SPEICHER	
00CC	höherwertiges Byte – 1. Zahl
00CD	niederwertiges Byte – 1. Zahl
.	.
.	.
.	.
00D4	niederwertiges Byte-Ergebnis
00D5	höherwertiges Byte-Ergebnis

Als letztes wird dann die RTS-Anweisung (Return from subroutine) ausgeführt. Die in $D5 und $D4 gespeicherten Werte werden als Dezimalwert für die Variable A an das BASIC-Programm übergeben.
Und so sieht das Unterprogramm in Maschinensprache aus:


```
100     *=$0600
110     EXOR PLA      <-- Zahl der Datenbytes
120      PLA          <-- hole höherwertiges Byte – erste Zahl
130      STA $CC      <-- und speichere
140      PLA          <-- nun das niederwertige Byte
150      STA $CD
160      PLA          <-- höherwertiges Byte – zweite Zahl
170      EOR $CC      <-- exclusiv oder
180      STA $D5      <-- und speichere
190      PLA          <-- nun die niederwertigen Bytes
200      EOR $CD
210      STA $D4
220      RTS          <-- kehre jetzt zurück
230     END
```

EINGEBEN DES UNTERPROGRAMMS

Führen Sie nun die im Assembler-Benutzer-Manual beschriebenen Schritte aus:

1. Bringen Sie den Assembler Modul in den Rechner und geben Sie das Assembler-Unterprogramm ein.
2. Assemblieren Sie es, um sicher zu gehen, daß Sie keinen Fehler gemacht haben.

```
EDIT
ASM

0000            0100          *=    $0600
0600  68        0110    EXOR  PLA
0601  68        0120          PLA
0602  85CC      0130          STA   $CC
0604  68        0140          PLA
0605  85CD      0150          STA   $CD
0607  68        0160          PLA
0608  45CC      0170          EOR   $CC
060A  85D5      0180          STA   $D5
060C  68        0190          PLA
060D  45CD      0200          EOR   $CD
060F  85D4      0210          STA   $D4
0611  60        0220          RTS
                0230    END

EDIT
■
```

3. Verwenden Sie 3(a) zum Speichern auf Cassette oder 3(b) zum Speichern auf Platte und assemblieren Sie erneut.
   (a) Für Kassette:

```
.
.
EDIT
ASM,,#C:
```

Drücke RETURN
Nach dem „BEEP" drücke nochmals RETURN

   (b) Für Platte:

```
ASM,,#D:EXOR.OBJ
```

Drücke RETURN

4. Nehmen Sie den Assembler Modul heraus und legen Sie stattdessen den BASIC Modul ein.
5. Warten Sie auf die BASIC READY Nachricht.

```
.
READY
■
```

6. Verwenden Sie 6 (a) für Cassette oder 6 (b) für Platte
   (a) Laden Sie wie üblich vom Band mittels:

```
CLOAD
```

RETURN geben

   (b) Tippen Sie DOS zur Aktivierung des Platten-Betriebssystems.

```
DISK OPERATING SYSTEM                    9/24/79
COPYRIGHT 1979 ATARI

A. DISK DIRECTORY          I. FORMAT DISK
B. RUN CARTRIDGE           J. DUPLICATE DISK
C. COPY FILE               K. BINARY SAVE
D. DELETE FILE(S)          L. BINARY LOAD
E. RENAME FILE             M. RUN AT ADDRESS
F. LOCK FILE               N. DEFINE DEVICE
G. UNLOCK FILE             O. DUPLICATE FILE
H. WRITE DOS FILE

SELECT ITEM
■
```


Tippen Sie dann L zum Laden einer Binär-Datei.

```
.
.
.
SELECT ITEM
L
LOAD FROM WHAT FILE?
■
```

Tippen Sie nun: EXOR.OBJ und drücken Sie die RETURN-Taste.

```
.
.
SELECT ITEM
L
LOAD FROM WHAT FILE?
EXOR.OBJ
```

Ist die Datei geladen, gibt der Rechner den Hinweis SELECT ITEM PROMPT.

```
.
.
.
SELECT ITEM
■
```

Geben Sie B und RETURN, um zu BASIC zurückzukehren.

```
.
.
.
.
SELECT ITEM
B

READY
■
```

7. Sie müßten nun wieder im BASIC READY Modus sein.

8. Geben Sie das BASIC-Programm ein.

```
100   REM ** EOR UNTERPROGRAMM **
110   CLS
120   A = USR (1536, 183, 30)
130   PRINT "183 EXCLUSIVES ODER MIT 30 IST": A
140   END
■
```

Sie sind nun soweit, daß Sie das Programm laufen lassen können – *fast*. Überprüfen Sie vorher, ob das Unterprogramm auch im Speicher ist, indem Sie die erste Zelle nachsehen. Tippen Sie:

PRINT PEEK (1536) und drücken Sie RETURN

Der Rechner müßte mit 104 (d. h. der Dezimaldarstellung des ersten Maschinenbefehls PLA) antworten.
Antwortet Ihnen der Rechner richtig, dann lassen Sie das Programm mit dem RUN-Befehl laufen. Tut er es nicht, müssen Sie noch einmal ganz von vorne anfangen, und diesmal *sehr, sehr* sorgfältig!

```
RUN
169
```

EOR-Ergebnis in dezimaler Form

128+32+8+1 oder

10101001 binär

Sie können das Exclusive OR auf andere Werte anwenden, indem Sie einfach Zeile 120 des BASIC-Programms ändern.

```
120 A = USR(1536,183,30)
```

Diese Werte können geändert werden.

