SONDERHEFT 71 Markt&Technik öS 120,-/sfr 16,-/lir 16000 hfl 21,-/dkr 75,-/fmk 62 DM 16,-

DISKETTE IM HEFT

64'er

Über 50 Programme auf Diskette 64'er

ASSEMBLER

LEICHT GEMACHT

Intensivkurs
Von Basic zur Maschinensprache

Wandposter
Alle Befehle auf einen Blick

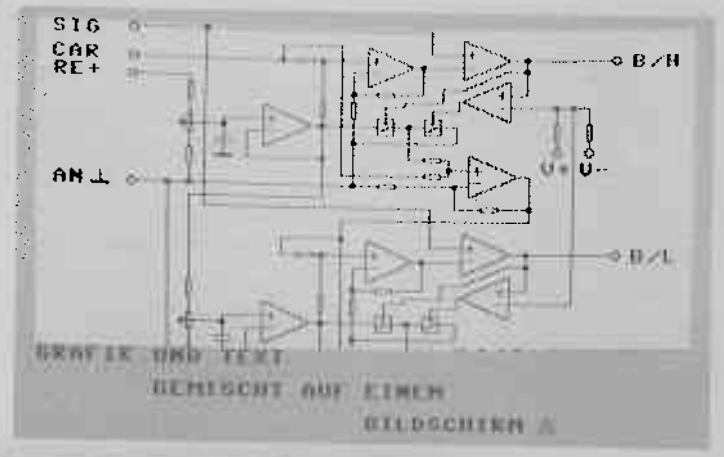
Lösungen
Assembler-Programmierung

Komplettpaket
Assembler, Reassembler, Speichermonitor

Tips & Tricks
Für Anfänger und Profis

INHALT

Getellter Bildschirm
Hires und Text zusammen bietet »Splitscreen«
Seite 49



Kurs

Poster

Tools

Mehr als 16 Farben
Wer hätte das gedacht – 136 Farben am C64
Seite 45

Tips & Tricks

Sonstiges

Alle Programme zu Artikeln mit einem -Symbol finden Sie auf der beiliegenden Diskette (Seite 19).

. . . auch nur mit

Der C64 bietet Grafik, Sound und vieles mehr. Doch für denjenigen, der alles ausprobieren möchte, ist die erste Zeit hart. Aus dem (etwas kümmerlichen) Handbuch werden die Basic-Beispiele mühsam in die Tastatur geklopft. Irgendwann kommen (Gott sei Dank!) die ersten Erfolgserlebnisse. Wer allerdings nach den ersten durchprogrammierten Nächten festgestellt hat, daß zwar alles funktioniert – aber unendlich langsam – steht vor der Alternative: Ich werde reiner Anwender oder versuche es mal mit Maschinensprache. Für den leichteren Weg gibt es eine Unzahl von Programmen. So viele, daß allein mit den Titeln ein Buch gefüllt werden könnte. Um hier zum Erfolg zu kommen, hilft nur suchen, kaufen, tauschen.

Für alle anderen beginnt ein harter Kampf nach Grundinformationen, Tips und Hilfestellungen – und genau für diese Gruppe zukünftiger Fachleute ist dieser Kurs gedacht.

Was ist Maschinensprache?

Beginnen wir unseren Excurs mit der Anwendersprache Basic, genau so, wie sich unser C64 nach dem Einschalten meldet. Halt! Diese Behauptung stimmt nicht, denn bis zu dem Augenblick, da Sie die Einschaltmeldung am Bildschirm sehen, hat der Computer schon eine Unmenge von Maschinenprogrammen abgearbeitet. Beispielsweise wurde der Bildschirm-Chip initialisiert, alle Sound-Parameter zurückgesetzt und der Speicher auf die (vielleicht) folgende Arbeit vorbereitet. Und das ist noch lange nicht alles: auch jetzt ist er damit beschäftigt, den Cursor blinken zu lassen und wartet auf Ihre Eingaben. Selbst wenn Sie jetzt die berühmte Befehlsfolge

```
PRINT"HALLO"
```

eingeben, und diesen Text mit <RETURN> bestätigen, beginnt er wieder eine Reihe Maschinenprogramme abzuarbeiten. Zuerst wird überprüft, ob die Schreibweise von »PRINT« einem Befehl entspricht, dann werden die darauffolgenden Zeichen eingelesen, gecheckt usw. Danach erscheint – nach langer Arbeit für den C64, aber in Sekundenbruchteilen für uns – der Text »HALLO« am Bildschirm. Die Unzahl von Maschinenprogrammen, die unser Computer gerade abgeschlossen hat, nennen wir »Basic«. Diese Programmiersprache ist ein ausgeklügeltes System, das alle Eingabefehler kommentiert, ja zum Teil verhindert und uns komfortable Befehle zur Verfügung stellt. Alle Systemroutinen bestehen aus einer Folge von Zahlen, die der Reihe nach abgearbeitet werden.

Irgendwer (Microsoft) hat irgendwann (1976) einmal das Grundkonzept von Basic entworfen. Und da es ziemlich aufwendig ist, aus einer Tabelle von Befehlen entsprechende Zahlen herauszusuchen und diese dann einzeln in den Speicher zu bringen, verwendete man ein Hilfsprogramm, den Assembler. Er stellt (wieder) ein Maschinenprogramm zur Verfügung, welches leicht zu merkende (3buchstabige) Mnemonics in einen vom Mikroprozessor direkt verwendbaren Code umwandelt. Diese Bytes werden ab einer vom Programmierer bestimmten Stelle im Speicher abgelegt.

Übrigens ist der Mikroprozessor nicht das einzige Bauelement unseres C64. Es gibt zusätzlich: den Video-Chip (VIC), einen Sound-Chip (SID), eine Menge Schaltkreise, die alles verknüpfen und natürlich die 64 K-Byte Schreib-Lese-Speicher, von denen unser Computer seinen Namen hat. Sie alle stehen aber unter dem Kommando des Mikroprozessors.

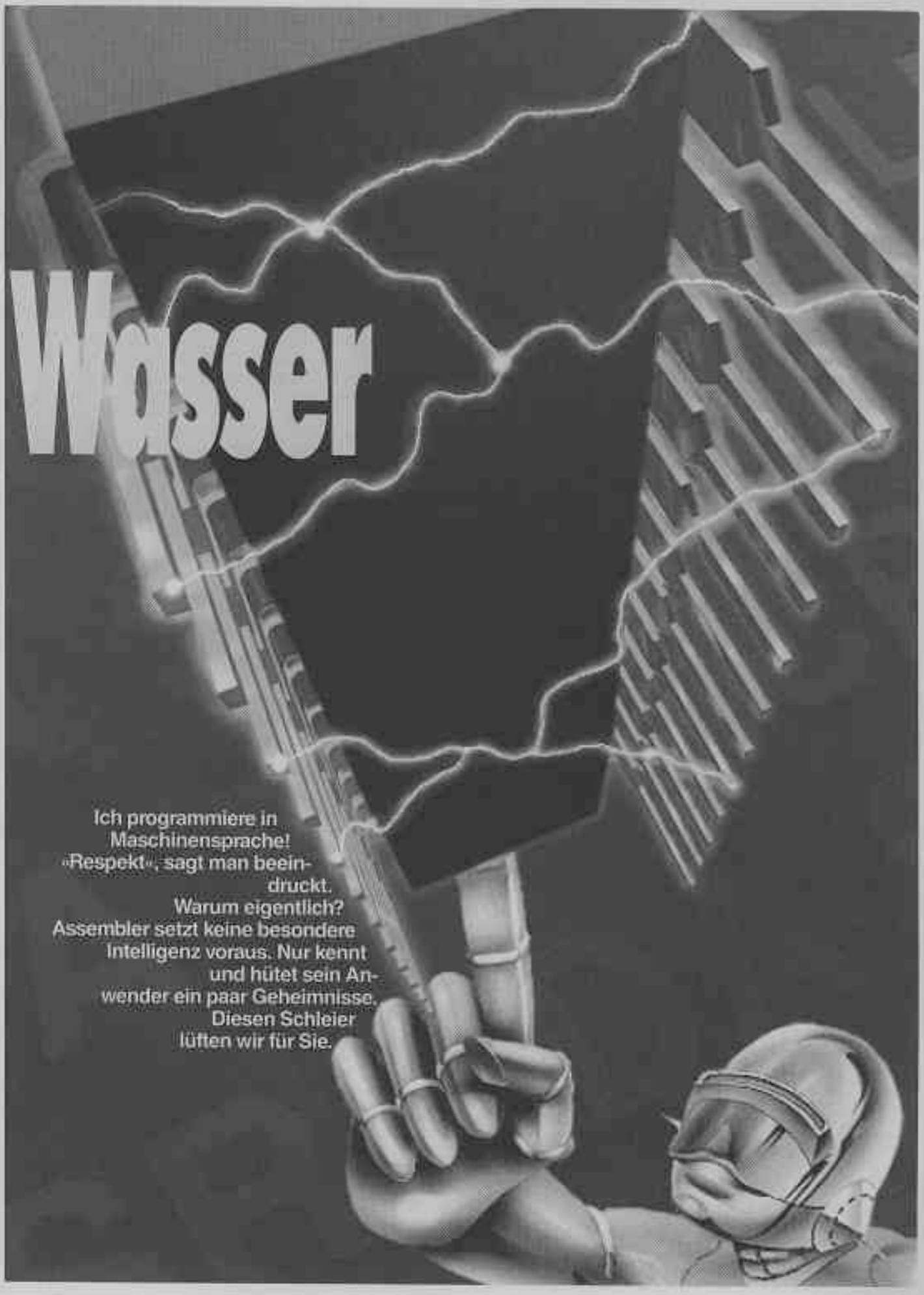
Wie funktioniert der Mikroprozessor?

Sie haben es vorhin schon gehört: Ein Computer verarbeitet Zahlen. Das ist so, auch wenn Sie am Bildschirm Grafiken bewundern, der Printer Texte druckt, und der Lautsprecher Töne von sich gibt. Alle Tätigkeiten finden ihren Ursprung in einer Anreihung von Zahlen im Speicher, die der Mikroprozessor analysiert und die ihm sagen, was er zu tun hat – und (sehr wichtig!) wie viele der nachfolgenden Speicherstellen er dazu benötigt. Gehen wir von einem unverrückbar festen Zustand aus: dem RESET. Immer wenn Sie den C64 einschalten oder den RESET-Knopf (falls vorhanden) drücken, beginnt der Mikroprozessor zu arbeiten. Er springt an eine definierte Stelle im Speicher und liest den ersten Zahlenwert. Man kann diese erste Zahl auch als Befehl bezeichnen. Er legt fest, wie viele der nachfolgenden Speicherstellen zur Befehlsausführung nötig sind. Es können null bis zwei Byte sein.

Was ist ein Absturz

Damit wird auch klar, was passiert, wenn Sie willkürlich an eine Stelle im Programm springen (z.B. mit »SYS64738« aus Basic). Erwischt der Mikroprozessor eine Stelle, an der ein zufälliger Wert, aber kein Befehl steht, versucht er diese Zahl als Anweisung zu interpretieren. Das funktioniert natürlich nicht, da die nachfolgenden Stellen nichts mit diesem vermeintlichen Befehl zu tun haben, also unserem Mikroprozessor falsch einsagen. Meistens (so auch bei unserem Beispiel) führt dies zum Absturz des Systems (d.h. unser C64 läßt sich erst wieder durch RESET oder Ein- und Ausschalten zum Leben erwecken). Manchmal passieren auch seltsame Dinge: z.B. die Bildschirmfarben ändern sich oder der Cursor blinkt schneller usw. Diese Reaktion kann übrigens auch bei einem Programmierfehler auftauchen. Sie ist der unangenehmste Unterschied zu einer Hochsprache: Basic fängt alle falschen Eingaben ab, steigt mit einer Fehlermeldung aus dem laufenden Programm und der Programmierer kann anhand der Fehlermeldung seine falsche Eingabe korrigieren – danach versucht er's halt nochmal.

Ein Maschinenprogramm dagegen stürzt bei gravierenden Fehlern ab, d.h. der Computer reagiert einfach nicht mehr. Um den Fehler zu analysieren, muß er zuerst aus seinem Tiefschlaf gerüttelt werden, erst danach läßt sich das Programm überprüfen. Doch wie hole ich den Computer wieder ins Leben zurück, wenn er hängt?


Wasser
Ich programmiere in Maschinensprache! «Respekt», sagt man beeindruckt. Warum eigentlich? Assembler setzt keine besondere Intelligenz voraus. Nur kennt und hütet sein Anwender ein paar Geheimnisse. Diesen Schleier lüften wir für Sie.

1. Es gibt die brutale Methode: aus- und wieder einschalten. Sie funktioniert zwar immer, aber hat den Nachteil, daß unser Programm (falls überhaupt gespeichert) wieder neu geladen werden muß.

2. Die elegantere Methode ist ein RESET-Taster. Er wurde zwar vom Hersteller weggelassen, läßt sich aber leicht nachrüsten (Sonderheft 57, Seite 44). Drei Möglichkeiten bestehen für die Installation: Extern am User-Port, Extern am seriellen Ausgang, oder intern eingebaut. So ein Taster gaukelt dem C64 vor: "Du bist gerade eingeschaltet worden". Und tatsächlich, der Mikroprozessor beginnt wieder zu leben. Im Unterschied zum Ausschalten ist aber der Speicher nicht gelöscht (wenn es auch zunächst so aussieht). Die Programme sind alle noch vorhanden und lassen sich mit ein bißchen Know-how zurückholen (Sonderheft 57, Seite 24 »Reset ohne Reue«).

Begriffserklärungen

Byte

Die Behauptung, ein Computer könne nur Zahlen verarbeiten, stimmt nicht ganz, denn er kennt eigentlich nur zwei Zustände: Strom eingeschaltet oder nicht. Da es für alles eine Bezeichnung gibt, nennen wir dieses Geschehen 1 Bit. Damit allein läßt sich natürlich nicht viel anfangen. Denken Sie an Ihre Nachttischlampe: Sie ist entweder eingeschaltet oder nicht, und damit sehen Sie entweder etwas oder nicht. Um mehr Möglichkeiten zu erreichen, hat man mehrere dieser Bits zusammengefaßt. Denken Sie an zwei Nachttischlampen: Entweder Sie sehen etwas, oder Ihr Bettnachbar, oder beide. Beim (binären) Zählen steht jeder Möglichkeit eine Zahl gegenüber. Nehmen wir als Beispiel die Kombinationen von 4 Bit. Mit ihnen lassen sich 16 Variationen (Zahlen) darstellen. Welche davon einem Zahlenwert entspricht, wurde folgendermaßen festgelegt:

```
 0 = 0000
 1 = 0001
 2 = 0010
 3 = 0011
 4 = 0100
 5 = 0101
 6 = 0110
 7 = 0111
 8 = 1000
 9 = 1001
10 = 1010
11 = 1011
12 = 1100
13 = 1101
14 = 1110
15 = 1111
```

Sie sehen hier wird ein festes System verwendet. Fügt man ein Bit hinzu, verdoppelt sich jeweils die Anzahl. Unser C64 faßt genau 8 Bit zusammen (und bearbeitet sie auch gleichzeitig). Damit ergeben sich 256 Kombinationen. Da man diese Zusammenfassung ein Byte nennt, läßt sich pro Byte eine Wertereihe von 0 bis 255 darstellen.

Mnemonic

... ist lt. Fremdwörterlexikon ein Mittel, um das Gedächtnis durch Merk- oder Lernhilfsmittel zu unterstützen, die mit dem zu Merkenden in äußerliche Verbindung gebracht werden können. In verständlicherem Deutsch sind dies (mehr oder weniger) verständliche Abkürzungen für Tätigkeiten, die der Rechner erledigen soll. Beispielsweise bedeutet »LDA«: Lade den Akku (das Hauptregister s.unten).

Mikroprozessor (CPU)

... ist der wichtigste Bestandteil eines Heimcomputers. Und in der Tat ist dieses Bauteil ein (mikroskopisch kleiner, ca. 5 x 6 mm großer) integrierter Schaltkreis (IC) mit der Bezeichnung »6510«, der alle anderen elektronischen Bauteile Ihres C64 steuert und zusätzlich noch eigenständige Berechnungen durchführt. Damit man ihn überhaupt im Rechner einlöten kann, machte man ihn künstlich größer - er wurde in einen Plastikmantel eingegossen. Ohne den Mikroprozessor könnte unser C64 nicht einmal den Cursor am Bildschirm darstellen. Der 6510 ist ein Nachfolgetyp des bei Commodore häufig verwendeten 6502 (2001 PET, 30xx-Serie, Floppystationen usw.). Die Befehlssätze beider Mikroprozessoren sind identisch. Der größte Unterschied zu seinen Vorgängern ist ein eingebauter Port (sechs herausgeführte, frei programmierbare Leitungen). Dieser steuert im C64 die Kassettenfunktionen und die Speicherverwaltung, denn der C64 verwaltet mehr als 64 KByte Speicher. Aber dazu kommen wir später.

Warum braucht man einen Assembler?

Wie wir schon mehrmals gehört haben, besteht ein Maschinenprogramm aus Zahlenkolonnen unterschiedlicher Länge. Für die Kombination von Befehlen und den nachfolgenden Bytes gibt es schier unzählig viele Kombinationen (s. Poster S. 26/27). Also ist die Methode: Befehl aus der Tabelle heraussuchen, in die Speicherstelle POKEn, die Anzahl der nötigen nachfolgenden Zahlen suchen und danach deren Werte in die nächsten Speicherstellen POKEn - nicht gerade effektiv. Das Hilfsprogramm »Assembler« nimmt uns diese Arbeit ab. Es übersetzt für uns verständliche Bezeichnungen (Mnemonics) in Zahlen (Befehlscode) für den Mikroprozessor, berechnet die Anzahl der nachfolgenden Speicherstellen und füllt sie mit den richtigen Werten. Zusätzlich wird beim Assemblieren (Übersetzen in Maschinencode) ein Protokoll ausgegeben. Leichte Fehler werden sogar angezeigt. Zusätzlich läßt sich der Quelltext auf Diskette speichern, später wieder laden und umarbeiten. Damit entfällt auch ein Nachteil der direkten Maschinenprogrammierung: bei jeder noch so kleinen Änderung muß (durch unterschiedliche Befehlslängen bedingt) alles neu berechnet und eingetragen werden. Der Assembler macht's automatisch.

Woraus besteht der Mikroprozessor?

