Teil 1 – Magic Disk 01/90

Einleitung:
Dieser Assembler-Kurs soll allen BASIC-Programmierern den Einstieg in Assembler ermöglichen. Ich werde mich in der Folge darum bemühen, die komplizierten Fachbegriffe zu umgehen und leicht verständliche Erklärungen zu geben.
Zunächst sollen innerhalb dieses Kurses alle Assembler-Befehle behandelt werden. In späteren Teilen wird dann auf die Verwendung von Kernal-Routinen und die Interrupt-Technik eingegangen. Zu jedem Teil des Kurses befindet sich ein Programm auf der zweiten Seite der Diskette ("ASSEMBLER-KURS 1"), das einige Assembler-Routinen enthält und Ihnen deshalb gleich die Auswirkungen der jeweiligen Assembler-Befehle demonstrieren kann.

Das Werkzeug:
Zum Programmieren in Assembler benötigen Sie unbedingt ein Programm, das die Befehle in reine Maschinensprache umwandelt. Diese Programme werden Assembler oder Monitore genannt. Ein solches Programm finden Sie unter anderem auf der Rückseite dieser Ausgabe (WIZ-MON); die Beschreibung dazu entnehmen Sie der entsprechenden Rubrik.

LDA, STA und BRK:
Und jetzt können wir schon mit den ersten Befehlen loslegen.Wir fangen an mit LDA (LoaD Accumulator) Der Akkumulator (oder kurz Akku genannt) ist eine Art Verteilerstelle. Mit diesem Befehl lädt man also einen Wert (Zahl) in den Akku.
STA (STore Accumulator) schreibt diesen Akkuinhalt dann in die angegebene Speicherstelle.
Der Befehl BRK entspricht etwa dem BASIC-Befehl STOP und sollte zunächst immer am Ende Ihrer ersten Programmierversuche mit Assembler stehen.
LDA und STA sind durchaus mit PEEK und POKE aus dem BASIC vergleichbar.

Beispiel: POKE 8192,8

Dieser Befehl schreibt also den Wert 8 in die Speicherstelle 8192. Eine Routine in Assembler, die die gleiche Aufgabe erfüllt, lautet:

```
LDA #$08
STA $2000
BRK
```

Lassen Sie sich nicht dadurch verwirren, daß hier Hexadezimalzahlen (Erklärung im Programm!) verwendet werden. Das Zeichen "#" in der ersten Zeile zeigt an, daß eine konstante Zahl und nicht der Inhalt einer Speicherzelle in den Akku geladen wird. Die hexadezimale Zahl $2000 entspricht haargenau der dezimalen Zahl 8192. Zunächst wird also die Zahl 8(bzw.$08) in den Akku geladen. In der zweiten Zeile wird dieser Wert in der Adresse 8192($2000) abgelegt.

Ein anderes Beispiel: Nach der Eingabe von

```
POKE 12288, PEEK (8192)
```

hätte die Adresse 12288 ebenfalls den Inhalt 8. Die entsprechenden Assembler-Befehle lauten:

```
LDA $2000
STA $3000
BRK
```

Mit LDA wird diesmal keine Zahl, sondern der Inhalt der Speicherstelle $2000(ähnlich PEEK(8192) in den Akku geladen. STA legt den Akkuinhalt daraufhin bei $3000(dezimal 12288) ab.

Der Umgang mit dem Monitor:
Eingegeben werden die Assembler-Befehle mit einem Monitor durch:

A Speicherstelle Assembler-Befehl z.B.: **A C000 LDA #$08**.
Wenn Sie Ihre eingegebenen Befehlsfolgen noch einmal betrachten wollen, so können Sie dies mit
"**D Speicherstelle**" z.B.: **D C000**
tun. Es wird Ihnen dann bei unserem Beispiel folgende Zeile gezeigt:

```
. C000   A9 08   LDA #$08
```

Die Zahl C000 ist die Speicherstelle, in der der Befehl steht. A9 ist die Codezahl für den LDA-Befehl. Die darauffolgende Zahl zeigt den Wert an, der in den Akku geladen wird. Uns brauchen zunächst nur die Assembler-Befehle im Klartext zu interessieren.
Laden Sie nun das Beispielprogramm "ASSEMBLER-KURS 1" von der zweiten Seite. Es wird Ihnen weitere Hilfen zu den ersten Assembler-Befehlen und zum hexadezimalen Zahlensystem geben.
In der nächsten Ausgabe behandeln wir die Indexregister und die dazugehörigen Assembler-befehle. Bis dahin wünschen ich Ihnen viel Spaß beim ausprobieren der gelernten Befehle!

(rt)

Teil 2 – Magic Disk 02/90

Diesmal werden gleich 15 neue, aber dafür doch recht einfache Befehle vorgestellt.
Vorher müssen wir uns jedoch etwas näher mit den Indexregistern (X-Register und Y-Register) beschäftigen: Jedes dieser beiden Register kann ein Byte aufnehmen, d.h. Werte zwischen $00 und $FF. Sie werden hauptsächlich im Zusammenhang mit den verschiedenen Adressierungsarten verwendet.