Ändern Sie *auf keinen Fall* den Wert 1536, da dies die Start-Adresse des Unterprogramms ist. Die beiden anderen Werte (183 und 30) können durch beliebige ganze Zahlen von 0 bis 65535 ersetzt werden.
Die restlichen Programme des Kapitels können Sie alle direkt über den Debugger-Modus des Assemblers laufen lassen. Wollen Sie es mit der Version des Zugangs über BASIC, wie im EOR-Programm, versuchen, dann tun Sie Ihren Gefühlen keinen Zwang an.

EIN PROGRAMM ZUR KLANGERZEUGUNG

Dies Programm verwendet drei Klang-Register (oder Kanäle), um Akkorde zu spielen. Die Daten werden in die entsprechenden Register gebracht, und der erzeugte Ton wird an Ihr Fernsehgerät weitergegeben. Stellen Sie Ihr Gerät vor dem Programmlauf auf eine ausreichende Lautstärke. Das Programm steuert zwar die erzeugte Lautstärke, der Fernseher muß aber genügend „aufgedreht" sein, damit man den Ton hören kann.
Der zuletzt erzeugte Akkord erklingt so lange, bis eine andere Note in die Klang-Register gebracht wird. Daher setzt der letzte Teil des Programms die Lautstärke auf Null. Es gibt freilich andere Möglichkeiten, dies zu erreichen, aber wozu, wenn es der Rechner für Sie tun kann.
Wir haben das Quell-Programm kommentiert und damit jeden Teil des Programms und die verwendeten Werte gekennzeichnet. Dies ist eine nützliche Angewohnheit beim Programmieren. Wenn Sie nämlich Ihr Programm auf Platte oder Band bringen, werden Ihnen die Kommentare bei späterer Wiederverwendung hilfreich sein. Abb. 12–2 zeigt den Programm-Ablauf.

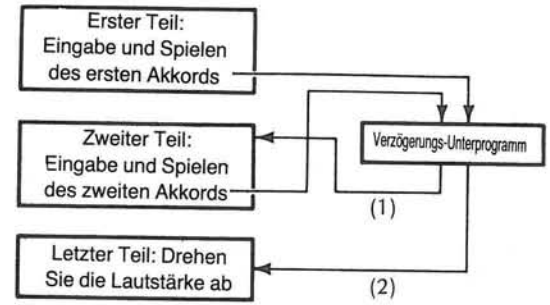


Abb. 12-2 Ablauf des Programms zur Klangerzeugung

QUELL-PROGRAMM ZUR KLANGERZEUGUNG

```
100   *=$0600
110   ;ABSCHNITT FUER DEN ERSTEN AKKORD
120   LDA  #$79          ERSTE NOTE
130   STA  $D200
140   LDA  #$88          ZWEITE NOTE
150   STA  $D202
160   LDA  #$99          DRITTE NOTE
170   STA  $D204
180   LDA  #$AF          TON UND LAUTSTAERKE
190   STA  $D201
```

```
200   STA  $D203
210   STA  $D205
220   JSR  DELAY          SPIELE EINE WEILE
230   ;ABSCHNITT FUER DEN ZWEITEN AKKORD
240   LDA  #$C1           ERSTE NOTE
250   STA  $D200
260   LDA  #$D9           ZWEITE NOTE
270   STA  $D202
280   LDA  #$F3           DRITTE NOTE
290   STA  $D204
300   LDA  #$A8           TON UND LAUTSTAERKE
310   STA  $D201
320   STA  $D203
330   STA  $D205
340   JSR  DELAY          SPIELE EINE WEILE
350   ;ABSCHNITT FUER DAS ABDREHEN DER LAUTSTAERKE
360   LDA  #$A0
370   STA  $D201
380   STA  $D203
390   STA  $D205
400   BRK                 ENDE DES HAUPTPROGRAMMS
410 DELAY LDX #$FF
420 LOOP  LDY  #$FF
430 LOOP1 DEY
440   BNE  LOOP1
450   DEX
460   BNE  LOOP
470   RTS
480 END                   ENDE DES QUELL-PROGRAMMS
```

Geben Sie das Programm ein, assemblieren Sie es, und lassen Sie es wie üblich laufen. Gefallen Ihnen die Töne? Wenn nicht, dann schauen Sie einmal in die Notentabelle in Abb. 12–3. Sie können die Frequenzen im Programm ändern, solange Sie dabei nur im Bereich von 0 (höchste Note) bis FF hexadezimal (tiefste Note) bleiben.
Um die Ton- und Lautstärken-Steuerung zu verstehen, müssen Sie einen Blick auf die Binärform der Lautstärke werfen.

Beispiel:
195 dezimal = AF hexadezimal
= 1010 1111

A F ← größte Lautstärke

wählt reinen Ton aus (keine Überlagerung)

Tonleiter	*Frequenzwert (hex)*	
C	1D	hoch
B	1F	
A#	21	
A	23	
G#	25	
G	28	
F#	2A	
F	2D	
E	2F	
D#	32	
D	35	
C#	39	
C	3C	
B	40	
A#	44	
A	48	
G#	4C	
G	51	
F#	55	
F	5B	
E	60	
D#	66	
D	6C	
C#	72	
C	79	mittel
B	80	
A#	88	
A	90	
G#	99	
G	A2	
F#	AC	
F	B6	
E	C1	
D#	CC	
D	D9	
C#	E6	
C	F3	tief

Abb. 12-3 Näherungswerte für die Drei-Oktaven-Tonleiter

Die erste Hex-Stelle (im Beispiel A) steuert den Störanteil im Ton. Wir haben einen reinen Ton genommen. Die zweite Hex-Stelle steuert die Lautstärke, die sich zwischen 0 (leisester Ton = aus) und F (lautester Ton) bewegen kann.
Sie sollten es mit verschiedenen Werten versuchen, um andere Klangeffekte zu erzielen. Sie müssen dazu lediglich die Werte für die Frequenz, den Ton und die Lautstärke ändern. Schauen Sie sich dazu die Kommentare im Quell-Programm an.