Das ist ein so komplexes Thema, daß wir damit mehr als ein Buch füllen könnten. Lassen wir darum komplexe Details weg:

Zunächst ist der Mikroprozessor ein eigenständiges Bauteil, das intern aus mehreren einzelnen Baugruppen besteht. Damit diese auch wirklich Hand in Hand arbeiten, benötigt die CPU einen Arbeitstakt (Clock). Man kann ihn sich wie ein Pendel vorstellen, das den Prozessor für jeden Arbeitsprozeß einmal anstößt. Im Gegensatz zu einem Uhren-Pendel, bewegt sich das Clock-Pendel sehr schnell, nämlich 970000 mal pro Sekunde.

Im 6510 existiert für jede Aufgabe eine einzelne Baugruppe:

1. Die arithmetische Recheneinheit (ALU - Arithmetic Logic Unit) - ist für Organisation und Rechenoperationen zuständig.

2. Das Hauptregister (Akku) - normalerweise wird hier ein Byte geladen, bearbeitet und wieder zurück in den Speicher geschrieben. Wir werden später sehen, daß einige Operationen auch mit anderen Registern möglich sind.

3. x-Register - ein Hilfsregister, das eingeschränkte Rechenoperationen beherrscht.

4. y-Register - das zweite Hilfsregister, auch mit ihm sind eingeschränkte Rechnungen möglich.

5. Statusregister - spiegelt die Reaktionen auf Rechenoperationen wider. Man sagt auch »zeigt den Prozessorstatus an«.

6. zusätzliche Hilfsregister (Status und Stapel)

7. der Programmzähler (PC - hier: Programm-Counter)

Alle Bestandteile, mit der Ausnahme des PCs, können nur mit einem Byte umgehen. Der Programm-Counter wird zwar mit 16 Bit angesprochen, besteht aber aus zwei 8-Bit-Registern (PCL, PCH). Der Grund dafür ist, daß er alle Speicherzellen anwählen und erreichen muß. 16 Bit ergeben 65536 Möglichkeiten, also genau die Anzahl, die unser C64 an Speicherzellen verwaltet. Zum Unterschied zu dieser elektronischen Baugruppeneinteilung gibt es die für Sie bedeutend wichtigere Gliederung in programmierbare Einheiten. Im Textkasten »Programmierbare Bestandteile des 6510« sind

Programmierbare Bestandteile des 6510

Programmzähler (PC)

... sein Inhalt bestimmt, aus welcher Stelle im Speicher der Mikroprozessor die nächsten Daten liest. Vereinbart ist, daß der erste gelesene Wert einem Befehl entspricht. Dieser Zahlenwert (Befehl) bestimmt, was getan werden soll und wieviel Speicherzellen zur Ausführung benötigt werden. Von diesem Zahlenwert hängt es also ab, welche Register angesprochen werden, ob der Programmzähler selbst manipuliert wird (z.B. durch eine Reaktion auf ein Rechenergebnis) oder eine Rechnung mit dem Hauptregister durchgeführt wird. Der Programmzähler kann Werte von 0 bis 65535 verarbeiten und damit jede Stelle des Speichers erreichen. Befehle, die den Programmcounter ändern, sind:

JMP, JMP ($$$$), RTS, RTI, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BRK und NOP

Hauptregister (Akku)

... stellt das wichtigste Register dar. Mit ihm läßt sich aus Speicherstellen laden, addieren, subtrahieren, logisch mit einer Speicherstelle oder einem Wert verknüpfen und das Ergebnis in eine Speicherstelle oder in das x-, y-Register übertragen. Ebenso läßt sich ein Wert direkt aus den x-/y-Registern in den Akku übertragen. Der Akku kann nur Werte von 0 bis 255 verarbeiten. Befehle für den Akku:

LDA, STA, TAX, TXA, TAY, TYA, ASL, ROL, LSR, ROR, AND, EOR, ORA, BIT, ADC, SBC, CMP, PLA, PHA, PHP und PLP

x-Register

... ist das erste Hilfsregister, kann aufwärts und abwärts zählen und Additionen bzw. Subtraktionen durchführen. Zusätzlich hat es eine wichtige Aufgabe: sein Inhalt läßt sich mit einer Reihe von Befehlen mit dem Akku oder einer Speicherstelle verknüpfen und deutet dann auf Speicherstelle+x (x-indizierte Adressierung). Auch hier ist nur ein Wertebereich von 0 bis 255 erlaubt. Befehle, die direkt das x-Register betreffen:

LDX, STX, INX, DEX, TSX und TXS.

Befehle, bei denen das x-Register beteiligt ist:

LDA $$$$,X, LDY $$$$,X und STA $$$$,X

y-Register

... ist das zweite Hilfsregister und kann das meiste des x-Registers (Ausnahme TSX und TXS). Zusätzlich hat es eine Erweiterung der Indizierung. Zusammen mit dem Akku kann es jeweils zwei Speicherstellen der Zero-Page (= Speicherbereich < 255) als Zeiger (Pointer) verwenden. Das Interessante dabei ist: nicht der Wert aus diesen Speicherstellen wird verwendet, sondern die Adresse aus den Zellen, plus den y-Wert. Wie das genau funktioniert, werden wir noch kennenlernen. Das y-Register verkraftet Werte von 0 bis 255.

Befehle, die direkt das y-Register betreffen:

LDY, STY, INY und DEY

Befehle, bei denen das y-Register beteiligt ist:

LDA $$$$,Y, LDX $$$$,Y, STA $$$$,Y, LDA ($$$$),Y und STA ($$$$),Y

Statusregister

...zeigt die Reaktionen bei Rechenoperationen an, z.B. ein negatives Ergebnis, ein Überfließen oder ein Ergebnis = 0. Dabei hat jedes einzelne Bit eine eigene Funktion. Sie werden sich sicher gefragt haben, wie man größere Zahlen als 255 bearbeitet, wenn im Akku und den einzelnen Registern nur Werte einschließlich 255 erlaubt sind. Das Statusregister gibt Auskunft darüber, ob bei einer Rechnung das Ergebnis kleiner 0, gleich 0 ist, oder etwa der Bereich (255) überschritten wurde. Anhand dieses Zustands kann (muß aber nicht) auf einzelne Ereignisse reagiert werden. Um aber für Rechenoperationen die Voraussetzungen zu schaffen, lassen sich einzelne Bits direkt beeinflussen. Zusätzlich läßt sich durch Setzen eines Bits der IRQ sperren (SEI) oder durch Löschen (CLI) wieder freigeben. Ein neuer Begriff – IRQ. Sie haben sicher schon einmal davon gehört. Jede 60stel-Sekunde unterbricht der Mikroprozessor sein Programm, merkt sich den letzten Status, und führt (bei Basic) die Tastaturabfrage durch. Dieser Zustand läßt sich im Wortschatz des Mikroprozessors mit einem speziellen Befehl ab- und wieder einschalten. Damit wird das entsprechende Bit im Status-Register gesetzt. Befehle, die einzelne Bits des Status-Registers beeinflussen:

SEI, CLI, CLC, SEC, CLV, CLD und SED.

Stack

... ist der Zwischenspeicherbereich (Stapel) des Mikroprozessors. Die Bezeichnung »Stapel« ist bezeichnend für das Prinzip der Speicherung: »last in, first out« (das zuletzt gestapelte muß als erstes wieder hinaus). Wir alle kennen dieses Prinzip. Stellen Sie sich einen Stapel Papier vor. Das zuletzt abgelegte Papier muß als erstes wieder entnommen werden, um an die unteren zu kommen. Genau nach diesem Prinzip funktioniert der Prozessor-Stack (Adresse 256 bis 511). Interessant ist, daß Sie sich um die Verwaltung des Stapels nicht kümmern müssen: er wird automatisch vom Prozessor verwaltet.

Beispiel: Die CPU merkt sich beim IRQ die Position, an der das Programm unterbrochen wurde (zusätzlich noch den Status). Nach Beenden des Interrups holt er sich diese Daten wieder. Um sie aber zu finden, muß er sich merken, an welcher Stelle im Stack er sich gerade befindet, denn darunter befinden sich die vorher abgelegten Daten. Dafür ist der Stack-Pointer zuständig. Sein Inhalt (0 bis 255) kann durch einige direkte Befehle verändert werden. Doch Vorsicht: ein unkontrolliertes Manipulieren des Stacks führt fast immer zum Absturz des Mikroprozessors:

PHA und PLA.

diese Einheiten beschrieben. Zusätzlich erhalten Sie hier eine Auflistung der Mnemonics für die betreffenden Befehle. Eine Erklärung folgt später.

Die Philosophie von Maschinensprache

In Basic bewirkt ein Befehl, je nachdem unter welchen Voraussetzungen Sie in verwendet haben, eine Reihe von Reaktionen. Nehmen wir als Beispiel »PRINT«: Dieser Befehl gibt einen Text auf den Bildschirm, auf den Drucker oder sogar auf die Floppy aus. So komplexe Anweisungen gibt es in Maschinensprache nicht. Hier begeben Sie sich zu den Wurzeln des C64. Jeder von 56 Befehlen ruft eine ganz bestimmte Reaktion des Computers hervor. Daß eine Reihenfolge dieser Maschinenbefehle aber wieder komplexe Reaktionen hervorrufen können, sieht man anhand von Basic. Es besteht aus vielen unterschiedlichen Maschinenprogrammen, die sich gegenseitig ergänzen. Hier schließt sich der Kreis. Der Ursprung aller Programmiersprachen sind Maschinenprogramme, die sich selbst überprüfen und zum Teil sehr komplexe Aufgaben erfüllen. Wie das möglich ist, wollen wir uns anhand eines Beispiels ansehen:

```
POKE 1024,13
```

Wie Sie wissen, bringt dieser Befehl in die Speicherstelle 1024 den Wert 13. Das ist nichts Außergewöhnliches, werden Sie sagen. Aber zufällig ist »1024« der Beginn des Bildschirmspeichers (Ende = 2024). Er ist ein Teil des normalen Speichers des C64 – mit einem Unterschied: Von Zeit zu Zeit (jede 50stel Sekunde) liest der Video-Interface-Chip (VIC) diesen Speicherbereich und schaut nach, welche Werte hier vorhanden sind. Nach (für uns) nicht spürbarer Zeit hat er ein Muster aus einem anderen Speicherbereich herausgesucht, das diesem Wert entspricht und ein »M« links oben dargestellt.

»PRINT« kann das auch, sogar noch mehr. Löschen Sie doch mal den Bildschirm mit <SHIFT CLR/HOME>, fahren Sie mit dem Cursor ein paar Zeilen tiefer und geben Sie ein:

```
PRINT CHR$(19)"M"
```

Nach <RETURN> erscheint »M« wieder an der gleichen Stelle wie vorher.

Der gravierende Unterschied zwischen beiden Methoden ist die Philosophie, die dahintersteckt. Beim POKEn wurde ein Maschinenbefehl simuliert. Wir benötigten die Kenntnis, wo der Bildschirmspeicher beginnt und in welche Speicherstelle wir welchen Wert POKEn mußten, um das »M« am richtigen Platz sichtbar zu machen. Beim PRINT mußten wir nur angeben: gehe an die obere, linke Bildschirmposition (CHR$(19)) und stelle dort ein »M« dar ("M").

Sie sehen den Unterschied zwischen Maschine und Basic. Bei Basic gibt es umfassende Befehle, die Ihnen Denkarbeit abnehmen, bei Maschine benötigen Sie genauere Kenntnis, was im Computer vorgeht. Natürlich gibt es Programmierer, die alle diese Dinge auswendig beherrschen, aber das sind die wenigsten. Für Sie empfehlen wir ein paar Grundlagenwerke und -artikel, in denen beschrieben wird, welche Speicherstellen für welchen Zweck reserviert sind. Denn beim C64 sind alle Reaktionen durch Schreiben (in Basic POKEn) und Lesen (PEEK) in/aus Speicherstellen möglich. Man muß nur wissen wo. Dazu einige Literaturverweise zu unseren 64'er Sonderheften (Preis: 16 Mark inkl. Diskette):

»Sprites«, SH 62, S.34
»Zero-Page« SH 65, S.24
»Port-Bausteine« SH 65, S. 30
»VIC+Interrupt« SH 65, S. 34
Markt & Technik Leserservice, CSJ, Postfach 1 40 20, 8000 München 5, Tel. 089/2025 1528

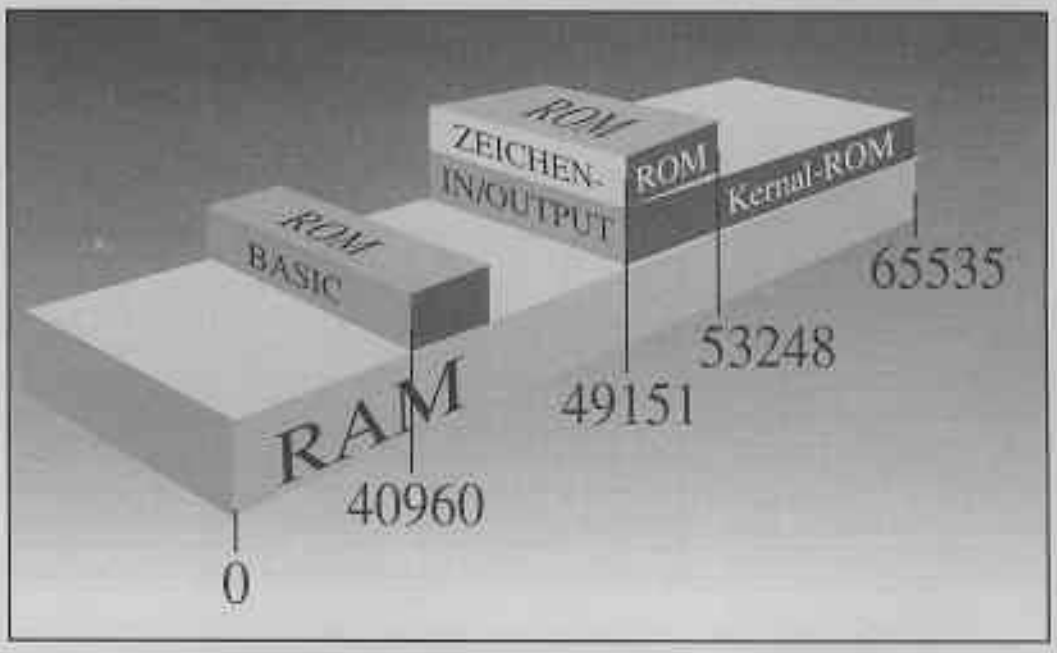
Zusätzlich empfehlen wir Ihnen »das« Standardwerk für den Assembler-Programmierer:

Brückmann/Englisch/Fell/Gelfand/Gerits/Krsnik, Das neue Commodore-64-Intern-Buch, 836 Seiten, 29,80 Mark, ISBN 3-89011-307-9, Data Becker GMBH, Merowingerstr. 30, 4000 Düsseldorf 1, Tel. 0211/31 00 10.

Die Speicheraufteilung des C64

Erinnern Sie sich noch an die Aussage: der C64 verwalte mehr als 64 KByte? Wir beschlossen, die Klärung dieser Behauptung auf später zu verschieben. Jetzt ist es soweit.

Wegen der Struktur des Adressbusses kann unser C64 maximal 64 KByte auf einmal verwalten. Tatsache ist allerdings, daß unser Computer mehr Speicher besitzt. Es sind dies 64 KByte RAM (Schreib-/ Lesespeicher), 20 KByte ROM (Nur Lesespeicher) und 4 KByte, über die alle Zusatzchips (VIC, SID, CIAs) angesprochen werden. Klar, im ersten Augenblick ist es etwas unverständlich, wie diese zusätzlichen Speicherbereiche mit ins Gesamtkonzept passen. Des Rätsels Lösung: Gehen wir davon aus, daß es nicht nötig ist, immer den gesamten Schreib-/Lesespeicher zur Verfügung zu haben; denn wenn der Mikroprozessor ein Programm abarbeitet, interessiert ihn für die Ausführung in erster Linie nur der Bereich, in dem das Programm steht. Etwas anders sieht es mit den Speicherzellen aus, in die er Werte überträgt, oder aus denen er Daten holt. Sie können überall im Speicher verstreut liegen. Was spricht also dagegen, bei verschiedenen Datenbereichen zwischen einzelnen Bausteinen umzuschalten (Bank-switching)? Für Sie als Programmierer bedeutet dies allerdings: bevor Sie diese Speicherbereiche manipulieren, muß im Programm festgelegt sein, welches der Bauteile angesprochen werden soll. Damit dieses Verfahren für den Anwender nicht zu kompliziert wird, hat man sich darauf beschränkt, nur größere Blöcke (4 bis 8 KByte) in festgelegten Bereichen umschaltbar zu machen. Zur Verdeutlichung sind die erreichbaren Bausteine und deren Bereich in Abb.1 übereinander gezeichnet.



[1] Das RAM des C64 mit den über Speicherstelle 1 erreichbaren ROMs

Bei der Erklärung des Begriffs »Mikroprozessor« haben Sie schon den eingebauten Port kennengelernt. Er wird über Speicherstellen $00 und $01 angesprochen und steuert dieses Bank-switching; aber auch (damit es nicht zu einfach wird) die Kassettenoperationen. Betrachten wir uns zuerst $00, das Datenrichtungsregister. Es bestimmt, wie der Name auch sagt, die Datenrichtung der einzelnen Leitungen des Prozessor-Ports ($01):

Nach dem Einschalten wird diese Speicherstelle auf den Wert 47 gesetzt. Als Dualzahl entsteht damit »00101111«. Jedes der einzelnen Bits entspricht der Richtung des Prozessor-Ports; wobei ein gesetztes Bit (1) die entsprechende Leitung auf Ausgabe, ein gelöschtes Bit (0) auf Eingabe schaltet. Damit sind die drei niederwertigen Bits auf Ausgabe geschaltet (rechts). Und tatsächlich, über diese drei Leitungen wird ein spezielles Bauteil, zuständig fürs Bank-switching, gesteuert. Die drei nächsten Bits der Dualzahl sind für Kassettenoperationen zuständig. Die letzten Bits verdienen keinerlei Beachtung, da der Port nur sechs Leitungen besitzt.

Beachten Sie: Lassen Sie die Bits 0 bis 2 auf Ausgabe, da sie das Bank-switching steuern.