1. Das Quell-Programm läßt sich ändern.
   (a) Im Edit-Modus ändern sich die Zeilen:
   120, 140, 160, 180 für den ersten Akkord
   240, 260, 280 und 300 für den zweiten Akkord
   (b) Assemblieren Sie das Programm erneut.
   (c) Lassen Sie es über den Debugger-Modus laufen.
2. Das Objekt-Programm läßt sich ändern.
   Verwenden Sie den Debugger-Modus zur Änderung der Zellen:
   0601, 0606, 060B und 0610 für den ersten Akkord
   061E, 0623, 0628 und 062D für den zweiten Akkord.

EIN NOTEN-PROGRAMM

Dieses Programm arbeitet mit einem einzigen Tonkanal, um eine Reihe von Tönen nacheinander zu spielen. Zwischen den Noten ist eine Zeitverzögerung eingebaut, um die Töne genügend lang hören zu können. Achten Sie auch hier auf ausreichende Lautstärke bei Ihrem Fernseher. Die Tabelle in Abb. 12–3 zeigt Ihnen, daß das Programm eine Oktave der Tonleiter spielt.

ASSEMBLER PROGRAMM

```
100   *=$0600
110   ;SPIELE NOTEN
120   LDA    #$AF      <-- Lautstärke
130   STA    $D201
140   LDA    #$F3      <-- Frequenz 1. Note
150   JSR    RPT
160   LDA    #$D9      <-- Frequenz 2. Note
170   JSR    RPT
180   LDA    #$C1      <-- Frequenz 3. Note
190   JSR    RPT
200   LDA    #$B6            etc.
210   JSR    RPT              |
220   LDA    #$A2             |
230   JSR    RPT              |
```

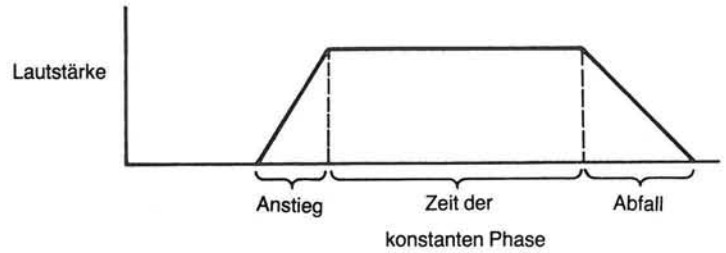


```
240   LDA    #$90
250   JSR    RPT
260   LDA    #$80
270   JSR    RPT
280   LDA    #$79
290   JSR    RPT
300   LDA    #$A0      DREHE DIE LAUTSTAERKE AB
310   STA    $D201
320   BRK              <-- ENDE DES PROGRAMMS
330 RPT STA  $D200
340   LDX    #$ FF
350   JSR    DELAY
360   RTS              <-- Zurück zum Unterprogramm
370 DELAY LDY #$FF
380 LOOP DEY           <-- Spiele die Note eine Weile
390   BNE    LOOP
400   DEX
410   BNE    DELAY
420   RTS              <-- zurück zum RPT-Unterprogramm
430 END
```

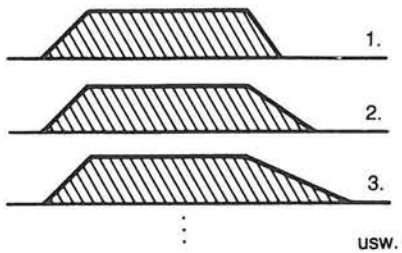
Geben Sie das Programm ein, assemblieren Sie es und lassen Sie es laufen. Klingt es vernünftig? Wenn nicht, können Sie es dort ändern, wo die Frequenzen in den Akkumulator geladen werden. Die Tondauer kann durch die Befehle LDY und LDX im RPT-Unterprogramm geändert werden.
Man könnte das Programm auch so umbauen, daß es sich die Frequenzen aus einer Datentabelle holt. Der Zugriff zu den Frequenzen würde dann durch eine indizierte Lade-Anweisung bewerkstelligt. Das ist allerdings nicht ganz einfach, da die Register X und Y bereits im Unterprogramm zur Zeitverzögerung verwendet werden. Sie müßten die Werte „stapeln" und nach der Verzögerung wieder zurückbringen. Wir haben die entsprechenden Stapelspeicher-Befehle behandelt. Sie wissen also, wie man es machen muß.
In den Zeilen 300 und 310 wurde, wie Sie sehen, der Wert A0 in die Zelle D201 gebracht. Dies setzt die Lautstärke auf Null. Würden diese beiden Zeilen fehlen, so würde die letzte Note trotz Programmende weiterzuhören sein.

EIN PROGRAMM ZUR TONGESTALTUNG

Dies Programm verwendet einen Tonkanal, um den erzeugten Ton zu gestalten. Wenn Sie sich den Graphen der Lautstärke eines erzeugten Tones über eine gewisse Zeitspanne ansehen würden, sähe das etwa so aus:



Die Tonfunktion durchläuft, wie Sie sehen, drei verschiedene Phasen. Sie haben zuvor die Frequenz-, Ton- und Lautstärke-Variablen benutzt. Nun haben Sie die Möglichkeit, die Gestalt des Tones zu ändern.
Wir haben eine der drei Zeitkonstanten ausgewählt – die Abfallzeit. Sie wird während des Programm-Ablaufs geändert. Sie wird innerhalb einer Schleife von 50 bis 200 (dezimal) variiert, und zwar in Schritten von 25 (50, 75, 100, 125, 150, 175 und 200). Der Ton erhält dadurch folgendes Aussehen:



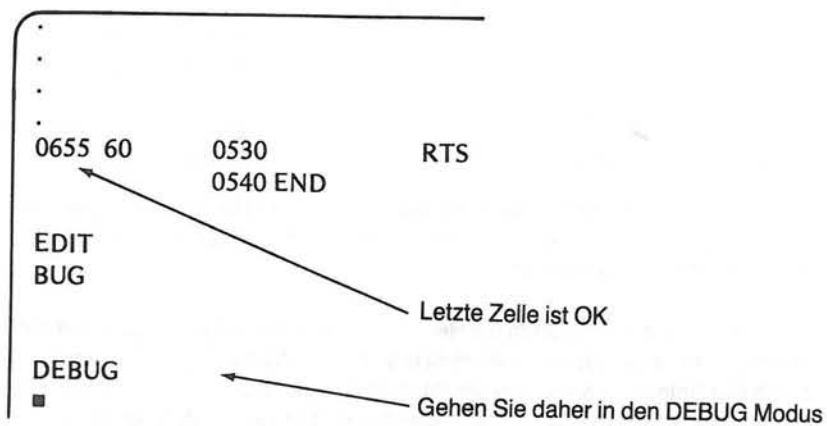
PROGRAMM ZUR TONGESTALTUNG

```
100    *=$0600
110    ;ZEITINITIALISIERUNG
120    LDA   #$3C       FREQUENZ
130    STA   $D200
140    LDA   #$32       KONSTANTE TONHOEHE
150    STA   $CD
160    LDA   #$0A       ANSTIEG
170    STA   $CC
180  LOOP    LDA $1000      ZAEHLER
190    TAX
200    LDA   $1000,X
210    STA   $CE        ABFALL
220    DEC   $1000
230    LDA   #$A0       LAUTSTAERKE
240  ATTK    STA $D201
250    LDX   $CC
260    JSR   DELAY
270    CLC
```



```
280    ADC   #$01
290    CMP   #$80
300    BNE   ATTK
310    LDA   #$0E
320  PEAK    LDX $CD
330    JSR   DELAY
340    SEC
350    SBC   #$01
360    BNE   PEAK
370    LDA   #$AF       TON, LAUTSTAERKE
380  DCAY    STA $D201
390    LDX   $CE
400    JSR   DELAY
410    SEC
420    SBC   #$01
430    CMP   #$9F
440    BNE   DCAY
450    LDA   $1000
460    BNE   LOOP
470    BRK
480  DELAY LDY #$13
490  ROUND DEY
500    BNE   ROUND
510    DEX
520    BNE   DELAY
530    RTS
540  END
```

Dies mag bislang Ihr umfangreichstes Programm sein. Geben Sie es in den Rechner und assemblieren Sie es. Überprüfen Sie die größte verwendete Adresse. Sie erinnern sich: Wir müssen zwischen 0600 und 06FF bleiben, sonst kommen wir in eine Speicherregion, die wir besser meiden.



Ehe Sie das Programm laufen lassen, denken Sie daran, daß die nötigen Daten für Zähler und Abfallzeit fehlen könnten. Geben Sie die Daten aus unten stehender Tabelle im Debugger-Modus ein.

Adresse	*Daten*	
1000	07	← 7 Noten müssen gespielt werden
1001	C8	
1002	AF	
1003	96	← Abfallzeiten
1004	7D	
1005	64	
1006	4B	
1007	32	

Abb. 12-4 Daten für das Tongestaltungs-Programm

Jetzt lassen Sie das Programm laufen. Hören Sie die durch die verschiedenen Abfallzeiten verursachten Tonunterschiede?
Vielleicht möchten Sie es mit anderen Parametern versuchen (Anstiegszeit, Zeit der konstanten Phase, Frequenz oder Lautstärke). Ändern Sie das Programm Ihren eigenen Wünschen entsprechend.

EIN DRUCK-PROGRAMM FÜR DEN BILDSCHIRM

Im Speicher Ihres Atari sind eine ganze Reihe hübscher kleiner Unterprogramme versteckt, die Ihnen eine Menge Zeit und Anstrengung ersparen können. Eines davon entdeckten wir in Zelle $F6A4. Es bringt das Zeichen auf den Bildschirm, dessen ATASCII-Code sich im Akkumulator befindet.

Beispiel:

```
LDA #$41      ← ATASCII Code für den Buchstaben A
JSR $F6A4     ← Zelle des Unterprogramms für die Ausgabe
```

Spring ins Unterprogramm

Diese Anweisungen hätten die Ausgabe von A auf dem Schirm zur Folge, der Cursor würde eine Stelle nach rechts rücken, um die nächste Ausgabe, wie sie auch immer aussehen mag, vorzunehmen.