Allerdings nützt uns das Datenrichtungsregister allein nichts - denn wir müssen natürlich bestimmen können, welchen Zustand die Ausgangsleitungen und damit die Speicheraufteilung haben soll. Dafür ist Speicherstelle $01 zuständig. Ihr Wert nach dem Einschalten ist 55; d.h. dual »00110111«. Auch hier entspricht jedes Bit einer Port-Leitung. Im Unterschied zum Datenrichtungsregister schaltet der Inhalt des Datenregisters ($00) aber die Leitungen; bzw. teilt uns mit, ob die Leitungen High oder Low sind. Das Lesen der niedrigstwertigen Bits zeigt uns diesmal, wie der Speicher eingeteilt ist. Ein Beschreiben hat (endlich) eine Änderung der Speichereinteilung zu Folge:

Bit	Funktion
0	Basic-ROM =1, RAM = 0
1	Kernal-ROM =1, RAM = 0
2	I/O = 1, Zeichensatz = 0
3	Datenausgabe von Datasette
4	Taste von Datasette gedrückt = 0 nicht gedrückt = 1
5	Motor an = 1, Motor aus = 0
6	unbenutzt = 0
7	unbenutzt = 0

Die Kassettenfunktionen lassen wir, bis auf eine Bemerkung, dezent beiseite: Der Port-Ausgang Bit 5 ist über einen Transistor verstärkt, und kann (wenn kein Kassettenrecorder verwendet wird) direkt einen Motor oder ähnliches steuern.

Wichtig zur Speichereinteilung: Ganz egal, wie Sie den C64 konfiguriert haben, ein Schreibzugriff wird niemals an einen ROM-Bereich geleitet, sondern grundsätzlich an das entsprechende RAM. Damit lassen sich einige Umschaltaktionen vermeiden. Beachten Sie bitte, daß Schreibzugriffe in das Zeichensatz-ROM den I/O-Bereich treffen, und damit seltsame Reaktionen hervorgerufen werden können.

Die wichtigsten Speicherkonfigurationen:

54 - Basic-ROM auf RAM geschaltet
53 - Basic-ROM und Kernal-ROM auf RAM geschaltet
52 - Basic-ROM, Kernal-ROM und Zeichen-ROM (bzw. I/O) auf RAM geschaltet
51 - blendet Zeichensatz ein

Zahlendarstellung in Assembler

Wir verwenden normalerweise bei Berechnungen das sog. Dezimalsystem. Schon in der Schule wurde uns beigebracht,

daß die Zahlen von 0 bis 9 jeweils eine Stelle bedeuten und die nächste Stelle jeweils das Zehnfache der Grundzahl ist. Mit diesem arabischen Zahlensystem können wir quasi im Schlafe umgehen (praktisch, da wir zehn Finger haben). Aber für unseren C64 ist genau dieses Zahlensystem das unpraktischste. Seine möglichen Zahlen sind, wie wir schon gehört haben, Null und Eins. In der Verknüpfung ist damit zwar »10« darstellbar (%000001010), aber »100« hat schon gar nichts mehr zu tun damit (%01100100), und gemein ist dabei, daß eine Speicherzelle 256 und nicht 100 Möglichkeiten hat. Das bedeutet für uns, daß wir erst umrechnen müssen, wenn wir wissen wollen, welchen Gesamtwert zwei aufeinanderfolgende Speicherstellen besitzen. Man nehme also den Inhalt der zweiten Speicherstelle, multipliziere ihn mit 256 und addiere den Inhalt der ersten dazu. Damit haben wir ihn - den Gesamtwert aus beiden Speicherstellen. Diese Methode funktioniert zwar, ist aber langwierig, fehlerträchtig und zudem noch kompliziert. Daher verwendet man zwei andere Systeme, die dem Konzept eines Computers näherkommen. Eines davon haben wir schon kennengelernt: das Binärsystem. Hier wird nur mit »0« und »1« argumentiert. Zur Unterscheidung von unserem Dezimalsystem kennzeichnet man diese Zahlen durch ein vorangestelltes %-Zeichen. Bei vielen Anwendungen ist dieses System durchaus sinnvoll. Denken wir an die Speichereinteilung. Hier entspricht eine 0-/1-Aussage jeweils einer Leitung des Prozessor-Ports. Aber spätestens, wenn wir wieder die zwei Zahlen aus den Speicherstellen verwenden wollen, wird diese Methode fürchterlich undurchsichtig. Es entsteht eine Zahl mit 16 Stellen, von denen jede 0 oder 1 sein kann. Das kann sich kein Mensch merken. Also verwendet man noch eine andere Methode - das Oktalsystem, auch Hexadezimalsystem genannt. Hier kann eine Stelle 16 Werte annehmen: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, und F. Die nächsthöhere hat daher auch die 16fache Wertigkeit. Damit läßt sich in zwei Stellen der Wert eines Bytes ausdrücken (16 x 16=256). Vier Stellen beinhalten also den gesamten Adressierungsbereich des C64 (256 x 256=56536). Hier wurden zwei Fliegen mit einer Klappe geschlagen:

1. Ganz egal welche Zelle des C-64-Speichers beschrieben werden soll, es genügen vier Stellen zur Darstellung (0000 bis FFFF)
2. Um den Gesamtwert aus mehreren Speicherstellen darzustellen, muß nicht mühsam berechnet werden, es genügt, die einzelnen Werte nebeneinanderzustellen.

Damit man dieses Zahlensystem von Dezimal- oder Binärzahlen unterscheiden kann, kennzeichnet man es durch ein vorangestelltes $-Zeichen.

Wie gebe ich Assemblerbefehle ein?

Nachdem Sie jetzt so viel von den Vorzügen des Assemblers, vom Mikroprozessor, von Speicher, Bank-switching und Zahlensystemen gehört haben, sind Sie bestimmt begierig, ihr erstes Assemblerprogramm zu schreiben. Dazu benötigen Sie natürlich ein Werkzeug: den Assembler. Um komfortabel programmieren zu können befindet sich auf der beiliegenden Diskette der Hypra-Ass. Seine Funktionsbeschreibung finden Sie ab Seite 35. Mit Editor und Reassembler bietet er eine große Vielfalt an Verwendungsmöglichkeiten. Damit es aber am Anfang nicht zu kompliziert wird, bedienen wir uns des Assemblers im Monitor. Laden Sie daher den SMON von der beileigenden Diskette mit

```
LOAD"SMON $C000",8,1
```

geben Sie anschließend NEW (<RETURN>) ein und starten Sie mit SYS49152 (<RETURN>). Der SMON begrüßt Sie mit der Anzeige der Register und wartet während der folgenden Erklärungen auf Ihre Eingaben.

Die ersten Befehle – LDA, STA, BRK und RTS

Das menschliche Gehirn hat dem Computer vieles voraus. Dazu gehört beispielsweise, daß ein Mensch allerlei Dinge gleichzeitig tun kann: gehen, sprechen, Musik hören, lächeln, Handbewegungen ausführen usw. Ihr C64 ist dazu nicht imstande. Er erledigt eine Aufgabe nach der anderen. Weil er das aber so schnell macht, hat es für uns den Anschein, es geschehe alles gleichzeitig. Das Maschinenprogramm ist die Kette solcher kleinen Aufgaben. Das erste Glied, das wir daraus kennenlernen wollen, ist der Befehl »LDA«.

Das bedeutet: Lade den Akkumulator. Alle Assembler-Befehlswörter bestehen aus drei Buchstaben (wie dieser hier auch). Es wurde schon erwähnt, daß einem solchen Befehl je eine 8-Bit-Codezahl entspricht. Das ist hier $A9 oder binär 10101001 oder schließlich dezimal 169. Die Codezahl muß in einem Speicherplatz stehen, z.B. in $1500 (entspricht dez. 5376). Assemblerlistings sehen dann folgendermaßen aus:

```
1500 LDA
```

Hier tritt also die Speicherplatznummer mit dem nachfolgenden Befehl anstelle der von Basic gewohnten Zeilennummer.

Es fehlt allerdings noch etwas Entscheidendes: Was soll denn in den Akku geladen werden? Genauso wie es in Basic Befehle gibt, die für sich alleine stehen können (CLR oder LIST), gibt es auch im Assembler solche Befehle. Weitaus häufiger sind allerdings andere, die ein »Argument« erfordern (in Basic z.B. PEEK (100); dabei ist 100 das Argument). In Assembler gibt es zwei Arten von Argumenten:

1. Argumente in 1-Byte-Format
2. Argumente in 2-Byte-Format

Bei einigen Befehlen existieren daher für ein einziges Befehlswort (hier LDA) 1-Byte-, 2-Byte- und 3-Byte-Befehle.

Das Argument von LDA ist also das, was in den Akku geladen werden soll. Laden wir daher eine »1« in den Akku. Dazu geben wir beim SMON ein:

```
A 1500
```

Diese Eingabe bringt SMON (nach <RETURN>) dazu, in den Assemblermodus zu gehen und hat zur Folge, daß »1500« am Bildschirm erscheint und der Cursor mit unwilligem Blinken zur weiteren Eingabe auffordert. Das tun Sie auch:

```
1500 LDA #01
```

nach <RETURN> wird der Befehl am Bildschirm gewandelt

```
1500 A9 01     LDA #01
```

und es erscheint »1502«. Dieser Wert sagt Ihnen die nächste zur Verfügung stehende Speicherstelle und erwartet wieder eine Eingabe:

```
1502 BRK
```

Sie verstehen nicht warum »BRK«? BREAK ist ein 1-Byte-Befehl (ohne Argument) und sagt dem Mikroprozessor, daß ein Programmabschnitt beendet ist und er zu einem festgelegten Programm verzweigen soll (in unserem Fall zum SMON). $1502 ist die nächste freie Speicherstelle, und wenn der Programmzähler nach dem LDA #01 auf 1502 deutet, erwartet der Mikroprozessor dort den nächsten Befehl. Wenn dort Unsinn steht, stürzt der Mikroprozessor im allgemeinen ab – je nachdem, welcher Code hier zufällig enthalten ist. Wir haben ja 256 Möglichkeiten ($00 bis $FF). Im Gegensatz zu Basic, wo man durch den Interpreter die Möglichkeit hat, Zeilennummern zu bauen, muß bei Maschine das Programm eine ununterbrochene Perlenschnur von Befehlen sein. Durch BRK läßt sich dieses Prinzip unterbrechen. Der Mikroprozessor unterbricht seine Arbeit und springt zurück zum Monitor.

Nachdem Sie die Eingabe wieder mit <RETURN> bestätigt haben, steht jetzt am Bildschirm:

```
1500 A9 01      LDA #01
1502 00         BRK
1503
```

Damit sind wir allerdings noch nicht fertig, denn SMON muß noch mitgeteilt werden, daß der Assemblierungsvorgang beendet ist. Geben Sie ein:

```
1503 F
```

danach wiederholt SMON die eingegebenen Befehle und unser kleines Programm ist assembliert.

Beachten Sie dabei, daß SMON alle Eingaben in hexadezimaler Schreibweise erwartet. Bei anderen Monitoren kann dies anders sein. Beim Hypra-Ass ist es sogar möglich, die unterschiedlichen Zahlenformate mit vorangestelltem Kennzeichen zu mischen ($ für hexadezimal, % für Binär). Doch was bedeutet # in unserem Assemblerbefehl? Es gibt viele Arten, den Akku zu laden. Direkt mit einer Zahl - wie hier - aber auch mit dem Inhalt einer Speicherstelle. Man spricht dabei von »Adressierung«. Es gibt eine ganze Menge davon, und jede wird auf eindeutige Art und Weise gekennzeichnet. Wenn wir mit unserem Akku direkt eine Zahl laden, dann ist das die »unmittelbare« Adressierung (immediate) und die kennzeichnet man mit #.

unmittelbare Adressierung (immediate)

Bei dieser Adressierungsart muß im Maschinencode unmittelbar nach dem Befehl der Wert erscheinen, der behandelt wird: Beispielsweise steht für »LDA #01« daher »$A9 $01«; also zuerst der Code für den Befehl, dann der zu ladende Wert.

Das kurze Listing ist übrigens mit auf Diskette und wird im SMON folgendermaßen geladen

```
L"LI.01"
```

danach läßt es sich mit

```
D 1500 1503
```

dissassemblieren. Starten wir einmal unser kleines Programm:

```
G 1500
```

Unmittelbar danach werden die Register angezeigt. Der Programmzähler (PC) steht auf 1503, im Akku (AC) steht 01, alle Flaggen außer der Breakflagge sind Null (die unbenutzte Flagge steht immer auf 1). Jetzt ändern wir das Argument in »LDA #00«. Geben Sie dazu ein:

```
D 1500 1503
```

Danach erscheint Ihr Listing:

```
1500 A9 01      LDA #01
1502 00         BRK
```

Ändern Sie in der Zeile 1500 neben LDA das #01 in #00 und bestätigen Sie diese Änderung mit <RETURN>.

Starten Sie nun wieder mit »G 1500« und sehen Sie sich die Register an: Programmzähler 1503, Akku jetzt 00, aber bei den Flaggen hat sich etwas geändert: Die Zero-Flagge ist auf 1 gesetzt. Wir sehen daran: die Zero-Flag ist so lange Null, bis in einer Operation (in unserem Fall der Akku) das Ergebnis oder ein Ladeprozeß Null ergibt. Ändern Sie jetzt das Programm in »LDA #FF«, oder laden Sie »LI.02«. Starten Sie danach wieder mit »G1500«. Natürlich steht FF im Akku, nur bei den Flaggen ist etwas merkwürdiges passiert: die Vorzeichenflagge steht auf 1. Das bedeutet, im Akku soll eine negative Zahl stehen! Wir wissen aber alle, daß $FF = dez. 255 ist. Es liegt allerdings kein Fehler vor: Immer wenn in einer Zahl das Bit 7 gleich 1 ist, geht zugleich die Vorzeichenflagge auf 1. Die Lösung des Rätsels sind die sog. negativen Binärzahlen. Bei Ihnen gilt eine Zahl als negativ, wenn Bit 7 gesetzt ist und als positiv, wenn Bit 7 gleich 0 ist. Sehen Sie sich dazu auch das Befehlsposter auf Seite 26/27 an.

Der LDA-Befehl beeinflußt die Vorzeichen- und die Zero-Flagge

Der nächste Befehl ist die Umkehrung von LDA und heiß »STA« (STore Accumulator), also lege den Akkumulatorinhal ab. Wie Sie sich denken können, muß auch hier ein Argumen auftauchen: nämlich, wohin der Akkuinhalt abgelegt wird. Wi legen den Akkuinhalt in die erste Bildschirmspalte ($0400) Damit müßte nach dem Programmstart ein Zeichen auf dem Bildschirm erscheinen. Laden Sie dazu das dritte Listing vo Diskette

```
L"LI.03"
```

und sehen Sie es sich mit »D1500150B« an. Mit STA i Zeile1502 lernen wir eine neue Adressierungsart kennen: di »absolute« Adressierung. Man erkennt Sie daran, daß keine Zusätze verwendet werden (STA 0400). Die Adresse 0400, i die der Akku abgelegt wird, ist nicht in einem Byte darstellba sondern wird aufgeteilt in zwei Bytes. Im Speicher steht jetz ab 1502:

```
8D 00 04
```

absolute Adressierung

Bei dieser Adressierungsart erscheint nach dem Code für den Befehl die Speicherposition, in der das zu behandelnde Argument steht. Die Darstellung erfolgt im Low-/High-Byte-Format. z.B. steht für »LDA 0400« der Code »$AD $00 $04« im Speicher.

»8D« ist der Befehlscode für STA, »00« ist das niederwertig Byte (LSB) und »04« das höherwertige Byte (MSB). Es liegt a so ein 3-Byte-Befehl vor und der nächste Befehl darf ab $150 beginnen. Von Basic her wissen wir, daß 1 der Bildschirmco de für »A« ist. Um dieses »A« aber vom Hintergrund abzuhe ben, bestimmen wir, daß es schwarz (Farbe 0) erscheinen so (LDA #00), und schreiben diesen Wert an die entsprechend Position ($D800) ins Farbregister (STA D800). Die nächst freie Speicherposition ist jetzt $150A. Damit unser Programm abgeschlossen ist, steht an dieser Position »BRK«. Es könnt hier auch RTS (ReTurn from Subroutine), also Rückkehr au dem Unterprogramm stehen. Auch damit ist das Programm abgeschlossen. Allerdings springt der Mikroprozessor nich zurück zum SMON, sondern zu der Stelle, von der wir de SMON gestartet haben, nach Basic. Ändern Sie doch mal i 150A den Befehl »BRK« in »RTS« und starten Sie mit G150 Wenn Sie mit dem Cursor nicht zu weit unten waren, sehe Sie jetzt (wie nicht anders zu erwarten, ein schwarzes »A« a Bildschirm, der evtl. erscheinende SYNTAX ERROR stört da bei nicht. Löschen Sie den Bildschirm (<SHIFT CL HOME>) und starten Sie erneut, diesmal aus Basic m SYS5376. Spätestens jetzt haben Sie ihr schwarzes »A« a Bildschirm und das gewohnte READY. Gehen wir zurück zu SMON (mit SYS49152); und sehen wir uns nochmals den B fehl RTS an (mit D150A 150A). Wie deutlich sichtbar ist di ein 1-Byte-Befehl ($60). Auch hier spricht man von ein Adressierungsart, nämlich von »implizit«. Man erkennt sie a Fehlen des Arguments. Die Adresse ist implizit, d.h. im Befe selbst enthalten. Für den Prozessor bedeutet dies: er holt si vom Stapel die oberste Adresse. Diese wurde dort abgeleg als der Prozessor mit SYS aus Basic sprang.

Adressierungsart: implizit

Hier ist mit dem Befehlscode der komplette Befehl beschrieben. D.h. er besteht aus einem Byte, benötigt also kein Argument.

Vier weitere Befehle: LDX, STX, LDY, STY

Die Kombination von LDA mit STA ist vergleichbar mit dem POKE-Befehl aus Basic. Man kann allerdings in Assembler nicht direkt eine Speicherstelle beschreiben, sondern muß den Umweg über ein Register machen. Eines davon haben wir bereits kennengelernt – den Akku. Außer ihm eignen sich die beiden Hilfsregister x und y. Die Befehle für das x-Register lauten: LDX (LoaD X – lade x-Register) und STX (STore X – lege x-Register-Inhalt ab). Für das y-Register ist zuständig: LDY (LoaD Y – lade y-Register) und STY (STore Y – lege y-Register-Inhalt ab). Probieren Sie das doch mal an Listing 4 (L"LI.04") aus. Es läßt sich mit »D1500 1519« disassemblieren, mit G1500 starten und bringt »ABA« auf den Bildschirm. Dabei ist das x-Register dreimal ausgelesen worden, der Akku zweimal und das y-Register einmal. Sie sehen, daß die Registerinhalte durch STA, STX und STY nicht verändert werden.

Ausführungszeiten der Befehle

Wenn Sie jetzt die Tabelle unten betrachten oder auf S. 26/27 aufblättern und sich die Ausführungszeiten (Zyklen) für die einzelnen Befehle ansehen (unmittelbar = imm, absolut = Abs), müßten Sie nachrechnen können, wie lange unser letztes Programm zur Ausführung gebraucht hat (1 Zyklus = ca. 1 Mikrosekunde). Es waren 48 Mikrosekunden (0,000048 Sekunden). Ein vergleichbares Basic-Programm benötigt für ein vergleichbares Programm 0,05 Sekunden, also etwa tausendmal so lange.

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
INX	implizit	1	E8	232	2	N,Z
INY	implizit	1	C8	200	2	N,Z
INC	absolut	3	EE	238	6	N,Z
DEX	implizit	1	CA	202	2	N,Z
DEY	implizit	1	88	136	2	N,Z
DEC	absolut	3	CE	206	6	N,Z
SED	implizit	1	F8	248	2	1 – D
CLD	implizit	1	D8	216	2	0 – D
BNE	relativ	2	D0	208	2	—
					+1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	

Kurzübersicht der Befehle fürs Zählen

Wir zählen: INX, INY, INC, DEX, DEY und DEC

Sie können »zählen« wörtlich nehmen, denn alle diese Befehle haben eines gemeinsam: sie erhöhen oder erniedrigen entweder Register oder Speicherstellen um »1«. Das wird schon beim Ausschreiben der Abkürzungen erkennbar: INX heißt INcrement x-Register, also »erhöhe das x-Register um 1«. Es wird sicherlich einleuchten, daß INY (INcrement y-Register) das gleiche mit dem y-Register macht. Ein wenig diffuser ist INC – INCrement memory (zähle zum Inhalt einer Speicherstelle 1 hinzu). INX und INY enthalten alles, was dem Computer zu sagen ist und sind daher 1-Byte-Befehle mit impliziter Adressierung. Mit INC verhält es sich anders. Hier muß dem Mikroprozessor noch mitgeteilt werden, welche Speicherstelle gemeint ist. Das läßt diesen Befehl zu einem 3-Byte-Befehl werden.

Die Umkehrung dieser Befehle lautet DEX (DEcrement x-Register), DEY (DEcrement y-Register) und DEC (DECrement memory). Da »decrement« »um eins verringern« bedeutet, erniedrigt der Mikroprozessor auch bei Anwendung dieser Befehle jeweils entweder x-Register, y-Register oder eine Speicherstelle um eins. Für die Adressierungsart und Anzahl der Bytes gilt das gleiche wie bei INX, INY und INC. Sehen wir uns dafür das Beispiel auf Diskette an (L"LI.05" und D 15001519).

Wenn Sie das Programm mit G1500 starten (der Bildschirm darf nicht scrollen), erscheint in der linken oberen Ecke »ABA« in schwarzer Schrift. Was ist geschehen? Wir haben den Inhalt des Akku (0 = Farbcode für Schwarz) ins Farbregister geschrieben (ab $D800), dann den Inhalt des x-Registers (1 = Code für den Buchstaben »A«) in die erste Bildschirm-Speicherzelle. Anschließend wurde das x-Register um 1 erhöht (2 = Code für »B«) und dieser Inhalt in die zweite Bildschirmzelle geschrieben. Außerdem mußte dieser Bildschirm-Farbspeicherplatz mit dem Farbcode 0 belegt werden. Durch DEX wurde das x-Register um 1 reduziert, somit wieder ein »A« erzeugt und in die dritte Bildschirmstelle abgelegt.

Es ist Ihnen sicher aufgefallen, daß man auf diese Weise Abläufe mitzählen kann. Soll z.B. ein Vorgang 20mal wiederholt werden, schreibt man ins x-Register (möglich ist auch das y-Register oder eine Speicherstelle) den Anfangswert 0, läßt den Computer eine Arbeit ausführen, erhöht das Register oder die Speicherstelle (mit INX, INY oder INC). Anschließend prüft man, ob dieser Inhalt schon 20 geworden ist usw. Nun sollten wir uns aber grundsätzlich vor Augen halten: Ein Register oder eine Speicherstelle kann nur Werte von 0 bis 255 erhalten. Was passiert also, wenn wir weiterzählen? Für ein Beispiel geben Sie ein:

```
1500 LDX #FF
1502 INX
1503 BRK
1504 F
```

und starten mit G1500. Sie werden sehen, daß 255 + 1 Null ergibt (siehe XR). Allerdings ist die Zero-Flagge gesetzt. Ein Überlauf wird nicht angezeigt (obwohl einer stattfindet). Das gleiche passiert, wenn wir herabzählen:

```
1500 LDY  #01
1502  DEY
1504  F
```

Auch hier ist nach dem Ablauf der Routine die Zero-Flagge gesetzt. Es sei verraten, daß die Befehle INX, DEX, INY, DEY, INC und DEC nur zwei Flaggen beeinflussen: die Zero-Flagge und die Negativ-Flagge. Beachten Sie, alle anderen Flaggen bleiben unverändert.

Befehlswort	Adressierung	Bytezahl	Code HEX	Code DEZ	Dauer in Taktzyklen	Beeinflussung von Flaggen
LDA	unmittelbar	2	A9	169	2	N, Z
	absolut	3	AD	173	4	N, Z
LDX	unmittelbar	2	A2	162	2	N, Z
	absolut	3	AE	174	4	N, Z
LDY	unmittelbar	2	A0	160	2	N, Z
	absolut	3	AC	172	4	N, Z
STA	absolut	3	8D	141	4	keine
STX	absolut	3	8E	142	4	keine
STY	absolut	3	8C	140	4	keine
RTS	implizit	1	60	96	6	keine

Die Ausführungszeiten der ersten Befehle

BRANCH-Befehle

Wir haben inzwischen schon etliche Befehle kennengelernt. Wissen inzwischen auch, daß die meisten davon einige Flaggen beeinflussen. Na und, werden Sie sagen, denn was Sie Erstaunliches damit anfangen können, ist Ihnen noch nicht bekannt. Wenn Sie mit dem jetzigen Wissen von 255 auf 0 herabzählen möchten, bleibt nichts anderes übrig, als zuerst z.B. das x-Register mit $FF zu laden und danach 255 mal DEX zu schreiben. Mal abgesehen davon, daß eine Menge Speicherplatz verbraucht wird, sind Sie eine ganze Weile mit dem Eintippen beschäftigt. Sie haben es sich sicherlich schon gedacht: es gibt eine schnellere und bessere Methode. Um sie kennenzulernen, verwenden wir einige neue Befehle. Der erste davon ist BNE - Branch if Not Equal zero, oder verzweige, wenn ungleich Null. Sie ahnen es sicher: Dieser Befehl hat etwas mit der Zero-Flagge zu tun. Genauer gesagt, es wird zu einer angegebenen Adresse gesprungen, wenn die Zero-Flagge nicht gesetzt ist (= 0). Sehen wir uns dazu Listing 6 von der Diskette an (L"LI.06" und D1500150B). Zunächst werden das x- und das y-Register mit dem Ausgangswert $FF geladen (initialisiert). Mit DEY wird dann das y-Register um 1 herabgezählt (ergibt $FE). Die Zero-Flagge ist das der »0« (Klar, das Register enthält nicht »0«). Daher wird bei der nachfolgenden Überprüfung durch BNE in die danach festgelegte Speicherposition ($F504) verzweigt. Dort steht DEY, worauf das y-Register wieder um 1 erniedrigt wird. Dieses Spiel wiederholt sich nun so lange, bis endlich »0« im y-Register steht. Zugleich geht die Zero-Flagge auf »1«. Damit verzweigt der BNE-Befehl nicht mehr - der nächste Befehl (DEX) wird durchgeführt. Da allerdings jetzt das x-Register auf $FE steht, wird mit BNE nach 1502 verzweigt und das y-Register wieder mit $FF geladen. Die erste Schleife läuft wieder ab und ....

Wir haben hier zwei Schleifen ineinander verschachtelt. Die äußere davon wird 255mal durchlaufen, die innere 65025mal. Zur Verdeutlichung programmieren wir einmal diese Schleife in Basic:

```
100 FOR I = 255 TO 0 STEP-1
110 FOR J = 255 TO 0 STEP-1
120 NEXTJ
130 NEXTI
```

Diese Befehlsfolge bewirkt dasselbe wie unsere Assemblerroutine - eine Verzögerung im Programmablauf. Nur ist Basic ungleich langsamer. Starten Sie unsere Maschinenroutine mit G1500. Sie werden eine merkliche Verzögerung feststellen.

Noch längere Verzögerungen erhalten Sie, wenn Sie mehrere Schleifen ineinanderschachteln. Dabei verwenden Sie den DEC-Befehl.

Wozu Sie solche Verzögerungen brauchen, ist eigentlich klar: Wenn Sie z.B. einen Text vom Bildschirm lesen wollen, bevor das Programm weiterläuft, oder mit Peripherie arbeiten, die langsamer als der Computer ist, oder ... Allerdings sollte man erwähnen, daß es elegantere Methoden zur Verzögerung gibt, als ein Lahmlegen des Computers, doch dazu kommen wir etwas später.

BEQ ist die Umkehrung des BRANCH-Befehls BNE. Bei BEQ wird verzweigt, wenn die Zero-Flagge gleich »1« ist.

Anhand der Registeranzeige im SMON kennen Sie noch andere Flaggen. Die Carry- (C), die Negativ- (N) und die Überlauf-Flagge (V). Behandeln wir als nächstes die Carry-Flagge; für sie gibt es zwei Verzweigungs-(BRANCH-)Befehle:

1. BCC (Branch Carry Clear - verzweige, wenn Carry gelöscht) und
2. BCS (Branch Carry Set - verzweige, wenn Carry gesetzt).

Befehlsart	Adressierung	Bytezahl	Code hex	Code dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
ADC	unmittelbar	2	69	105	2	N,V,Z,C
	absolut	3	6D	109	4	
CLC	implizit	1	18	24	2	0→C
SBC	unmittelbar	2	E9	233	2	N,V,Z,C
	absolut	3	ED	237	4	
SEC	implizit	1	38	56	2	1→C
BEQ	relativ	2	F0		2	keine Änderung
BCC	relativ	2	90		2	keine Änderung
BCS	relativ	2	B0		2	keine Änderung
BMI	relativ	2	30		2	keine Änderung
BPL	relativ	2	10		2	keine Änderung
BVC	relativ	2	50		2	keine Änderung
BVS	relativ	2	70		2	keine Änderung
						+1 bei Verzweigung
						+2 bei Überschreiten einer Seitengrenze

Die Arithmetik auf einen Blick

Zusätzlich besteht die Möglichkeit, diese Flagge quasi von Hand zu setzen oder zu löschen:

1. SEC (SEt Carry - setze die Carry-Flagge)
2. CLC (CLear Carry - lösche die Carry-Flagge)

To carry heißt »tragen«. Hier stellt sich die Frage, was wird eigentlich getragen? Das zeigt sich am besten in einem Beispiel, in dem wir mit Binärzahlen 128 und 130 addieren:

```
      128      10000000
+     130      10000010
-------------------------
      258  (1) 00000010
```

Das Ergebnis ist mit 258 zwar richtig, aber es paßt einfach nicht mehr in eine 8-Bit-Darstellung und damit auch nicht mehr in eine Speicherstelle. Ein Bit wurde überTRAGEN in ein extra dafür vorgesehenes Plätzchen - das Carry-Bit, auch Carry-Flagge genannt. Jedesmal, wenn so ein Übertrag bei einer Rechenoperation stattfindet, zeigt die Carry-Flagge eine »1«.

Je nach der Art Ihres Programmiervorhabens können Sie dieses Carry-Bit weiterverarbeiten. Es gibt auch Aufgaben, bei denen man es einfach vernachlässigen darf, oder solche, bei denen es in einer Rechnung weiterverwendet wird. Schließlich kann es uns noch anzeigen, wenn das größte Rechenergebnis %1111 1111 (255) sein darf.

Die Negativ-Flagge haben wir schon mal gestreift. Sie ist immer zugleich mit Bit 7 gesetzt und zeigt negative Zahlen an, wenn mit Zweierkomplementzahlen gearbeitet wird (pos. = 0 bis 127, neg. = 128 bis 255). Verzweigungsbefehle für diese Flagge sind:

1. BMI (Branch if MInus - springe, wenn Ergebnis minus, Negativ-Flagge = 1) und
2. BPL (Branch if PLus - springe, wenn Ergebnis Plus, Negativ-Flagge = 0).

Bleibt nur eine Flagge für BRANCH-Befehle übrig: die Overflow-Flag (V). Sie zeigt uns bei Addition zweier positiver Zahlen im Zweierkomplement ein falsches Ergebnis:

```
      64       01000000
+     66       01000010
-------------------------
     -126      10000010
```

Das ist offensichtlich falsch. Bei der Addition ist durch Zusammenzählen der Bits 6 auch Bit 7 gesetzt worden. Da wir es aber mit der Zweierkomplement-Darstellung zu tun haben, wird die Überlauf-Flagge gesetzt. Leider ist die Sache nicht ganz so einfach, daß sie immer gesetzt wird, wenn von Bit 6 nach Bit 7 ein Übertrag stattfindet. Prinzipiell wird sie sich nur in folgenden zwei Fällen auf »1« ändern:

1. Es findet ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag (Carry).
2. Es findet kein interner Übertrag von Bit 6 nach Bit 7 statt, aber ein äußerer Übertrag.

Merken kann man sich das am besten folgendermaßen: Immer dann, wenn quasi aus Versehen das Vorzeichenbit 7 verändert wurde, wird die V-Flagge auf 1 gesetzt. Das erfordert natürlich, daß man sich bei allen Operationen vorher überlegen muß, welche Fehler durch versehentliches Vorzeichenändern passieren können. Die Verzweigungsbefehle für die Overflow-Flagge sind:
1. BVS (Branch if oVerflow Set – springe, wenn V gesetzt) und
2. BVC (Branch if oVerflow Clear – springe, wenn V nicht gesetzt)

Arithmetik in Assembler – ADC

Der erste arithmetische Befehl, den wir kennenlernen, ist ADC (ADd with Carry – addiere mit Carry). Dazu addieren wir zuerst zwei Zahlen, die so klein sind, daß kein Überlauf stattfindet. Laden und betrachten Sie dazu Listing 7 (L"LI.07" und D1200 1209). Der Beginn der Befehlsfolge ist CLC, also lösche die Carry-Flagge. Warum? Nun, wir wissen nicht wie sie momentan aussieht und es gibt eine Menge Vorgänge in einem Assemblerprogramm, die diese Flagge beeinflussen. Weil jedoch ADC auch das Carry-Bit mitaddiert, sollte man dafür sorgen, daß es vor jeder Addition gelöscht ist. Dazu haben wir schon weiter oben den Befehl CLC kennengelernt. In unserem kleinen Programm wird der Akku mit $0C (12) geladen und mit ADC $07 addiert. Das Ergebnis wird in Speicherstelle $1244 abgelegt. Starten Sie doch mal das kleine Programm (G 1200), obwohl Sie diese Rechnung sicherlich schneller im Kopf berechnen könnten. »M12441245« zeigt das Ergebnis. Wie nicht anders zu erwarten, steht $13 (19) in Speicherstelle $1244. Es ist bei diesem Programm ziemlich mühsam, andere Werte einzusetzen, da die Werte mit unmittelbarer (immediate) Adressierung geladen und addiert werden. Beide Befehle können aber auch absolut adressiert werden. Dazu laden Sie Listing 8 (L"LI.08") und betrachten sich die Befehle (D1200120B). Die Behandlung wird bedeutend einfacher, wenn Sie sich mit M12401244 die Speicherstellen ansehen. Momentan stehen hier noch willkürliche Werte, aber Sie können durch Überschreiben von $1240 und $1242 zwei Zahlen vorgeben, die dann (nach G1200 und M12401244) in $1244 das Ergebnis zeigen. Was wir bis jetzt gemacht haben, ist die Addition zweier 8-Bit-Zahlen. Weitaus häufiger werden in der Praxis 16-Bit-Zahlen addiert. Laden und betrachten Sie dazu Listing 9 (L"LI.09" und D12001214). Für dieses Beispiel sind ab $1240 schon die Zahlen vorgegeben. Wie wir schon beim Programm-Counter gehört haben, teilen wir dazu die 16-Bit-Zahl in zwei 8-Bit-Zahlen (LSB und MSB). Bei uns stehen diese Zahlen schon in den Speicherstellen $1240 bis $1243. Es sind $0880 (2176) und $03F1 (1009). Fällt Ihnen auf, daß jeweils das niederwertige Byte vor dem höherwertigen steht? $0880 steht als »80 08« und $03F1 als »F1 03« im Speicher. Obwohl wir es natürlich auch anders programmieren könnten, ist diese Schreibweise äußerst sinnvoll, da bestimmte Sprungbefehle, von denen wir noch hören werden, dieses Format benötigen. Ein bißchen ausgewählt sind unsere Zahlen allerdings. Es wurde darauf geachtet, daß im höherwertigen Byte kein Überlauf vorkommen kann. Wenn Sie mit G1200 starten, werden zuerst die niederwertigen Bytes addiert: $80+$F1=$71 und unser Carry ist gesetzt. Danach kommen die höherwertigen Bytes an die Reihe: $08+$03+Carry=$0C. Damit steht das Ergebnis $0C71 (3185) ab der Speicherstelle $1244. Natürlich im Format »71 0C«. Falls Sie dem Programm nicht trauen, rechnen Sie doch einfach nach.

SBC – Subtrahieren

Sie werden es nicht glauben, subtrahieren oder addieren ist für den Mikroprozessor Jacke wie Weste; er hat nur einen Arbeitsschritt mehr zu erledigen:

Nehmen wir an, wir subtrahieren von der Zahl 100 das Argument 97. Das Ergebnis kennen Sie: »3«. Aber mit einem Trick kann man diese Rechnung auch durch Addition ausdrücken:

Nehmen wir zuerst das Argument (97 = %01100001). Danach bilden wir das Komplement davon. Das heißt aus jeder »0« wird eine »1« und umgekehrt. Das Ergebnis ist %10011110. Dazu addieren wir 100 (%01100100). Das Ergebnis ist jetzt 2 (%00000010), aber gemeinerweise nicht 3, wie es sein sollte. Wir müssen also 1 zusätzlich addieren. Zur Verdeutlichung die Zahlenfolge untereinander:

```
(0)10100001 = $61 Argument
------------------
(0)10011110 = $9E Komplement des Arguments
(0)01100100 = $64 + Zahl 100
------------------
(1)00000010 = $02 Ergebnis + Übertrag 9. Stelle
(0)00000001 = $03 Offset (1) dazu
==================
(0)00000011 richtiges Ergebnis
```

Eine genaue Erklärung, warum die »1« zum Ergebnis addiert werden muß, würde zu weit führen. Nehmen Sie es als gegeben.