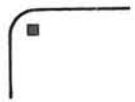
Wenn Sie in Ihrem Programm die Register X und Y benutzen, müssen Sie bei der Anwendung dieses Programms vorsichtig sein. Ihre Werte in X und Y werden nämlich durch das Unterprogramm, das die gleichen Register braucht, zerstört. Sie müssen sie daher zuvor sichern, und der Stapelspeicher ist dazu am besten geeignet.
Unser Programm ist sehr kurz, dafür müssen Sie aber eine Menge Daten in eine Datentabelle laden. Das Register X und der absolute indizierte Adressierungs-Modus dienen zum Laden des Akkumulators mit den ATASCII-Codes aus der Datentabelle. Den ATASCII-Zeichenvorrat finden Sie im Anhang F.


```
                SCHREIBE AUF DEN SCHIRM
100     *=$0600
110   ; SCHREIBE AUF DEN SCHIRM
120     LDX   #$2B        ANZAHL DER ZEICHEN
130   LOOP  TXA
140     PHA               ZWISCHENSPEICHERN DER ZAHL
150     LDA   $1100,X     LADE DEN CODE
160     JSR   $F6A4       ZEICHEN AUSGABE
170     PLA               ZURUECKHOLEN DER ZAHL
180     TAX
190     DEX               COUNT DOWN
200     BNE   LOOP  WENN NOCH NICHT ERLEDIGT, GEHE ZURUECK
210   LOOP1 JMP  LOOP1  ZIRKULIERE HIER
220   END
```

Geben Sie das Programm ein und assemblieren Sie es. Nun müssen die Daten geladen werden. Gehen Sie in den Debugger-Modus und bringen Sie die Daten wie unten angegeben in den Rechner.

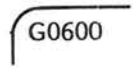
```
.
.
.
EDIT
BUG

DEBUG
C1101<9B,9B,9B,6,E,7,9B      ← Geben Sie am Ende jeder Zeile RETURN

DEBUG
C1108<2,0,16,9B,7,D,6,9B

DEBUG
C1110<0,0,0,9B,44,4E,45,20

DEBUG
C1118<65,68,74,9B,64,65,68,63

DEBUG
C1120<61,65,72,9B,65,76,61,68

DEBUG
C1128<20,75,6F,59

DEBUG
■
```

Wir wollen die Spannung nicht dadurch verderben, daß wir Ihnen hier schon verraten, was all diese Zahlen bedeuten. Wenn Sie es aber nicht abwarten können, brauchen Sie nur in Abb. 12–5 nachzulesen, welches Zeichen durch den jeweiligen Code dargestellt wird.
Ehe Sie das Programm laufen lassen, löschen Sie den Schirm durch Festhalten der SHIFT-Taste und Drücken der CLEAR-Taste.



Schirm ist gelöscht

Führen Sie nun das Programm aus, indem Sie tippen



Und RETURN

Und dies ist das Ergebnis

Die letzten Programmzeilen halten die Ausgabe auf dem Schirm, während der Rechner sich untätig im Kreise dreht. Halten Sie das Programm mit der BREAK-Taste an.
Hier nun die im Programm verwendeten ATASCII-Codes.

Speicher-Adresse	*Hex ATASCII Code*	*Ausgegebene Zeichen*
112B	59	Y
112A	6F	o
1129	75	u
1128	20	LEERSTELLE
1127	68	h
1126	61	a
1125	76	v

1124	65	e
1123	9B	CARRIAGE RETURN
1122	72	r
1121	65	e
1120	61	a
111F	63	c
111E	68	h
111D	65	e
111C	64	d
111B	9B	CARRIAGE RETURN
111A	74	t
1119	68	h
1118	65	e
1117	20	LEERSTELLE
1116	45	E
1115	4E	N
1114	44	D
1113	9B	CARRIAGE RETURN
1112	00	
1111	00	
1110	00	
110F	9B	CARRIAGE RETURN
110E	06	
110D	0D	
110C	07	
110B	9B	CARRIAGE RETURN
110A	16	
1109	00	
1108	02	
1107	9B	CARRIAGE RETURN
1106	07	
1105	0E	
1104	06	
1103	9B	CARRIAGE RETURN
1102	9B	CARRIAGE RETURN
1101	9B	CARRIAGE RETURN

Abb. 12-5 ATASCII Codes für die Darstellung auf dem Bildschirm

Nun können Sie Ihr eigenes Schirmbild erzeugen, indem Sie einfach unsere Codes durch irgendwelche anderen im Anhang F ersetzen. Sie können die Datentabelle verlängern oder kürzen, bringen Sie aber in jedem Fall die entsprechende Zahl in Zeile 120 des Quell-Programms. Füllen Sie Ihren Schirm und amüsieren Sie sich!!

SELBST IST DER MANN

Wir werden Sie an dieser Stelle verlassen. Sie haben nun grundsätzliches Wissen über die Arbeitsweise des Assembler Moduls erworben. Sie haben auch einige Kenntnisse über die Befehlsliste 6502. Mit Ihrem bislang entwickelten Verständnis können Sie den Assembler weiter erforschen. Dabei hilft Ihnen das Atari Benutzer-Handbuch. Sie sollten sich auch mit allen Befehlen der Befehlsliste 6502 vertraut machen. Je mehr Sie mit dem Assembler programmieren, desto größeres Vergnügen wird Ihnen dies bereiten. Für größere Programmiervorhaben in Assembler empfehlen wir den Atari 810 oder 815 Plattenspeicher sowie den Drucker 820, 822 oder 825. Wir haben mit dem Plattenspeicher 810 und dem Drucker 820 gearbeitet und meinen, daß sie für das Assembler-Programmieren sehr gut geeignet sind.
Auf gutes Programmieren!

6502 Anweisung - Betroffene Flag Bits

Die Eintragung X bedeutet, daß das entsprechende Flag Bit betroffen ist. Das Ergebnis hängt von der sich aus einer vorangegangenen Operation ergebenden Bedingung oder deren Status ab. Eine Eins (1) zeigt an, daß das Flag gesetzt ist, eine Null (0), daß das Flag gelöscht ist.