LDA	absolut,X	3	BD	189	4	N,Z
	0-page-abs,X	2	B5	181	4	N,Z
	absolut,Y	3	B9	185	4	N,Z
LDX	absolut,Y	3	BE	190	4	N,Z
	0-page-abs,Y	2	B6	182	4	N,Z
LDY	absolut,X	3	BC	188	4	N,Z
	0-page-abs,X	2	B4	180	4	N,Z
STA	absolut,X	3	9D	157	5	/
	absolut,Y	3	99	153	5	/
	0-page-abs,X	2	95	149	4	/
STX	0-page-abs,Y	2	96	150	4	/
STY	0-page-abs,X	2	94	148	4	/
INC	absolut,X	3	FE	254	7	N,Z
	0-page-abs,X	2	F6	246	6	N,Z
DEC	absolut,X	3	DE	222	7	N,Z
	0-page-abs,X	2	D6	214	6	N,Z
ADC	absolut,X	3	7D	125	4	N,V,Z,C
	absolut,Y	3	79	121	4	N,V,Z,C
	0-page-abs,X	2	75	117	4	N,V,Z,C
SBC	absolut,X	3	FD	253	4	N,V,Z,C
	absolut,Y	3	F9	249	4	N,V,Z,C
	0-page-abs,X	2	F5	245	4	N,V,Z,C
CMP	absolut,X	3	DD	221	4	N,Z,C
	absolut,Y	3	D9	217	4	N,Z,C
	0-page-abs,X	2	D5	213	4	N,Z,C
BIT	absolut	3	2C	44	4	N,V,Z
	0-page-abs.	2	24	36	3	N,V,Z
CLV	implizit	1	B8	184	2	V
NOP	implizit	1	EA	234	2	/
TAX	implizit	1	AA	170	2	N,Z
TAY	implizit	1	A8	168	2	N,Z
TXA	implizit	1	8A	138	2	N,Z
TYA	implizit	1	98	152	2	N,Z
JMP	absolut	3	4C	76	3	/
	indirekt	3	6C	108	5	/
JSR	absolut	3	20	32	6	/

Neue Adressierungsarten: Zeropage

Jetzt wird auch klar, warum wir dieses kleine Rechenbeispiel durchgearbeitet haben: Erinnern Sie sich noch an CLC und SEC? Wenn mit SEC die Carry-Flagge gesetzt ist, addiert der Mikroprozessor zusätzlich noch die »1«. Und da der

Mikroprozessor die Subtraktion intern genauso ausführt wie oben besprochen, müssen wir davor das Carry mit SEC setzen. Kommt ein Übertrag bei der Rechnung vor (z.B. 29-60= -31), wird das Carry gelöscht. Diese Befehlsfolge läßt sich auch in Assembler simulieren:

```
LDA     #  64 ;
Lädt Argument
EOR     #  FF ;
 Invertiert die Bits
SEC     ;
setzt Carry-Flagge
ADC     #  61 ;
addiert $64 + 1
BRK     ;
führt zurück in den Monitor
```

Stören Sie sich nicht an EOR, dieser Befehl macht eine Exclusiv-ODER-Verknüpfung mit $FF (%11111111). Damit werden alle Bits invertiert. Zur Erklärung kommen wir später. Interessanter ist, daß wir zuerst unsere zu subtrahierende Zahl in den Akku laden müssen. Da dies eine ungewöhnliche Rechenfolge ist, gibt es einen eigenen Befehl – SBC (SuBtract with Carry – subtrahiere mit der Carry-Flagge).

Unsere Rechnung lautet damit:

```
SEC
LDA #$64
SBC #$61
```

Nach der Ausführung dieser Befehlsfolge steht $03 im Akku und Carry bleibt gesetzt. Zum ausführlichen Test laden Sie Listing 10 von Diskette (L"LI.10"). Betrachten Sie es mit »D1200120C«. Wenn Sie die Speicherstellen $1240 mit der Zahl und $1242 mit dem Subtrahenden überschreiben, erscheint das Ergebnis nach G1200 in $1244. Sie werden feststellen, daß SBC die Negativ-, Overflow-, Zero- und natürlich die Carry-Flagge beeinflußt.

Wenn Sie die Erklärung von SBC aufmerksam verfolgt haben, erklärt es sich von selbst, daß bei einer gelöschten Carry-Flagge das Ergebnis minus eins im Akku steht. Diese Eigenschaft nützt uns bei der 16-Bit-Subtraktion einiges. Laden und betrachten Sie dazu Listing 11 (L"LI.11" und D12001215). In Speicherstelle $1240 gehört das Low-Byte in $1241 das High-Byte der 16-Bit-Zahl, von der die 16-Bit-Zahl (Low-Byte in $1243, High-Byte in $1244) subtrahiert werden soll. Das Ergebnis steht nach G1200 in $1244 (LSB) und $1245 (MSB). Bei dieser Rechnung wird ähnlich der Addition zuerst das Low-Byte behandelt, danach das High-Byte.

Vergleichen – CMP, CPX, CPY

Eigentlich sind diese Befehle nichts anderes, als die oben beschriebene Subtraktion - mit einer Ausnahme, das Rechenergebnis wird nicht festgehalten, es werden lediglich die Flaggen beeinflußt. Und für uns sehr wichtig, diese Vergleichsfunktionen lassen sich auch beim x- und y-Register verwenden. CMP (CoMPare to Accumulator – Vergleiche mit dem Akku-Inhalt) ist die entsprechende Funktion für den Akku und beeinflußt die Negativ-, Zero- und Carry-Flagge. Zusammen mit den Branch-Befehlen, lassen sich mit kurzen Programmen die kompliziertesten Abfragen aufbauen.

Zur Verdeutlichung noch einmal die Funktion der Branch-Befehle. Listing 12 wiederholt die Funktion von BEQ (L"LI.12" und D2000200B). Hier wird ein Wert aus Speicherstelle $200B in den Akku geladen und wenn er Null ist, zur Position $200A verzweigt (BRK). Ist der Inhalt ungleich Null, wird $00 in den Akku geladen und wieder in $200B geschrieben. Diese Routine ergibt zunächst keinen Sinn. Darum schreiben wir sie um (A 2000):

```
2000 LDA C6
2002 BEQ 2000
2004 LDA #00
2006 STA C6
2008 BRK
2009 F
```

Danach starten wir mit G2000. Und siehe da, der Compute hängt. Doch das ist nur vermeintlich der Fall. Drücken Si doch mal irgendeine Taste (außer SHIFT): Sie erhalten wiede die Registeranzeige des SMON.

Was ist passiert? Die Speicherzelle $C6 enthält die Infor mation, ob eine Taste der Tastatur gedrückt wurde. Sie errin nern sich: Beim Ablauf von Basic-Programmen merkt sich der C64 bis zu zehn Tastentips. Dazu wird jede 60stel Sekun de die Tastatur abgefragt (im Interrupt). Die Information, wi viele Tasten gedrückt wurden, steht in $C6 (198). Kein Tasten druck = 0, daher überprüfen wir mit BEQ diese Speicherstel le. Ist das Ereignis Tastendruck eingetreten ($C6 =1 oder grö ßer), muß allerdings mitgeteilt werden, daß wir dies erkannt haben. Daher setzen wir $C6 wieder auf Null. Eine Tastatur Warteschleife ist sinnvoller als eine Warteschleife rein au Zeitbasis, da der User selbst entscheiden kann, wann es in Ablauf weitergeht. Doch es ist noch etwas Bemerkenswertes passiert. Sehen Sie sich die Zeile 2000 an. Hier steht jetzt

```
2000 A5 C6     LDA C6
```

Unser Akku hat aus der Speicherstelle $C6 den Wert gela den. Nur ist der Befehl »LDA C6« kein 3-Byte-Befehl wie »LDA 3000«, sondern besteht aus den Bytes »A5 C6«. »C6« ist, wie unschwer zu erkennen ist, die Speicherstelle, aus der gela den werden soll, »A5« ist der Befehlscode. Diese Adressie rungsart nennt man Zero-Page (oder in unserem Poste S.26/27 »0Page«). Sie funktioniert nur auf den ersten 256 By tes (0 bis 255) und, Sie haben es richtig erkannt, hat ihren Na men von der Bezeichnung dieses Bereichs - Zero-Page.

Adressierungsart Zero-Page

Bezieht sich auf die ersten 256 Byte (Zero-Page) des Speicherbereichs. Da das High-Byte wegfällt entsteht ein 2-Byte-Befehl. »LDA C6« wird zu »$A5 $C6«.

Zurück zu unserem Compare-Befehl:

Das Betriebssystem speichert natürlich nicht nur die An zahl der Tastenimpulse, sondern auch welche Tasten ge drückt wurden (im ASCII-Code). Der dafür zuständige Be reich (Tastaturpuffer) reicht von dezimal 631 bis 640 ($0277 bis $0280). Also mal angenommen, wir laden unmittelbar nachdem ein Tastendruck aufgetreten ist, den Wert aus Spei cherstelle 631 den ASCII-Code und vergleichen ihn mit dem Wert einer von uns gewünschten Taste, dann lassen sich im Programm schon einige Entscheidungen treffen:

```
A 2008
2008 LDA 0277
200B SEC
200C CMP #0D
200E BNE 2000
2010 BRK
2011 F
```

In Zeile 2008 laden wir das Hauptregister mit dem Wert aus der ersten Stelle des Tastaturpuffers. Danach setzen wir erstmal die Carry-Flagge. Anschließend vergleichen wir mit dem Wert $0D. Das ist dezimal 13, der Code für die RETURN-Taste. Hat unser Akku einen anderen Wert, verzweigen wir zurück zur Abfrage in Zeile 200 (ist eine Taste gedrückt?). Ansonsten führt der Programmablauf weiter. Starten Sie mal mit G2000. Nur <RETURN> bringt Sie zurück in den Monitor. Die ASCII-Codes finden Sie übrigens im Handbuch Ihres C64.

Dieselbe Wirkung läßt sich auch mit den beiden anderen Compare-Befehlen erreichen:

1. CPX (ComPare to register X - vergleiche mit x-Register) und
2. CPY (ComPare to register Y - vergleiche mit y-Register)

Auch hier muß vor dem Vergleich die Carry-Flagge gesetzt sein (SEC).

Relative Adressierung

Mit den Verzweigungsbefehlen BNE, BEQ, BPL, BMI, BVS, BCC und BCS haben wir sie zwar schon kennengelernt, aber noch nicht erläutert. Laden und betrachten Sie daher nochmal Listing 12 von Diskette (L"LI.12" und D20002000B). Fällt Ihnen in Zeile 2003 auf, daß zwar »BEQ 200A« steht, im Code links daneben aber »F0 05«. Wenn wir »F0« als Befehlscode annehmen (was er auch ist), dann ist »05« aber zunächst nicht mit »200A« in Verbindung zu bringen; normal müßte doch »0A 20« neben dem Befehlscode stehen. Daß dem nicht so ist, liegt an der Benutzerfreundlichkeit des SMON. Er rechnet die absolute Adresse $200A in die relative ($05) um. Wir haben nur 2 Byte zur Verfügung. Eines für den Befehlscode und ein zweites für die relative Adressierung. Wenn wir dieses zweite Byte als Wert für die Abweichung zur momentanen Position des Programm-Counters (Offset) bezeichnen, wird seine Funktion schon deutlicher. Anhand dieses einen Bytes sehen wir auch, daß die Verzweigungen nicht allzuweit zur momentanen Position geschehen können - normalerweise 256 Speicherpositionen. Aber das stimmt nicht ganz. Auch hier ist klar, warum nicht: Ein Verzweigungsbefehl könnte sonst nur in eine Richtung erfolgen. Daß dem nicht so ist, ersehen wir aus Listing 13 (L"LI.13" und D10001006). Hier haben wir den BNE-Befehl in Zeile 1003; er verzweigt laut SMON auf die Speicherposition $1002. Beim Befehlscode steht »D0 FD«, D0 für den Befehlscode und »FD« für den Offset.

Ist Ihnen etwas aufgefallen? Ein relativer Sprung nach vorne im Speicher wird mit einer positiven 1-Byte-Zahl markiert, ein Sprung nach hinten mit einer negativen. Zur Erinnerung: 1-Byte-Zahlen sind negativ, wenn Bit 7 gesetzt, also die Zahl größer als 127 ist. Ist Bit 7 gelöscht, wird die Zahl positiv angenommen. Jetzt verstehen Sie auch den Sinn von Zeile 2003: Springe $05 Positionen nach vorne im Speicher, wenn das Ereignis (BEQ = Zero-Flagge gesetzt) eintritt. Da der Programm-Counter immer auf den Beginn des nächsten Befehls deutet, springe nach $2005+$05=$200A. Um die relative Sprungadresse für Zeile 1003 zu berechnen, müssen wir zuerst die negative Zahl berechnen: $00-$FD=$03, wir müssen quasi über Eck rechnen. Damit ergibt sich für diese Operation ein Sprung nach $1005-$03=$1002. Sowohl der SMON als auch der Hypra-Ass nimmt Ihnen die Umrechnung ab.

Achtung: Bei der relativen Adressierung kann der Mikroprozessor maximal 127 Schritte nach vorne verzweigen. Nach rückwärts sind es 126 Schritte; 128 minus Befehlslänge des Branch-Befehls (=2).

Indizierte Adressierung

Indizieren heißt, etwas mit einem Index (Zeichen oder Nummer) zu versehen. Sie haben bestimmt schon mal ein Jahres-Inhaltsverzeichnis (z.B von unseren Stammheften) gesehen. Damit ist Ihnen auch schon eine Art der Indizierung in die Finger geraten. Wenn Sie einen Artikel gesucht und gefunden haben, steht daneben die Ausgabe (z.B. 1/91) und die Seitenzahl (z.B. S.32). Mit anderen Worten: Ihr gesuchter Artikel ist über die Ausgabe 1/91 mit der Seitenzahl 32 indiziert. Anhand dieser Indizierung nehmen Sie das Heft 1/91 in die Hand und schlagen Seite 32 auf. Dort befindet sich der gesuchte Artikel; und zwar dieser und kein anderer. So ähnlich können wir uns auch die Funktion der indizierten Adressierung vorstellen. Nehmen wir als Beispiel: LDA $1500,X. Man

Fortsetzung S. 21

spricht hier von einer absolut-X-indizierten Indizierung (bei unserem Poster »Abs,X«).

Das Assembler-Wort LDA haben wir schon oft gehört. Bei unserem Beispiel soll der Akku den Wert aus der Speicherstelle holen, die sich durch $1500 plus dem Inhalt des x-Registers ergibt. Steht im x-Register also zum Zeitpunkt des Befehlsaufrufs »$05«, dann wird der Inhalt aus Speicherstelle $1500+$05, also aus $1505 geladen. Da das x-Register Werte zwischen 0 und 255 annehmen kann, können wir allein durch Änderung des x-Registers einen Wert von $1500 bis $15FF indizieren - bei unserem Beispiel in den Akku übernehmen. Mit dieser Adressierung lassen sich plötzlich fantastische Dinge machen: Tabellen vergleichen oder in andere Speicherbereiche schieben usw. Geben Sie dazu im SMON ein:

```
A2000
2000 LDX #00
2002 LDA A09E,X
2005 STA 0400,X
2008 DEX
2009 BNE 2002
200A BRK
200B F
```

und starten Sie mit G2000. Falls Ihr Bildschirm nicht gescrollt ist, sehen Sie am oberen Bildschirmrand 6½ Zeilen Zeichen. Mit <CBM SHIFT> auf Kleinschreibung umgeschaltet, erkennen Sie etliche Befehlswörter aus Basic. Der letzte Buchstabe ist jeweils invertiert. Was haben wir angestellt? Gehen wir einmal den Programmweg durch: In Zeile 2000 wird das x-Register mit $00 geladen und in Zeile 2002 des Akku aus Speicherstelle $A09E+$00 (x-Register) geladen. Was Sie vielleicht nicht wußten: Ab dieser Speicherposition befindet sich im Interpreter eine Liste der Basic-Befehlswörter - den ersten Buchstaben davon haben wir gerade in den Akku geladen. Zeile 2005 bringt den Akku dazu, seinen Inhalt in die Speicherstelle $0400+$00 (x-Register) abzulegen; und hier befindet sich natürlich der Bildschirmspeicher. In Zeile 2008 passiert etwas mit dem x-Register - es wird um »1« verringert. Damit enthält es nicht nicht mehr $00, sondern »$FF«. Würde jetzt die Negativ-Flagge (oder Carry, wenn vorher gesetzt) überprüft, wäre der ganze Vorgang bereits beendet. Aber das geschieht natürlich nicht:

Wir haben bestimmt, daß die Zero-Flagge überprüft wird, und zwar, ob Sie nicht gesetzt ist (BNE). Das kann sie nicht sein, da $FF im x-Register steht, also verzweigt der Mikroprozessor an die Position $2002. Dort lädt er diesmal den Wert aus Speicherstelle $A09E+$FF (=$A19D) in seinen Akku, speichert ihn in Speicherstelle $0400+$FF (=$04FF). Dann verringert er den Wert des x-Registers, überprüft, ob er endlich $00 ist - nein, also zurück zu $2002 und alles solange wiederholt, bis das x-Register gleich $00 ist. Danach erfolgt das wohlverdiente BRK.

Wir ersehen aus dieser Routine, daß auch noch andere Befehle (im Beispiel STA) x-indiziert adressiert werden können. Selbst das y-Register kann »absolut x-indiziert« geladen werden (LDY $$$$,X). Aber Achtung, ein »Abs,X«-speichern dieses Registers ist nicht möglich. Unser obiges Beispiel läßt sich auch umschreiben: man kann auch absolut y-indizieren (LDA $$$$,Y und STA $$$$,Y). Hier gilt die gleiche Besonderheit: Das x-Register läßt sich zwar absolut y-indiziert laden aber so nicht speichern. Und da wir gerade bei Besonderheiten sind: INC und DEC (Erhöhen bzw. Erniedrigen einer Speicherstelle) läßt zwar diese Indizierungsart zu, aber nur mit dem x-Register (s.a. Poster S.26/27).