Buchstaben-code	*Durchgeführte Operation*	*Status Flags*							
		N	*V*		*B*	*D*	*I*	*Z*	*C*
ADC	Addiere den Inhalt des Speichers mit Carry zum Akkumulator	X	X					X	X
AND	AND Speicher mit Akkumulator	X						X	
ASL	Verschiebe ein Bit nach links (Speicher oder Akkumulator)	X						X	X
BCC	Springe, wenn kein Carry da (wenn C=0)								
BCS	Springe, wenn Carry da (wenn C=1)								
BEQ	Springe, wenn das Ergebnis Null ist (wenn Z=1)								
BIT	Vergleiche die Bits im Akkumulator mit dem Speicher	X	X					X	
BMI	Springe bei negativem Ergebnis (wenn N=1)								
BNE	Springe, wenn das Ergebnis ungleich Null ist (wenn Z=0)								
BPL	Springe bei positivem Ergebnis (wenn N=0)								
BRK	Force Break (Unterbrechung)						1		
BVC	Springe, wenn kein Überlauf (wenn V=0)								
BVS	Springe, wenn Überlauf (wenn V=1)								
CLC	Lösche das Carry-Flag								0
CLD	Lösche den Dezimalmodus					0			
CLI	Lösche das Interrupt disable Flag						0		
CLV	Lösche das Overflow-Flag		0						
CMP	Vergleiche Speicher und Akkumulator	X						X	X
CPX	Vergleiche Speicher und Index X	X						X	X
CPY	Vergleiche Speicher und Index Y	X						X	X
DEC	Vermindere den Speicherinhalt um eins							X	X
DEX	Vermindere den Index X um eins							X	X
DEY	Vermindere den Index Y um eins							X	X
EOR	Ausschließlich OR (oder) Speicher mit Akkumulator							X	X

Buchstaben-code	Durchgeführte Operation	Status Flags							
		N	V		B	D	I	Z	C
INC	Erhöhe den Speicherinhalt um eins							X	X
INX	Erhöhe den Index X um eins							X	X
INY	Erhöhe den Index Y um eins							X	X
JMP	Springe zur neuen Adresse								
JSR	Springe zur neuen Adresse, merke Rückkehradresse								
LDA	Lade den Akkumulator mit dem Speicherinhalt							X	X
LDX	Lade den Index X mit dem Speicherinhalt							X	X
LDY	Lade den Index Y mit dem Speicherinhalt							X	X
LSR	Verschiebe ein Bit nach rechts (Speicher oder Akkumulator)	0						X	X
NOP	Keine Operation								
ORA	OR Speicher mit Akkumulator	X						X	
PHA	Bringe Akkumulator in den Stapelspeicher								
PHP	Bringe Statusregister in den Stapelspeicher								
PLA	Hole Akkumulator vom Stapelspeicher							X	X
PLP	Hole Statusregister vom Stapelspeicher	X	X	X	X	X	X	X	X
ROL	Rotiere ein Bit nach links (Speicher oder Akkumulator)	X						X	X
ROR	Rotiere ein Bit nach rechts (Speicher oder Akkumulator)	X						X	X
RTI	Rückkehr vom Interrupt	X	X	X	X	X	X	X	X
RTS	Kehre vom Unterprogramm zurück								
SBC	Subtrahiere den Speicher und „borge" vom Akkumulator (negativer Übertrag)	X	X					X	X
SEC	Setze das Carry Flag								1
SED	Setze den Dezimalmodus					1			
SEI	Setze das Interrupt disable Flag						1		
STA	Bringe den Inhalt des Akkumulators in den Speicher								
STX	Bringe den Index X in den Speicher								
STY	Bringe den Index Y in den Speicher								
TAX	Übertrage den Akkumulator in das Indexregister X	X						X	
TAY	Übertrage den Akkumulator in das Indexregister Y	X						X	
TSX	Übertrage den Stapelkopf in das Indexregister X	X						X	
TXA	Übertrage Index X in den Akkumulator	X						X	
TXS	Übertrage Index X in den Stapelkopf								
TYA	Übertrage Index Y in den Akkumulator	X						X	

Flag-Abkürzungen:

N Negative result flag (Vorzeichen Flag Bit)
V Overflow flag (Überlauf Flag Bit)
Expansion flag (Ohne Marke) (Erweiterungs-Flag Bit)
B Break command flag (Unterbrechungs-Flag-Bit)
D Decimal mode flag (Dezimal-Modus Flag Bit)
I Interrupt disable flag
Z Zero result flag (Nullanzeige Flag Bit)
C Carry flag (Übertrag Flag Bit)