Unterprogramme – JSR, RTS

Bei unserem Beispiel haben wir zwar ziemlich viele Befehlswörter auf den Bildschirm gebracht. Aber es ist ziemlich mühsam, die einzelnen voneinander zu unterscheiden. Daher verschieben wir im nächsten Beispiel nicht eine komplette 256Byte lange Seite (Page), sondern nur den Bereich in der Länge des ersten Wortes.

```
2000 LDY #00
2002 LDA #0D
2004 JSR FFD2
2007 LDA A09E,Y
200A JSR FFD2
200D SEC
200E INY
200F CPY #03
2011 BNE 2007
2013 BRK
2014 F
```

Diesmal haben wir ein paar (noch) unverständliche Befehle mitverwendet. Bis Zeile 2002 (LDA #0D) kennen wir uns noch aus, aber »JSR FFD2« sollte näher erläutert werden. Mit JSR (Jump SubRoutine - springe in ein Unterprogramm) teilen wir dem Mikroprozessor mit: Merke dir an welcher Stelle du dich gerade befindest und springe an die im Argument angegebene Speicherposition. Der Mikroprozessor legt demzufolge die Position des Programm-Counters auf seinen Stack und ändert den Eintrag im Programm-Counter, in unserem Beispiel auf die Position $FFD2. Danach deutet dieser nicht mehr auf $200D, sondern auf die neue Adresse. Da nach dieser Anweisung der Befehl zu Ende ist, wird das Programm ab Speicherposition $FFD2 fortgesetzt.

Dieser Befehl versetzt uns also in die Lage, aus unserem Programm heraus an eine beliebige andere Speicherposition zu springen; und er macht noch mehr: er merkt sich die Stelle, von der er aus gesprungen ist. In unserem Fall steht ab $FFD2 ein Teil der Betriebsystem-Routinen - die Character-out-Routine (Buchstabenausgabe). Sie wertet das im Akku befindliche Zeichen nach seinem ASCII-Code aus. Die Ausgabe beginnt ab Cursor-Position.

Wir haben $0D (13) geladen. Dieser Wert ist die ASCII- Kodierung für einen Zeilenvorschub. Das heißt nach dem ersten Aufruf dieser Routine befinden wir uns für die Ausgabe des nächsten Zeichens am Zeilenanfang, eine Zeile tiefer. Dieses erste $0D ist deshalb wichtig, da sonst die folgenden Ausgaben unmittelbar nach oben erwähnter Null beginnen. Unser Mikroprozessor ist also nach $FFD2 gesprungen und führt dort die Ausgaberoutine aus. Aber wie kommt er wieder zurück?

Wir haben früher schon einmal von dem dafür zuständigen Befehl gehört - RTS (ReTurn from Subroutine, kehre vom Unterprogramm zurück). Dieser Befehl muß hinter dem letzten Befehl des angesprungenen Programmteils stehen. In der Character-out-Routine ist das der Fall. Ein RTS ändert nur den Programm-Counter, indem sein Befehlscode dem Mikroprozessor mitteilt: nimm die beiden obersten Werte vom Stack und schiebe sie in den PC. Damit ist dein Befehl beendet. Falls wir nun zwischenzeitlich nichts am Stack geändert haben, springt die Befehlsausführung zu dem Programmschritt zurück, von dem aus die Unterroutine aufgerufen wurde.

Allein an dieser Beschreibung haben Sie schon gemerkt, daß man die Rücksprungadresse manipulieren kann, doch Vorsicht: willkürliches Ändern hat in den meisten Fällen den Absturz der CPU zur Folge.

Sehen wir uns unsere Routine weiter an:

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
AND	absolut	3	2D	45	4	N, Z
	0-page-abs	2	25	37	3	N, Z
	unmittelbar	2	29	41	2	N, Z
	abs.-X-indiz.	3	3D	61	4*	N, Z
	abs.-Y-indiz.	3	39	57	4*	N, Z
	indiz.-indir.	2	21	33	6	N, Z
	indir.-indiz.	2	31	49	5*	N, Z
	0-page-X-indiz	2	35	53	4	N, Z
ORA	absolut	3	0D	13	4	N, Z
	0-page-abs.	2	05	05	3	N, Z
	unmittelbar	2	09	09	2	N, Z
	abs.-X-indiz.	3	1D	29	4*	N, Z
	abs.-Y-indiz.	3	19	25	4*	N, Z
	indiz.-indir.	2	01	01	6	N, Z
	indir.-indiz.	2	11	17	5*	N, Z
	0-page-X-indiz	2	15	21	4	N, Z
EOR	absolut	3	4D	77	4	N, Z
	0-page abs.	2	45	69	3	N, Z
	unmittelbar	2	49	73	2	N, Z
	abs.-X-indiz.	3	5D	93	4*	N, Z
	abs.-Y-indiz.	3	59	89	4*	N, Z
	indiz.-indir.	2	41	65	6	N, Z
	indir.-indiz.	2	51	81	5*	N, Z
	0-page-X-indiz	2	55	85	4	N, Z
ASL	»Akkumulator«	1	0A	10	2	N, Z, C
	absolut	3	0E	14	6	N, Z, C
	0-page-abs.	2	06	06	5	N, Z, C
	abs.-X-indiz.	3	1E	30	7	N, Z, C
	0-page-X-indiz	2	16	22	6	N, Z, C

* bedeutet: Bei seitenüberschreitenden Indizierungen muß noch ein Taktzyklus dazugerechnet werden.

Die logischen Operationen AND, ORA und EOR auf einen Blick. ASL finden Sie auf Seite 24.

Wir befinden uns mittlerweile am Anfang der Zeile nach G2000. Falls dies zufällig die letzte Bildschirmzeile war, ha die Character-out-Routine dafür gesorgt, daß der Bildschirm gescrollt hat. In Zeile 2000 sorgte der Befehl LDY #00 für ein Laden des Werts $00 in das y-Register. In Zeile 2007 wird $A09E, y geladen und danach die Character-out-Routine wie der aufgerufen. Kurz eine Erklärung zu dem »e« ($45), das sich gerade im Akku befindet. Wir haben im ersten Beispiel dieses »e« genommen und einfach in den Bildschirmspeicher übertragen. Der Dank dafür war, daß wir ohne Umschaltung auf Kleinschrift einen Strich an der ersten Bildschirmposition hatten. Nach dem Umschalten wurde er ein großes »E«. Der Grund für diese ungewöhnliche Reaktion: die Basic-Befehlswörter sind im ASCII-Code gespeichert. Wir hatten aber diesen Code direkt in den Bildschirmspeicher geschrieben und dabei Glück, daß überhaupt etwas erkennbar war - denn Bildschirm- und ASCII-Code stimmen nicht überein. Auf jeden Fall verwenden wir diesmal die richtige Methode und geben über $FFD2 das ASCII-Zeichen aus. Wir erhöhen jetzt das y-Register um »1« und vergleichen mit $03 (weil wir wissen, daß es drei Ziffern sind). Ergibt dieser Vergleich nicht Null, wird das nächste Zeichen ausgegeben, ansonsten geht's mit BRK zurück zum SMON.

Logische Operationen– AND, ORA, EOR und BIT

Von den Basic-Befehlsworten haben wir vorhin das erste Wort angezeigt, aber nur weil wir wußten, daß es drei Buchstaben besitzt. Das ist einigermaßen unergiebig, da ja alle Befehle unterschiedlich lang sind. Also sollten wir uns eine Methode überlegen, bei der dieser Unterschied berücksichtigt wird:

```
A 200A
200A AND #80
200C BEQ 201A
200E LDA A09E,Y
2011 AND #7F
2013 JSR FFD2
2016 INY
2017 BNE 2002
2019 BRK
201A LDA A09E,Y
201D INY
201E BNE 2004
2020 BRK
2021 F
```

Nach diesen Zeilen sollten Sie D20002021 eingeben, um die gesamte Routine zu sehen. Bis Zeile 200A sind wir mit dem vorigen Beispiel identisch, doch hier erscheint ein neuer Befehl: AND. Er führt eine logische UND-Verknüpfung des Akkus mit dem Argument hinter AND durch (bitweise). Das Ergebnis steht anschließend im Akku. Um die Wirkung dieses Befehls zu durchleuchten, betrachten wir uns einmal die gerade verwendeten Zahlen in Binärdarstellung:

```
Akku $45 %0100101
AND  $80 %1000000
----------------
Erg. $00 %0000000
```

AND wirkt als bitweises Filter. Es läßt nur dann ein Bit durch, wenn sowohl im Akku als auch im Argument an der gleichen Bit-Position eine »1« steht. Dann erscheint als Ergebnis im Akku ebenfalls eine »1« an dieser Stelle. Anders ausgedrückt:

```
0 AND 0 ergibt 0
0 AND 1 ergibt 0
1 AND 0 ergibt 0
1 AND 1 ergibt 1
```

Von diesen vier möglichen Zuständen der sich entsprechenden Bits ergibt nur 1 AND 1 das logische Ergebnis 1. Sehen wir uns daher die Basic-Wort-Tabelle an; »MA09E A0A6« zeigt die ersten acht Inhalte und daneben die Zeichen (auf Groß/Kleinschrift umschalten!):

```
:a09e 45 4e c4 46 4f d2 4e 45  enDfoRne
```

Allein am Text rechts sieht man schon, daß mit dem »D« von enD und mit dem »R« von foR etwas geschehen sein muß; beide erscheinen in Großbuchstaben. Der Trick dieser Tabelle basiert auf der Tatsache, daß bei Großbuchstaben des ASCII-Satzes grundsätzlich Bit 7 gesetzt ist. »d« hat daher den Wert $44 (%01000100) und »D« entsprechend $c4 (%11000100). Unsere Methode, dies zu unterscheiden, war die AND-Verknüpfung mit $80 (%10000000). Das Ergebnis dieser AND-Verknüpfung kann bei uns nur dann ein höherer Wert als Null sein, wenn bei einem Wert der getesteten Tabelle Bit 7 gesetzt, also dieser Wert größer 128 ist. Dann ist auch im Akku dieses Bit gesetzt und damit die Zero-Flagge gelöscht.

Der BEQ-Befehl zwingt den Mikroprozessor dazu, bei gesetzter Zero-Flagge nach $201a zu verzweigen. Beim dritten Buchstaben aber ist Bit 7 gesetzt, die Zero-Flagge wird gelöscht und unser Programm wird ab $200E fortgeführt. Hier laden wir nochmal den letzten Wert in den Akku (wir hatten ihn ja mit AND verändert) und führen wieder eine AND-Verknüpfung durch; diesmal mit $F7 (%01111111). Der Effekt: Bit 7 wird gelöscht:

```
Akku $C4 %11000100
AND  $7F %01111111
-----------------
Erg. $44 %01000100
```

Wir können also mit AND nicht nur testen, sondern gezielt auch Bits löschen. Als nächsten Schritt (Zeile 2013) senden wir diesen Wert zur Character-out-Routine. Sie muß ein »d« ausgeben, da ja Bit 7 gelöscht und die Zahl damit kleiner als 128 ist. In der Folge erhöhen wir das y-Register und überprüfen, ob es bereits Null ist. Wenn nicht, beginnt die Routine einfach mit einem Zeilenrücklauf ab Zeile 2002 aufs neue. Damit beginnt das nächste Befehlswort in der nächsten Zeile. Im anderen Falle BRK.

Betrachten wir noch kurz den Programmteil ab Zeile 201A: Auch hier wird das entsprechende Zeichen aus der Tabelle geladen, dann das y-Register um eins erhöht und auf Null getestet (Null = BRK). Ist es jedoch nicht Null, verzweigt das Programm einfach zur Zeile 2004, in der die CHROUT-Routine aufgerufen wird (das Zeichen ist ja noch im Akku).

Wir könnten uns übrigens einiges an Programmlänge sparen, wenn wir den AND in Zeile 200A nicht verwenden und anschließend anstelle, von BEQ, über die Negativ-Flagge verzweigen (BPL - Branch if PLus). Eine andere (um ein Byte längere) Methode ist die Carry-Flagge zu setzen und mit dem Befehl »CMP #127« zu überprüfen, ob der Wert größer/gleich 128 ist. CMP #128 genügt dabei nicht, da 128 im Akku mit 128 verglichen (-128) Null ergibt, und damit die Carry-Flagge noch nicht löscht (kein Übertrag).

Starten Sie (mit G2000) jetzt einfach die Routine. Sie sehen die Basic-Befehle, die in den ersten 256 Bytes der Basic-Befehlstabelle enthalten sind, im Klartext.

Als nächsten logischen Befehl lernen wir ORA kennen (inclusive OR with Akkumulator - ODER-Verknüpfung eines Arguments mit dem Akku). Mit dieser Funktion lassen sich gezielt Bits setzen (kennen Sie sicher schon aus Basic). Hier wird der Akku bitweise mit dem Argument verknüpft. Das Ergebnis liegt wieder im Akku vor. Dabei gilt folgende Wahrheitstabelle:

```
0 ORA 0 = 0
1 ORA 0 = 1
0 ORA 1 = 1
1 ORA 1 = 1
```

Wir haben drei von vier der möglichen Bit-Kombinationen, bei denen das Ergebnis-Bit gesetzt wird. Dazu drei Beispiele:

```
Akku $C4 %11000100
ORA  $00 %00000000
-----------------
Erg. $C4 %01000100
```

Eine ODER-Verknüpfung mit Null ändert am Akku-Inhalt nichts.

```
Akku $C4 %11000100
ORA  $68 %01100010
-----------------
Erg. $E6 %11100110
```

Gesetzte Bits des Akku erscheinen genauso wie gesetzte Bits des Arguments im Ergebnis. Interessant ist diese Funktion zum Setzen von Bits in der Bit-Map (Grafikmodus des C64). Die anderen Bits werden nicht beeinflußt.

```
Akku $44 %01000100
ORA  $80 %10000000
-----------------
Erg. $C4 %11000100
```

Dieses Beispiel stellt die Umkehrfunktion zu der im Beispiel-Listing benützten UND-Verknüpfung dar.

EOR (Exclusive-OR - Exclusive ODER-Verknüpfung) haben wir schon beim Subtrahieren gestreift. Dieser Befehl invertiert die sich gegenüberstehenden gleichen Bits. Ungleiche Bits werden nicht geändert. Das Ergebnis ist wieder im Akku:

```
0 EOR 0 = 0
1 EOR 0 = 1
0 EOR 1 = 1
1 EOR 1 = 0
```

oder als Zahlenbeispiel:

```
Akku $44 %01000100
EOR  $C2 %11000010
-----------------
Erg. $BF %10111111
```

Eine Anwendung haben wir schon kennengelernt, zusätzlich läßt sich mit dieser Funktion z.B. eine Bit-Map invertieren. Im Betriebssystem wird mit dieser Funktion das Cursor-Blinken erzeugt (Bildschirm-Code für Space ist $20).

```
Bsch $20 %00100000 (Space)
EOR  $80 %10000000
-----------------
Bsch $A0 %10100000 (Space invertiert)
```

Beim nächsten Aufruf der Routine erscheint:

```
Bsch $A0 %10100000 (Space invertiert)
EOR  $80 %10000000
-----------------
Bsch $20 %00100000 (Space)
```

Zwischen diesen Invertierungen muß verzögert werden. Der Wechsel geschähe sonst so schnell, daß nur ein Flimmern sichtbar wäre. Bei der Routine für den Cursor passiert dies im Interrupt. Jede 60stel Sekunde werden Zähler erhöht. Wenn das Low-Byte einen bestimmten Wert erreicht, wird einfach die Invertierungs-Routine aufgerufen.

Bleibt nur noch ein logischer Befehl: BIT (test BITs - prüfe die Bits). Dieser Befehl führt eine UND-Verknüpfung der Bits des Akkus mit dem Argument durch. Das Ergebnis erscheint allerdings nicht im Akku, sondern in der Negativ- (Bit 7) und der Overflow-Flagge (Bit 6). Sowohl Akku als auch Argument behalten ihren ursprünglichen Wert. Das Ergebnis der Verknüpfung der Bits 0 bis 5 wird zwar nicht festgehalten, wohl gibt aber die Zero-Flagge Auskunft darüber, es Null (Z=1) oder größer Null war (Z=0). Bei geeigneter Maskenwahl kann also über die drei Flaggen jedes Bit getestet werden:

```
Akku $C4 %11000100
BIT  $44 %01000100
------------------
Erg. $44 %01000100 (wird nicht festgehalten)
Negativ-Flagge  (N) = 0
Overflow-Flagge (V) = 1
Zero-Flagge     (Z) = 0
```

Wenn wir zwei andere Zahlen verwenden, wird auch die Funktion der Zero-Flagge sichtbar:

```
Akku $C4 %11000100
BIT  $80 %10000000
------------------
Erg. $44 %10000000 (wird nicht festgehalten)
Negativ-Flagge  (N) = 1
Overflow-Flagge (V) = 0
Zero-Flagge     (Z) = 0
```

und als zweite Zahl

```
Akku $C4 %11000100
BIT  $20 %00100000
------------------
Erg. $00 %00000000 (wird nicht festgehalten)
Negativ-Flagge  (N) = 0
Overflow-Flagge (V) = 0
Zero-Flagge     (Z) = 1
```

Bei der Verwendung von BIT sollten Sie berücksichtigen, daß dieser Befehl nur die Adressierungsarten Absolut (BIT $$$$) und Zero-Page (BIT $$) beherrscht. Sie müssen das Argument also in einer Speicherstelle parat haben. Falls Sie eine Maske wollen, läßt sich dies mit LDA #$$ bewerkstelligen.

Befehls-art	Adressierung	Byte-zahl	Code Hex	Code Dez	Takt-zyklen	Beeinflussung von Flaggen
LSR	»Akkumulator«	1	1A	26	2	N,Z,C
	absolut	3	4E	78	6	N,Z,C
	0-page-absolut	2	46	70	5	N,Z,C
	absolut-X-indiz.	3	5E	94	7	N,Z,C
	0-page-X-indiz.	2	56	86	6	N,Z,C
ROL	»Akkumulator«	1	2A	42	2	N,Z,C
	absolut	3	2E	46	6	N,Z,C
	0-page-absolut	2	26	38	5	N,Z,C
	absolut-X-indiz.	3	3E	62	7	N,Z,C
	0-page-X-indiz.	2	36	54	6	N,Z,C
ROR	»Akkumulator«	1	6A	106	2	N,Z,C
	absolut	3	6E	110	6	N,Z,C
	0-page-absolut	2	66	102	5	N,Z,C
	absolut-X-indiz.	3	7E	126	7	N,Z,C
	0-page-X-indiz.	2	76	118	6	N,Z,C

Drei Befehle zum bitweisen Verschieben

Verschiebebefehle – ASL, ROL, LSR und ROR

Sie sind mittlerweile fast Programmierprofi geworden, doch einige mathematische Befehle sollten Sie noch kennenlernen. Die Basis aller Rechnungen des Mikroprozessors ist ein Byte. Ein Byte besteht aus acht Bit. Jedes höherwertige Bit hat die doppelte Wertigkeit des jeweils niedrigerwertigen. Das schauen wir uns einmal genauer an:

%00000001	%00000010	%00000100	%00001000
1	2	4	8

Was haben wir bei unserem Beispiel gemacht? Wir haben, um von 1 auf »2« zu kommen, das Bit um eine Stelle nach links verschoben. Eine weitere Verschiebung nach links macht aus »2« eine »4«. Das heißt eine Stellenverschiebung nach links entspricht einer Multiplikation mit zwei.