6502-Anweisungen - Adressierungs-Modi

BEFEHLSLISTE MIT ADRESSIERUNGS-MODI

Buchstaben Code	Op Codes												
	Akkumulator	Unmittelbar	Null-Seite	Null-Seite, X	Null-Seite, Y	Absolut	Absolut, X	Absolut, Y	Impliziert	Relativ	Indiziert indirekt	Indirekt indiziert	Indirekt
ADC	–	69	65	75	–	6D	7D	79	–	–	61	71	–
AND	–	29	25	35	–	2D	3D	39	–	–	21	31	–
ASL	0A	–	06	16	–	0E	1E	–	–	–	–	–	–
BCC	–	–	–	–	–	–	–	–	–	90	–	–	–
BCS	–	–	–	–	–	–	–	–	–	B0	–	–	–
BEQ	–	–	–	–	–	–	–	–	–	F0	–	–	–
BIT	–	–	24	–	–	2C	–	–	–	–	–	–	–
BMI	–	–	–	–	–	–	–	–	–	30	–	–	–
BNE	–	–	–	–	–	–	–	–	–	D0	–	–	–
BPL	–	–	–	–	–	–	–	–	–	10	–	–	–
BRK	–	–	–	–	–	–	–	–	00	–	–	–	–
BVC	–	–	–	–	–	–	–	–	–	50	–	–	–
BVS	–	–	–	–	–	–	–	–	–	70	–	–	–
CLC	–	–	–	–	–	–	–	–	18	–	–	–	–
CLD	–	–	–	–	–	–	–	–	D8	–	–	–	–
CLI	–	–	–	–	–	–	–	–	58	–	–	–	–
CLV	–	–	–	–	–	–	–	–	B8	–	–	–	–
CMP	–	C9	C5	D5	–	CD	DD	D9	–	–	C1	D1	–
CPX	–	E0	E4	–	–	EC	–	–	–	–	–	–	–
CPY	–	C0	C4	–	–	CC	–	–	–	–	–	–	–
DEC	–	–	C6	D6	–	CE	DE	–	–	–	–	–	–
DEX	–	–	–	–	–	–	–	–	CA	–	–	–	–
DEY	–	–	–	–	–	–	–	–	88	–	–	–	–

Buchstaben Code	Op Codes												
	Akkumulator	Unmittelbar	Null-Seite	Null-Seite, X	Null-Seite, Y	Absolut	Absolut, X	Absolut, Y	Impliziert	Relativ	Indiziert indirekt	Indirekt indiziert	Indirekt
EOR	–	49	45	55	–	4D	5D	59	–	–	41	51	–
INC	–	–	E6	F6	–	EE	FE	–	–	–	–	–	–
INX	–	–	–	–	–	–	–	–	E8	–	–	–	–
INY	–	–	–	–	–	–	–	–	C8	–	–	–	–
JMP	–	–	–	–	–	4C	–	–	–	–	–	–	6C
JSR	–	–	–	–	–	20	–	–	–	–	–	–	–
LDA	–	A9	A5	B5	–	AD	BD	B9	–	–	A1	B1	–
LDX	–	A2	A6	–	B6	AE	–	BE	–	–	–	–	–
LDY	–	A0	A4	B4	–	AC	BC	–	–	–	–	–	–
LSR	4A	–	46	56	–	4E	5E	–	–	–	–	–	–
NOP	–	–	–	–	–	–	–	–	EA	–	–	–	–
ORA	–	09	05	15	–	0D	1D	19	–	–	01	11	–
PHA	–	–	–	–	–	–	–	–	48	–	–	–	–
PHP	–	–	–	–	–	–	–	–	08	–	–	–	–
PLA	–	–	–	–	–	–	–	–	68	–	–	–	–
PLP	–	–	–	–	–	–	–	–	28	–	–	–	–
ROL	2A	–	26	36	–	2E	3E	–	–	–	–	–	–
ROR	6A	–	66	76	–	6E	7E	–	–	–	–	–	–
RTI	–	–	–	–	–	–	–	–	40	–	–	–	–
RTS	–	–	–	–	–	–	–	–	60	–	–	–	–
SBC	–	E9	E5	F5	–	ED	FD	F9	–	–	E1	F1	–
SEC	–	–	–	–	–	–	–	–	38	–	–	–	–
SED	–	–	–	–	–	–	–	–	F8	–	–	–	–
SEI	–	–	–	–	–	–	–	–	78	–	–	–	–
STA	–	–	85	95	–	8D	9D	99	–	–	81	91	–
STX	–	–	86	–	96	8E	–	–	–	–	–	–	–
STY	–	–	84	94	–	8C	–	–	–	–	–	–	–
TAX	–	–	–	–	–	–	–	–	AA	–	–	–	–
TAY	–	–	–	–	–	–	–	–	A8	–	–	–	–
TSX	–	–	–	–	–	–	–	–	BA	–	–	–	–
TXA	–	–	–	–	–	–	–	–	8A	–	–	–	–
TXS	–	–	–	–	–	–	–	–	9A	–	–	–	–
TYA	–	–	–	–	–	–	–	–	98	–	–	–	–

Frequenz-Werte für die Drei-Oktaven-Tonleiter

Tonleiter	Frequenz-Wert	
C	1D	Hoch
B	1F	
A#	21	
A	23	
G#	25	
G	28	
F#	2A	
F	2D	
E	2F	
D#	32	
D	35	
C#	39	
C	3C	
B	40	
A#	44	
A	48	
G#	4C	
G	51	
F#	55	
F	5B	
E	60	
D#	66	
D	6C	
C#	72	
C	79	Mittel
B	80	
A#	88	
A	90	
G#	99	

Tonleiter	*Frequenz-Wert*	
G	A2	
F#	AC	
F	B6	
E	C1	
D#	CC	
D	D9	
C#	E6	
C	F3	Tief

Atari Assembler Fehlercodes

Wenn ein Fehler auftritt, ertönt ein kurzes Tonsignal, und die Fehler-Nummer erscheint auf dem Bildschirm.