Versuchen wir dieses Rechenbeispiel mit anderen Zahlen:

```
%00001011 (dez. 11)
%00010110 (dez. 22)
```

Auf diese Art lassen sich also auch größere Zahlen verdoppeln - und, natürlich läßt sich dieser Vorgang auch Umdrehen. Das heißt, eine Stellenverschiebung nach rechts entspricht einer Division durch zwei.

Unser Mikroprozessor beherrscht diese Rechenarten:

ASL (Arithmetik Shift Left - rechnerische Linksverschiebung) verschiebt den Akku oder die adressierte Speicherstelle um ein Bit nach links, in das nullte Bit wird eine »0« geschoben. Wäre das alles, ließe sich nur mit einem Byte rechnen. Das siebente Bit, das jetzt nicht mehr ins Byte hineinpaßt, bedeutet einen Übertrag. Und richtig, es wird in die Carry-Flagge geschoben:

```
A5000
5000 LDA #8B (%10001011)
5002 ASL
5003 BRK
5004 F
```

ergibt nach dem Start (G5000) im Akku $16 (%00010110) und eine gesetzte Carry-Flagge.

Um ein höherwertiges Byte damit zu verknüpfen, müssen wir dieses zuerst um ein Bit verschieben, dann die Carry-Flagge in die niedrigste Stelle bringen. Dafür ist ein anderer Befehl zuständig:

ROL (ROtate Left one bit - verschiebe ein Bit nach links). Bei ihm wird zuerst um ein Bit nach links geschoben, das siebte (jetzt achte) Bit gemerkt und die Carry-Flagge in die nullte Stelle geschoben. Der gemerkte Übertrag wandert wieder in die Carry-Flagge. Wozu braucht man in der Praxis diesen Befehl?

Nehmen wir an, Sie haben im Speicher ab Speicherposition $4000 eine Tabelle angelegt, jeweils im Low- / High-Byte-Format. Sie wollen von dieser Tabelle die beiden Bytes laden, die an 129ster Stelle liegen. Da Ihre Tabelle je aus 2Byte besteht, benötigen Sie also den (129 x 2=258) 258sten und 259sten Wert, und zwar ab Speicherstelle $4000. Wir gehen folgendermaßen vor:

```
A3A00
3A00 LDA #81
3A02 STA  A8
3A04 LDA #00
3A06 STA  A9
3A08 ASL  A8
3A0A ROL  A9
3A0C CLC
3A0D LDA  A9
3A0F ADC #40
3A11 STA  A9
3A13 BRK
3A14 F
```

In Zeile 3A00 wird 129 ($81) geladen und in Speicherstelle $A8 abgelegt. Da 129 in ein Byte paßt, kommt in die zweite Speicherstelle ($A9) der Wert Null. In Zeile 3A08 wird das Low-Byte mit zwei multipliziert, der Übertrag ist in der Carry-Flagge, und in Zeile 3A0A wird das High-Byte mit zwei multipliziert und der Inhalt der Carry-Flagge in das niedrigstwertige Bit übernommen. Anschließend wird zum High-Byte $40 addiert. Wenn Sie jetzt mit G3A00 starten und sich mit M00A8 Low- und High-Byte betrachten, steht dort:

```
:00A8 02 41 usw.
```

Damit ist der richtige Wert in den Speicherstellen. Man kann diese als Pointer verwenden. Doch zuvor noch zwei andere Befehle:

LSR (Logical Shift Right - logisches Verschieben nach rechts) verschiebt den Inhalt des Bytes um ein Bit nach rechts, und Bit 7 wird Null. Bit 0 kommt in die Carry-Flagge.

LSR ist die Umkehrung von ASL.

ROR (ROtate Right one bit - verschiebe um ein Bit nach rechts) verschiebt den Inhalt des Bytes um ein Bit nach rechts und bringt die Carry-Flagge in Bit 7. Auch hier wird Bit 0 in die Carry-Flagge übertragen. ROR ist die Umkehrung von ROL.

Die indirekt-indizierte und die indiziert-indirekte Adressierung

Wir sagten, daß unsere Werte als Pointer verwendet werden können. Pointer heißt »Zeiger« und hat etwas mit indirekt-indizierter Adressierung zu tun. Vervollständigen wir unser Programmbeispiel:

```
A3A13
3A13 LDY #00
3A15 LDA (A8),Y
3A17 STA 4000,Y
3A1A INY
3A1B SEC
3A1C CPY #04
3A1E BNE 3A15
3A20 BRK
3A21 F
```

Zeile 3A13 lädt das y-Register mit dem Wert Null. Das ist uns wohl bekannt. Aber Zeile 3A15 ist neu für uns. Hier wird nicht, wie man zunächst vermuten könnte, aus $A8 plus Y geladen, sondern in $A8 und $A9 steht ein Zeiger (Pointer) auf eine Adresse. In unserem Fall (durch die vorherige Rechnung) steht in $A8 »02« und in $A9 »41«. Damit deutet der Zeiger auf die Speicherstelle $4102. Wir laden in unserem Fall den Wert aus der Speicherstelle $4102 + Y (=0) und speichern ihn in die Zelle $4000 + Y. Danach erhöhen wir das y-Register, vergleichen mit $04 und wiederholen die Aktion: Diesmal holen wir aus Speicherstelle $4102 + $01 = $4103 und speichern in $4001. Der Vorgang wird solange wiederholt, bis im y-Register die Zahl »04« steht, dann erfolgt BRK. Damit haben wir vier Werte übernommen und auf einen anderen Bereich übertragen. Sie sehen den Vorteil dieser »indirekt-indizierten« Adressierung. Durch sie lassen sich die Speicherpositionen berechnen, aus denen Werte geholt und bearbeitet werden. Zwei Dinge sind dabei zu beachten:

1. Indirekt-indiziert läßt sich nur auf die Zero-Page anwenden.
2. Diese Adressierung läßt nur das y-Register zu.

Für das x-Register steht eine andere Indizierungsart parat: die indiziert-indirekte Adressierung. Ihre Schreibweise:

```
LDA (A9,X)
```

hat also eine gewisse Ähnlichkeit. Der angegebene Wert (A9) muß wieder eine Zero-Page-Adresse sein. Der Wert in der Speicherstelle, auf die »indiziert« wird, dient als High-Byte und das x-Register als Low-Byte. Bei uns steht in $A9 der Wert $41. Wenn das x-Register auf $00 steht, wird oben aus Speicherstelle $4100 geladen. Hat das x-Register $01 zum Inhalt, wird $4101 übernommen usw.

Welche Befehle für welche Adressierungsarten erlaubt sind, sehen Sie auf unserem Poster auf S. 26/27.

Eine indirekte Adressierung soll hier nicht vergessen werden: der indirekte Sprung. Wir hatten bei JSR gehört, daß, zu einem Unterprogramm gesprungen, vorher die Position des Mikroprozessors gemerkt wird und RTS wieder an die ursprüngliche Position zurückkehrt. Der Befehl JMP (JuMP to adress - springe zu einer Adresse) ist die dem »GOTO«-BASIC-Befehl entsprechende Anweisung in Assembler. Hier wird ohne Rücksicht auf die derzeitige Position auf die Adresse verzweigt, die als Argument dient.

```
JMP 4000
```

springt nach $4000. Dieser Befehl ist aber auch indirekt anwendbar. Für unser Beispiel könnte man schreiben:

```
JMP (00A8)
```

Damit Sie den indirekten Sprung kennenlernen, befindet sich Listing 14 auf Diskette. Laden und betrachten Sie es sich (L"LI.14" und D1400140A).

Transportbefehle im Mikroprozessor – TXA, TAX, TYA, TAY, TSX und TXS

Ab und zu ist es nötig, Registerinhalte gegeneinander auszutauschen. Viele Dinge können nur im Akku geschehen (Addition, Subtraktion usw.). Wollen Sie eine dieser Operationen z.B. mit dem x-Register durchführen, verschieben Sie einfach zuerst den x-Inhalt in den Akku mit TXA (Transfer register X into Accumulator - kopiere den Inhalt des x-Registers in den Akku). Dann führen Sie die Operation durch und kopieren wieder den Akku zurück ins x-Register mit TAX (Transfer Accumulator into register X - kopiere den Akku ins x-Register). Das gleiche läßt sich mit dem y-Register bewerkstelligen:

TYA (Transfer register Y into Accumulator - kopiere y-Register in den Akku) und TAY (Transfer Accumulator into register Y - kopiere Akku ins y-Register).

6510: alle Befehl

Verschiebebefehle

ASL – Arithmetic shift left

Akkumulator oder Speicherstelleninhalt um ein Bit nach links verschieben. Bit 0 wird gelöscht, Bit 7 ins C-Flag geschoben. Das Ergebnis steht in der Datenquelle.

Adr.Art	Code	Länge	Zyklen
Akku	$0a	1	2
Abs:	$0e	3	6
OPge	$06	2	5
Abs.X	$1e	3	7
OPge,X	$16	2	6

Flags: NVBDIZC
* **

ROL – Rotate left one bit

Akkumulator oder Speicherstelleninhalt wird um ein Bit nach links verschoben. C-Flag-Inhalt nach Bit 0 und Bit 7 ins C-Flag.

Adr.Art	Code	Länge	Zyklen
Akku	$2a	1	2
Abs:	$2e	3	6
OPge	$26	2	5
Abs.X	$3e	3	7
OPge,X	$36	2	6

Flags: NVBDIZC
* **

LSR – Logical shift right

Akkumulator oder Speicherstelleninhalt um ein Bit nach rechts verschieben. Bit 7 wird gelöscht, Bit 0 ins C-Flag geschoben. Das Ergebnis steht in der Datenquelle.

Adr.Art	Code	Länge	Zyklen
Akku	$4a	1	2
Abs:	$4e	3	6
OPge	$46	2	5
Abs.X	$5e	3	7
OPge,X	$56	2	6

Flags: NVBDIZC
0 **

ROR – Rotate right one bit

Akkumulator oder Speicherstelleninhalt wird um ein Bit nach rechts verschoben. C-Flag-Inhalt nach Bit7 und Bit0 ins C-Flag.

Adr.Art	Code	Länge	Zyklen
Akku	$6a	1	2
Abs:	$6e	3	6
OPge	$66	2	5
Abs.X	$7e	3	7
OPge,X	$76	2	6

Flags: NVBDIZC
* **

Arithmetik

INC – Increment memory

Der Inhalt der adressierten Speicherstelle wird um 1 inkrementiert

Adr.Art	Code	Länge	Zyklen
Abs:	$ee	3	6
OPge	$e6	2	5
Abs.X	$fe	3	7
OPge,X	$f6	2	6

Flags: NVBDIZC
* *

INX – Increment X-Register

Der Inhalt des X-Registers wird um 1 inkrementiert

Adr.Art	Code	Länge	Zyklen
Implizit	$e8	1	2

Flags: NVBDIZC
* *

INY – Increment Y-Register

Der Inhalt des Y-Registers wird um 1 inkrementiert.

Adr.Art	Code	Länge	Zyklen
Implizit	$c8	1	2

Flags: NVBDIZC
* *

DEC – Decrement memory

Der Inhalt der adressierten Speicherstelle wird um 1 dekrementiert

Adr.Art	Code	Länge	Zyklen
Abs:	$ce	3	6
OPge	$c6	2	5
Abs.X	$de	3	7
OPge,X	$d6	2	6

Flags: NVBDIZC
* *

DEX – Decrement X-Register

Der Inhalt des X-Registers wird um 1 dekrementiert

Adr.Art	Code	Länge	Zyklen
Implizit	$ca	1	2

Flags: NVBDIZC
* *

DEY – Decrement Y-Register

Der Inhalt des Y-Registers wird um 1 dekrementiert.

Adr.Art	Code	Länge	Zyklen
Implizit	$88	1	2

Flags: NVBDIZC
* *

Überlegen in Speicher

STA – Store accumulator in memory

Der Inhalt des Akku wird in die adressierte Speicherstelle geschrieben.

Adr.Art	Code	Länge	Zyklen
Abs:	$8d	3	4
OPge	$85	2	3
Abs.X	$9d	3	5
Abs.Y	$99	3	5
(Ind,X)	$81	2	6
(Ind),Y	$91	2	6
OPge,Y	$95	2	4

Flags: keine Veränderung

STX – Store register X in memory

Der Inhalt des X-Register wird in die adressierte Speicherstelle geschrieben.

Adr.Art	Code	Länge	Zyklen
Abs:	$8e	3	4
OPge	$86	2	3
OPge,Y	$96	2	4

Flags: keine Veränderung

STY – Store register Y in memory

Der Inhalt des Y-Register wird in die adressierte Speicherstelle geschrieben.

Adr.Art	Code	Länge	Zyklen
Abs:	$8c	3	4
OPge	$84	2	3
OPge,X	$94	2	4

Flags: keine Veränderung

Sprungbefehle

JMP – Jump to address

Der PC wird mit einer neuen Adresse geladen, was zu einem Sprung im Programm führt.

Adr.Art	Code	Länge	Zyklen
Abs:	$4c	3	3
Indirekt	$6c	3	5

Flags: keine Veränderung

JSR – Jump to subroutine

Der PC plus 2 wird auf den Stapel gebracht. Anschließend wird die neue Adresse in den PC geladen und die Unterroutine aufgerufen.

Adr.Art	Code	Länge	Zyklen
Abs:	$20	3	6

Flags: keine Veränderung

Beeinflussen der Flags

CLC – Clear carry

Das C-Flag wird auf 0 gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$18	1	2

Flags: NVBDIZC
0

SEC – Set carry

Das C-Flag wird auf 1 gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$38	1	2

Flags: NVBDIZC
1

CLV – Clear overflow flag

Das V-Flag wird auf 0 gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$b8	1	2

Flags: NVBDIZC
0

CLD – Clear dezimal mode

Das D-Flag wird auf 0 gesetzt. Die Befehle ADC und SBC arbeiten binär, bis das D-Flag wieder auf 1 gesetzt wird.

Adr.Art	Code	Länge	Zyklen
Implizit	$d8	1	2

Flags: NVBDIZC
0

SED – Set decimal mode

Das D-Flag wird auf 1 gesetzt. Die Befehle ADC und SBC arbeiten dezimal, bis das D-Flag wieder auf 0 gesetzt wird.

Adr.Art	Code	Länge	Zyklen
Implizit	$f8	1	2

Flags: NVBDIZC
1

CLI – Clear interrupt flag

Das I-Flag wird auf 0 gesetzt. Es werden weitere Programmunterbrechungen (interrupts) zugelassen.

Adr.Art	Code	Länge	Zyklen
Implizit	$58	1	2

Flags: NVBDIZC
0

SEI – Set interrupt mask

Es werden keine weiteren Programmunterbrechungen (Interrupts) zugelassen.

Adr. Art	Code	Länge	Zyklen
Implizit	$78	1	2

Flags: NVBDIZC
1

Addition/Subtraktion

ADC – Add with Carry

Addiert ein Argument zum Akku unter Berücksichtigung des C-Flag.

Besonderheiten:
- arbeitet binär oder dezimal
- für korrektes Addieren muß das C-Flag gelöscht sein

Adr.Art	Code	Länge	Zyklen
Abs:	$6d	3	4
OPge	$65	2	3
imm	$69	2	2
Abs.X	$7d	3	4*
Abs.Y	$79	3	4*
(Ind.X)	$61	2	6
(Ind),Y	$71	2	5*
OPge,X	$75	2	4

* Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
** **

SBC – Subtract with Carry

Subtrahiert ein Argument vom Akku unter Berücksichtigung des C-Flag.

Besonderheiten:
- arbeitet binär oder dezimal
- für korrektes Addieren muß das C-Flag gesetzt sein

Adr.Art	Code	Länge	Zyklen
Abs:	$ed	3	4
OPge	$e5	2	3
imm	$e9	2	2
Abs.X	$fd	3	4*
Abs.Y	$f9	3	4*
(Ind.X)	$e1	2	6
(Ind),Y	$f1	2	5*
OPge,X	$f5	2	4

* Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
** **

CPY – Compare to register Y

Die adressierten Daten werden vom Y-Register abgezogen, das Ergebnis jedoch nicht gespeichert. Die Flags N, Z und C werden entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1, C = 0, wenn Y kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt des Y-Register größer oder gleich ist.

Adr.Art	Code	Länge	Zyklen
Abs:	$cc	3	4
OPge	$c4	2	3
imm	$c0	2	2

Flags: NVBDIZC
* **

Verzweigungsbefehle

BNE – Branch if not equal to zero

Testet das Nullflag. Ist Z = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetztem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$d0	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BEQ – Branch if equal to zero

Testet das Nullflag. Ist Z = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$f0	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BPL – Branch if plus

Testet das Vorzeichenflag. Ist N = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetztem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$10	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BMI – Branch if minus

Testet das Vorzeichenflag. Ist N = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$30	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

auf einen Blick

CMP – Compare to accumulator

Die adressierten Daten werden vom Akku abgezogen, das Ergebnis jedoch nicht gespeichert. Die Flags N, Z und C werden entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1 und C = 0, wenn der Akku kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt des Akku größer oder gleich ist.

Adr.Art	Code	Länge	Zyklen
Abs:	$cd	3	4
OPge	$c5	2	3
imm	$c9	2	2
Abs.X	$dd	3	4*
Abs.Y	$d9	3	4*
(Ind.X)	$c1	2	6
(Ind),Y	$d1	2	5*
OPge,X	$d5	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
* **

CPX – Compare to register X

Die adressierten Daten werden vom X-Register abgezogen, das Ergebnis jedoch nicht gespeichert. Die Flags N,Z und C werden entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1 und C = 0, wenn X kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt des X-Register größer oder gleich ist.

Adr.Art	Code	Länge	Zyklen
Abs:	$ec	3	4
OPge	$e4	2	3
imm	$e0	2	2

Flags: NVBDIZC
* **

Laden aus dem Speicher

LDA – Load accumulator

Der Akku wird mit einem neuen Wert geladen.

Adr.Art	Code	Länge	Zyklen
Abs:	$ad	3	4
OPge	$a5	2	3
imm	$a9	2	2
Abs.X	$bd	3	4*
Abs.Y	$b9	3	4*
(Ind.X)	$a1	2	6
(Ind),Y	$b1	2	5*
OPge,X	$b5	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
* *

LDX – Load register X

Register X wird mit einem neuen Wert geladen.

Adr.Art	Code	Länge	Zyklen
Abs:	$ae	3	4
OPge	$a6	2	3
imm	$a2	2	2
Abs.Y	$be	3	4*
OPge,Y	$b6	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
* *

LDY – Load register Y

Register Y wird mit einem neuen Wert geladen.