Fehler-nummer	*Erklärung*
1	Zu wenig Speicherplatz für das zu assemblierende Programm.
2	Die Zahl xx kann für den „DEL xx, yy"-Befehl nicht gefunden werden.
3	Fehler bei der Spezifikation einer Adresse (Mini-Assembler).
4	Die genannte Datei kann nicht geladen werden.
5	Marke nicht definiert.
6	Syntax-Fehler in der Anweisung.
7	Marke mehr als einmal definiert.
8	Puffer Überlauf.
9	Vor „=" keine Marke.
10	Der Wert eines Ausdrucks ist größer als 255 – ein Byte war verlangt.
11	Ungültige Verwendung eines Null Strings.
12	Adresse oder spezifizierter Adressentyp nicht korrekt.
13	Phasen-Fehler – ein inkonsistentes Ergebnis von Pass 1 nach Pass 2 gefunden.
14	Nichtdefinierter Vorwärtsbezug.
15	Zeile zu lang.
16	Quell-Anweisung vom Assembler nicht bemerkt.
17	Zeile zu lang.
18	LOMEM wurde nach anderem Befehl(en) oder Anweisung(en) gegeben. Lomem muß, falls verwendet, der erste Befehl sein.
19	Keine Anfangsadresse gegeben.

Atari Betriebssystemfehler

Fehler-Nummern über 100 beziehen sich auf das Betriebssystem und das Platten-Betriebssystem (Disk Operating System). Eine vollständige Liste der DOS-Fehler finden Sie im DOS-Manual.

Fehler-nummer	*Erklärung*
128	Während einer Eingabe/Ausgabe-Operation wurde die BREAK-Taste berührt.
130	Ein nicht existentes Gerät wurde spezifiziert.
132	Für das ausgewählte Gerät ungültiger Befehl.
136	END OF FILE READ HAS BEEN REACHED. Dieser Fehler kann beim Lesen von der Kassette auftreten.
137	Satz (record) war länger als 256 Zeichen.
138	Das im Befehl spezifizierte Gerät antwortet nicht. Prüfen Sie den Anschluß und die Stromversorgung.
139	Das im Befehl spezifizierte Gerät verweigert das OK-Signal.
140	Serieller Format-Fehler.
142	Serieller Format-Fehler.
143	Serieller Prüfsummen-Fehler.
144	Geräte-Fehler.
145	Disketten-Fehler – Fehlerhaftes Abspeichern.
146	
162	Platte voll.
165	Dateinamen-Fehler.

ATASCII Zeichenvorrat

ATASCII ist eine Abkürzung von ATARI ASCII. Buchstaben und Zahlen haben die gleichen Werte wie die in ASCII, bis auf einige Spezialzeichen. Die Zeichen 80-FF (hex) sind Austauschfarben von 1-7F, ausgenommen die gezeigten Zeichen (9B-9F und FD, FE und FF).

HEX	*ZEICHEN*	*HEX*	*ZEICHEN*	*HEX*	*ZEICHEN*
0		A		14	
1		B		15	
2		C		16	
3		D		17	
4		E		18	
5		F		19	
6		10		1A	
7		11		1B	
8		12		1C	
9		13		1D	

HEX	ZEICHEN
1E	
1F	
20	Leerstelle
21	!
22	"
23	#
24	$
25	%
26	&
27	'
28	(
29	)
2A	*
2B	+
2C	,
2D	-
2E	.
2F	/
30	0
31	1

HEX	ZEICHEN
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9
3A	:
3B	;
3C	<
3D	=
3E	>
3F	?
40	@
41	A
42	B
43	C
44	D
45	E

HEX	ZEICHEN
46	F
47	G
48	H
49	I
4A	J
4B	K
4C	L
4D	M
4E	N
4F	O
50	P
51	Q
52	R
53	S
54	T
55	U
56	V
57	W
58	X
59	Y

HEX	ZEICHEN
5A	Z
5B	[
5C	\
5D	]
5E	^
5F	—
60	
61	a
62	b
63	c
64	d
65	e
66	f
67	g
68	h
69	i

HEX	ZEICHEN
6A	j
6B	k
6C	l
6D	m
6E	n
6F	o
70	p
71	q
72	r
73	s
74	t
75	u
76	v
77	w
78	x
79	y

HEX	ZEICHEN
7A	z
7B	
7C	
7D	
7E	
7F	
9B	Neue Zeile
9C	
9D	
9E	
9F	
FD	 Summer
FE	 Zeichen löschen
FF	 Zeichen einfügen

Stichwortverzeichnis

Notizen

Notizen

Notizen

Notizen

Der ATARI® Assembler

Don Inman Kurt Inman

Hier können Sie das Programmieren in Assembler lernen und sich gleichzeitig mit der Anwendung des Atari Assembler Moduls auf Ihrem Atari 400- oder 800-Modell vertraut machen. Dies Buch ist eine ausgezeichnete Einführung für Leser mit einigem Grundwissen in BASIC, setzt aber *keinerlei* Assembler-Kenntnisse voraus.

Ihr BASIC-Grundwissen führt Sie nach und nach zum Assembler-Programmieren hin. Der ATARI ASSEMBLER geht in einfacher Weise durch jedes Programm. Bildschirmskizzen zeigen Ihnen jedes Stadium der Eingabe bzw. Ausführung der Beispielprogramme. Schrittweise Befehlsverarbeitung hilft Ihnen, von BASIC zur Atari-eigenen Sprache zu gelangen.

Die Autoren wissen, wie wichtig beim Lernen die Praxis ist. Beim Kennenlernen des Atari Assembler Moduls sind Sie gezwungen, alle Anweisungen und Programme auf dem Rechner auszuprobieren und Ihre Ergebnisse auf dem Schirm zu überprüfen. Wenn Sie dies Buch durchgearbeitet haben, schreiben Sie bereits Ihre eigenen Programme in Assembler.

Inmann
Inmann

Der ATARI® Assembler

IDEA