Adr.Art	Code	Länge	Zyklen
Abs:	$ac	3	4
OPge	$a4	2	3
imm	$a0	2	2
Abs.X	$bc	3	4*
OPge,X	$b4	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
* *

Stack-Manipulationen

PLA – Pull accumulator

Der Akku wird mit dem Inhalt der Stapelspitze geladen.

Adr.Art	Code	Länge	Zyklen
Implizit	$68	1	4

Flags: NVBDIZC
* *

PHA – Push accumulator

Der Inhalt des Akku wird auf den Stapel gebracht.

Adr.Art	Code	Länge	Zyklen
Implizit	$48	1	3

Flags: keine Veränderung

PHP – Push processor status

Der Inhalt des Statusregisters wird auf den Stapel gebracht.

Adr.Art	Code	Länge	Zyklen
Implizit	$08	1	3

Flags: keine Veränderung

PLP – Pull processor status

Das Statusregister wird mit dem Inhalt der Stapelspitze geladen.

Adr.Art	Code	Länge	Zyklen
Implizit	$28	1	4

Flags: NVBDIZC

BVC – Branch if overflow clear

Testet das Überlaufsflag. Ist V = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetztem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$50	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BVS – Branch if overflow set

Testet das Überlaufsflag. Ist V = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$70	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BCC – Branch if carry clear

Testet das Übertragsflag. Ist C = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetzem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$90	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BCS – Branch if carry set

Testet das Übertragsflag. Ist C = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$b0	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

Rückkehr aus Unterprogrammen

RTS – Return from subroutine

Der PC wird vom Stapel zurückgeholt und auf den nächsten Befehl gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$60	1	6

Flags: keine Veränderung

RTI – Return from interrupt

Stellt den ursprünglichen Zustand des Statusregisters und des PCs nach einer Programmunterbrechung wieder her.

Adr.Art	Code	Länge	Zyklen
Implizit	$40	1	6

Flags: NVBDIZC
****

Sonstiges

BRK – break

Der PC und das Statusregister werden auf den Stapel gebracht. Als neue Adresse wird der Inhalt der Speicherstellen $fffe/$ffff übernommen. Zusätzlich wird das B-Flag gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$00	1	7

Flags: NVBDIZC
1 1

NOP – No operation

Wartet zwei Taktzyklen.

Adr.Art	Code	Länge	Zyklen
Implizit	$ea	1	2

Flags: keine Veränderung

Verschieben innerhalb der CPU

TAX – Transfer accumulator into register X

Kopiert den Inhalt des Akkumulators ins X-Register

Adr.Art	Code	Länge	Zyklen
Implizit	$aa	1	2

Flags: NVBDIZC
* *

TXA – Transfer register X into accumulator

Kopiert den Inhalt des X-Registers in den Akkumulator.

Adr. Art	Code	Länge	Zyklen
Implizit	$8a	1	2

Flags: NVBDIZC
* *

TAY – Transfer accumulator into register Y

Kopiert den Inhalt des Akkumulators ins Y-Register

Adr. Art	Code	Länge	Zyklen
Implizit	$a8	1	2

Flags: NVBDIZC
* *

TYA – Transfer register Y into accumulator

Kopiert den Inhalt des Y-Registers in den Akkumulator.

Adr. Art	Code	Länge	Zyklen
Implizit	$98	1	2

Flags: NVBDIZC
* *

TXS – Transfer register X into Stackpointer

Kopiert den Inhalt des X-Registers in den Stapelzeiger.

Adr.Art	Code	Länge	Zyklen
Implizit	$9a	1	2

Flags: keine Veränderung

TSX – Transfer Stackpointer into register X

Kopiert den Inhalt des Stapelzeigers ins X-Register

Adr.Art	Code	Länge	Zyklen
Implizit	$ba	1	2

Flags: NVBDIZC
* *

Logische Operationen

AND – AND Accu

UND-Verknüpfung eines Arguments mit dem Akku. Das Ergebnis steht im Akku.

0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1

Adr.Art	Code	Länge	Zyklen
Abs:	$2d	3	4
OPge	$25	2	3
imm	$29	2	2
Abs.X	$3d	3	4*
Abs.Y	$39	3	4*
(Ind.X)	$21	2	6
(Ind),Y	$31	2	5*
OPge,X	$35	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
* *

ORA – Inclusive OR with accumulator

ORA-Verknüpfung eines Arguments mit dem Akku. Das Ergebnis steht im Akku.

0 ORA 0 = 0
1 ORA 0 = 1
0 ORA 1 = 1
1 ORA 1 = 1

Adr.Art	Code	Länge	Zyklen
Abs:	$0d	3	4
OPge	$05	2	3
imm	$09	2	2
Abs.X	$1d	3	4*
Abs.Y	$19	3	4*
(Ind.X)	$01	2	6
(Ind),Y	$11	2	5*
OPge,X	$15	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
* *

EOR – Exclusive-OR

EXKLUSIV-ODER Verknüpfung eines Arguments mit dem Akku. Das Ergebnis steht im Akku.

0 EOR 0 = 0
1 EOR 0 = 1
0 EOR 1 = 1
1 EOR 1 = 0

Adr.Art	Code	Länge	Zyklen
Abs:	$4d	3	4
OPge	$45	2	3
imm	$49	2	2
Abs.X	$5d	3	4*
Abs.Y	$59	3	4*
(Ind.X)	$41	2	6
(Ind),Y	$51	2	5*
OPge,X	$55	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
* *

BIT – Test bits

Die adressierten Bytes werden UND-verknüpft, das Ergebnis wird jedoch nicht festgehalten. Die Bits 6 und 7 der adressierten Speicherstelle werden in die Flags V und N übernommen. Der Akku bleibt unverändert.

Adr.Art	Code	Länge	Zyklen
Abs:	$2c	3	4
OPge	$24	2	3

Flags: NVBDIZC
mm *

Leider lassen sich beim 6510 die x- und y-Register nicht untereinander austauschen. Hier müssen Sie als Zwischenspeicher den Akku verwenden.

Zwei andere, weitaus gefährlichere Transportbefehle sind: TSX (Transfer Stackpointer into register X - kopiere den Inhalt des Stapelzeigers ins x-Register) und TXS (Transfer register X into Stackpointer - kopiere Stapelzeiger in x-Register). Der Sinn dieser Befehle ist zunächst nicht ganz klar. Aber bedenken Sie, daß es sonst keine Möglichkeit gibt, an den Inhalt des Stapelzeigers zu kommen. Ein anderer Anwendungszweck ist:

Nehmen wir an, Sie haben eine komplizierte Tastaturauswertung. Sie springen mit JSR von dieser in ein Unterprogramm, dieses ruft ein weiteres Unterprogramm auf (z.B. Speicherung auf Diskette). Was tun Sie, wenn bei der letzten Routine ein Fehler auftritt und Sie sofort zur Fehleranzeige springen wollen und von dort unmittelbar zurück zur Tastaturauswertung? Die JSR-Sprünge zwingen Sie, jeweils mit RTS die alte Reihenfolge zurückzuspringen. Wenn Sie sich allerdings vor dem ersten Unterprogrammaufruf den Stapelzeiger gemerkt haben (TSX), können Sie an einer x-beliebigen Position den Mikroprozessor mit TXS (+3) zwingen, zur ersten Position zurückzuspringen.

Stack-Manipulationen – PLA, PHA, PHP und PLP

Der 6510 hat leider nur drei programmierbare Register: x- und y-Register und den Akku. Für einige Anwendungen reicht die Anzahl der Register nicht aus, z.B für eine verzögerte Ausgabe einer Tabelle. Nehmen wir an, die Tabelle liegt bei $5000:

```
3A00 LDX #00
3A02 LDA 5000,X
3A05 JSR 3A0F
3A08 JSR FFD2
3A0B DEX
3A0C BNE 3A02
3A0E BRK

3A0F PHA
3A10 TXA
3A11 PHA
3A12 TYA
3A13 PHA
3A14 LDX #00
3A16 LDY #00
3A18 DEY
3A19 BNE 3A18
A1B DEX
3A1C BNE 3A18
3A1E PLA
3A1F TAY
3A20 PLA
3A21 TAX
3A22 PLA
3A23 RTS
3A24 F
```

In Zeile 3A00 erhält das x-Register den Wert Null. In der nächsten Zeile laden wir den ersten Wert unserer Tabelle. Danach springen wir ins Unterprogramm ab 3A0F (eine Verzögerungsschleife). Zurückgekehrt wird das Zeichen im Akku ausgegeben, das x-Register erniedrigt und die Schleife solange durchlaufen, bis das x-Register gleich null ist.

So weit, so gut. Wenn von Zeile 3A0F bis 3A13 nicht die Register gerettet, und ab 3A1E wieder zurückholen würden, käme das x-Register immer mit dem Wert Null aus dem Unterprogramm – unsere Routine hätte kein Ende. Der Befehl P (PusH accumulator – rette Akkuinhalt) bringt den Inhalt Akkus auf den Prozessorstapel und erhöht den Stapelzei Leider gibt es keinen Befehl für x- oder y-Register, darum m zuerst der Wert der Register in den Akku gebracht werd danach kann dieser auf den Stapel gerettet werden. Ab Z 3A1E geschieht das Umgekehrte: Mit PLA (PuLl Accumula wird der erste Wert wieder vom Stapel geholt und in den A übertragen. Wie wir vom Stapelprinzip her wissen, ist dies zuletzt abgelegte Wert, also der des y-Registers. Er wird (Z le 3A1F) auch wieder ins y-Register übertragen. Danach schieht das gleiche auch mit dem x-Register und zum Sch mit dem Akku.

Bei unserer kleinen Routine wäre es nur nötig gewes den Akku und das x-Register zu retten, doch ein bißchen cherheit ist besser. Denn falls diese Verzögerung von ei anderen Programmstelle aufgerufen wird, wissen wir nic welche Register wir vielleicht benötigen. Man könnte so noch mehr tun:

PHP (PusH Processor status) bringt das Statusregister den Stapel, und PLP (PuLl Processor status) bringt ihn wie zurück vom Stapel ins Statusregister.

Interrupt – CLI, SEI, RTI

Mit drei Befehlen, die für die Interrupt-Behandlung zust dig sind, kommen wir zum Schluß unseres Kurses. Wie bereits wissen, führt der C64 jede 60stel Sekunde einen terrupt durch. Dieser Interrupt wird von einem der Tim Bausteine ausgelöst. Es gibt außer den Timern noch vi Möglichkeiten, den IRQ auszulösen. Eines aber haben diese Routinen gemeinsam: Sie enden mit dem Befehl (ReTurn from Interrupt - kehre vom Interrupt zurück). Im gensatz zu RTS muß RTI etwas mehr erledigen: RTS me sich, von welcher Position die Routine aufgerufen wurde, RTI kommt noch der Status dazu.

Trotzdem ist es manchmal interessant, den Interrupt aus schalten: z.B. wenn die Speicherkonfiguration geändert w In Assembler haben wir die Möglichkeit, die kompletten KByte Speicher als RAM zu adressieren. Dann geschi beim Interrupt etwas Unangenehmes: Der Mikroprozes versucht, in seinem Betriebssystem die IRQ-Routine aufzu fen. Wir haben allerdings auf RAM umgeschaltet. Daher det der Prozessor sein Programm nicht - und hängt sich Wir vermeiden dies mit SEI (SEt Interrupt mask – setze Interrupt-Flagge). Im Status-Byte ist nämlich eines der Bits den IRQ zuständig. Ist dieses Bit gesetzt, kann der Mikrop zessor keinen IRQ mehr ausführen. Ist es gelöscht, führt CPU die Unterbrechungen wie gewohnt durch. Der Bef zum Wiedereinschalten des IRQ ist CLI (CLear Interrupt fla

Ein Nachzügler – NOP

Fast hätten wir ihn vergessen, den Befehl NOP (No OPe tion – keine Tätigkeit). Er macht das, was sein Name sag nämlich zwei Taktzyklen lang nichts. Gebraucht wird er falls man eine kurze Verzögerung in zeitkritischen Routin benötigt und als Platzhalter bei geänderten Programmen

Falls Ihnen dieser Kurs Appetit auf mehr Informationen macht hat, finden Sie im Anschluß noch einige wichtige bellen. Außerdem empfehlen wir unser Assembler-Sond heft 35, zu bestellen bei Markt & Technik Leserservi CSJ, Postfach 1 40 20, 8000 München 5, Tel. 089/20 25 15 Es beinhaltet einen noch ausführlicheren Kurs, bei dem au Teile des Betriebssystems behandelt werden. Einige der d beschriebenen Routinen finden Sie auch auf unserer Disk te. Als erste Programmierhilfe finden Sie rechts eine Tabe zur Spritebehandlung. (Heimo Ponath, gr)

VIC – Video-Interface

Basis-Adresse = 53248 ($D000)

Sprites

... bauen sich aus einem Bit-Muster auf. Dabei werden 63 Byte benötigt. Pro Sprite stehen jeweils drei Byte in einer Bildschirmzeile. Aus 21 Zeilen besteht ein ganzes Sprite (Entwurf-Blatt). Bei Multicolor bilden je zwei Bit einen Zeiger auf eine der vier möglichen Farben:

00 = Hintergrund	01 = Farbe 1	10 = individuell	11 = Farbe 2

Sprite-Zeiger: stehen im unsichtbaren Teil des Bild-Schirm-RAM. Sie zeigen auf den eigentlichen Sprite-Block. In ihm ist das Muster festgelegt. Die Zeiger beginnen ab Bildschirm + $03F8 im Text-Modus, also im Normalfall bei $07F8. Acht Speicherstellen sind für das jeweilige Sprite verantwortlich. Die Sprite-Blocks sind jeweils 64 Byte ($40) lang. Sie sind von $00 bis $FF **numeriert.** Berechnet wird ihre Position im Speicher folgendermaßen (VIC ist die derzeitig adressierte Offset-Adresse- $00 bei RESET):

VIC + (Blocknummer * 64) = Position

(nach RESET): **0 + (13 * 64) = 832**

Koordinaten: erscheinen in Reihenfolge x-Achse, y-Achse in den Registern. Bei Überschreitung des Offsets sind sie ganz oder teilweise vom Bildschirmrand verdeckt.

Offset für x-Koordinate: < 24
Offset für y-Koordinate: < 29 oder > 229

$D000	x-Koordinate Sprite #0	$D008	x-Koordinate Sprite #4
$D001	y-Koordinate Sprite #0	$D009	y-Koordinate Sprite #4
$D002	x-Koordinate Sprite #1	$D00A	x-Koordinate Sprite #5
$D003	y-Koordinate Sprite #1	$D00B	y-Koordinate Sprite #5
$D004	x-Koordinate Sprite #2	$D00C	x-Koordinate Sprite #6
$D005	y-Koordinate Sprite #2	$D00D	y-Koordinate Sprite #6
$D006	x-Koordinate Sprite #3	$D00E	x-Koordinate Sprite #7
$D007	y-Koordinate Sprite #3	$D00F	y-Koordinate Sprite #7

$D010 neuntes Bit der x-Koordinaten

Ein gesetztes Bit bedeutet, daß die x-Koordinate >255 ist.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

Farb-Register: legen im Normalmodus die Farbe eines Sprites fest (0 bis 16). Bei Multicolor wird nur die die individuelle Farbe (%10) bestimmt.

$D027	Sprite #0	$D02B	Sprite #4
$D028	Sprite #1	$D02C	Sprite #5
$D029	Sprite #2	$D02D	Sprite #6
$D02A	Sprite #3	$D02E	Sprite #7

$D01C Multicolor-Modus

Ein gesetztes Bit schaltet das entspr. Sprite in Mulitcolor.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D025 und $D026 Farben für Multicolor

Bei Multicolor sind für Sprites jeweils zwei Farben identisch, Farbe 1 ($D025) und Farbe 2 ($D026).

$D015 Sprites ein/aus

Ein gesetztes Bit schaltet das entsprechende Sprite ein.

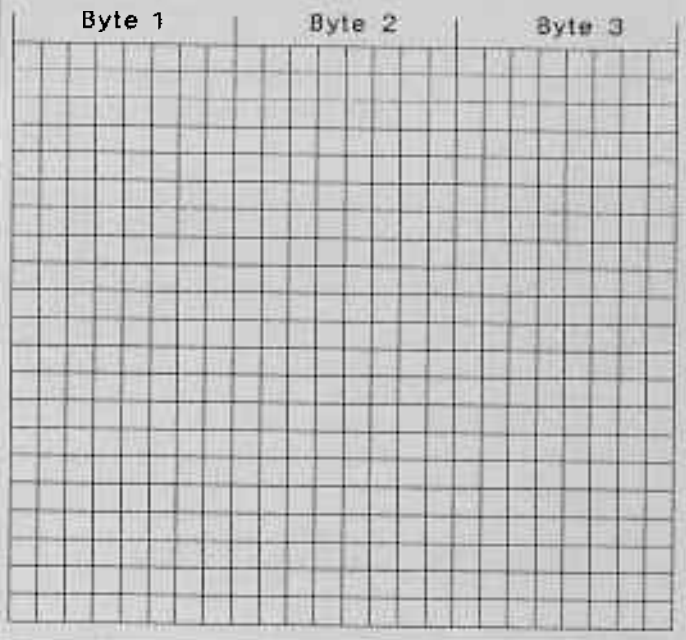
Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01B Priorität Sprite/Hintergrund

Ein gesetztes Bit schiebt das entspr. Sprite hinter den Text.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

Entwurf-Blatt:



$D017 vertikale Vergrößerung

Ein gesetztes Bit vergrößert das entspr. Sprite nach rechts

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01D horizontale Vergrößerung

Ein gesetztes Bit vergrößert das entspr. Sprite nach unten.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01E Kollision Sprite/Sprite

Ein gesetztes Bit zeigt welche Sprites beteiligt waren.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01F Kollision Sprite/Hintergrund

Ein gesetztes Bit zeigt welche Sprites beteiligt waren.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01A Interrupt Maskenregister

legt fest welches Ereignis einen IRQ auslöst

Bit	
0	Raster-Interrupt (1 = ja)
1	Sprite/Hintergrund Kollision (1 = ja)
2	Sprite/Sprite Kollision (1 = ja)
3	Lightpen (1 = ja)
4–7	Unbelegt (immer 1)

$D019 Interrupt Statusregister

zeigt, welches Ereignis einen IRQ ausgelöst hat.

Bit	
0	Raster-Interrupt (1 = ja)
1	Sprite/Hintergrund Kollision (1 = ja)
2	Sprite/Sprite Kollision (1 = ja)
3	Lightpen (1 = ja)
4–6	Unbelegt (immer 1)
7	Interrupt fand statt (1 = ja)