LDX	Beispiel: LDX #$0A oder LDX$1000 LDX entspricht in seiner Form dem schon bekannten LDA-Befehl.
LDY	ist der entsprechende Befehl für das Y-Register.
STX / STY	erinnern sehr stark an den STA-Befehl. Und so ist es auch. Bei STX bzw. STY wird der Inhalt des X bzw. Y-Registers in die angegebene Speicher-stelle geschrieben.

Sie bemerken sicherlich die enge Beziehung zwischen dem Akku und den Indexregistern. Für einen schnelleren Datenaustausch zwischen diesen Registern sind die Befehle TAX, TXA, TAY und TYA verantwortlich.

TAX / **TAY**	Der Inhalt des Akkus wird im X-Register abgelegt. Dadurch verändert sich jedoch der Inhalt des Akkumulators nicht. Für das Y-Register lautet der Befehl TAY.
TXA / **TYA**	ist die Umkehrung des TAX bzw. TAY. Nun wird der Inhalt des X bzw. Y-Registers in den Akku geschrieben, ohne das sich das X bzw. Y-Register verändert.
DEX / DEY / **DEC**	Während DEX und DEY das jeweilige Indexregister um 1 erniedrigen, wirkt der DEC-Befehl auf die nach ihm folgende Speicherstelle. Auch von deren Inhalt wird der Wert 1 abgezogen. Der Akku ist davon nicht betroffen.
INX / INY / **INC**	Diese Assemblerbefehle bewirken genau das Gegenteil der eben vorgestellten. Hier werden die jeweiligen Register um den Wert 1 erhöht.

Was passiert aber, wenn z.B. im X-Register der Wert $FF steht und der Prozessor auf den Befehl INX trifft?

Da ein Byte keinen größeren Wert als $FF annehmen kann, wird das X-Register kurzerhand auf $00 gesetzt. Ähnlich sieht es bei $00 und einem DEX aus. Dann wird einfach wieder "oben" angefangen und $FF gesetzt.

RTS steht gewöhnlich am Ende eines Assemblerprogrammes oder Unterprogrammes und soll künftig unser BRK ersetzen.

Wenn ein Programm auf den RTS-Befehl trifft, wird es nicht mehr unterbrochen, sondern es wird ins Basic gesprungen.

Laden Sie nun den zum Kurs gehörende Programm aus dem GAMESMENUE oder direkt von der ersten Seite dieser MAGIC DISK 64- Ausgabe. Es werden Ihnen die hier vorgestellten Befehle noch einmal verdeutlicht.
Auf der nächsten Seite folgt eine Zusammenstellung der neuen Befehle. Ich wünsche Ihnen viel Erfolg beim Ausprobieren der neu erlernten Befehle.

LDX	(LoaD X-register)
LDY	(LoaD Y-register)
STX	(STore X-register)
STY	(STore Y-register)
DEC	(DECrement memory)
DEX	(DEcrement X-register)
DEY	(DEcrement Y-register)
INC	(INCrement memory)
INX	(INcrement X-register)
INY	(INcrement Y-register)
TAX	(Transfer Accu into X)
TAX	(Transfer Accu into Y)
TXA	(Transfer X into Accu)
TYA	(Transfer Y into Accu)
RTS	(ReTurn from Subroutine)

In der nächsten Ausgabe der MAGIC DISK 64 behandeln wir Vergleichsbefehle und die bedingte Verzweigung.

(wk)

Kurs Teil 3 – Magic Disk 03/90

Diesmal geht es in unserem Kurs um die Vergleichsbefehle und die bedingte Verzweigung. Dafür muß jedoch zunächst das Statusregister des Prozessors näher erklärt werden. Dieses 8-Bit-Register bildet die Grundlage für die bedingte Verzweigung. Die einzelnen Bits des Statusregisters nennt man Flaggen (oder Flags). Jeder dieser Flaggen kann direkt angesprochen oder abgefragt werden und wird automatisch gesetzt oder gelöscht.
Es existieren folgende Flaggen:

Bit 0 Carry-Flag Dieses Flag wird gesetzt, wenn ein "Übertrag" stattfindet. Wenn z. B. bei der Addition zweier 8- Bit-Zahlen das Ergebnis größer als 255 ($FF) ist und nicht mehr in ein Byte paßt, dann wird das Carry-Bit gesetzt. Bei der Subtraktion erfüllt das Carry-Bit eine entgegengesetzte Aufgabe. Hierbei ist das Carry-Bit zunächst gesetzt. Beim Subtrahieren zweier Zahlen kann selbstverständlich kein Überlauf stattfinden. Sollte das Ergebnis jedoch kleiner als null werden ("Unterlauf"), so wird das Carry-Flag gelöscht.

Bit 1 Zero-Flag Dieses Bit ist gewöhnlich gelöscht. Wenn das Ergebnis einer Befehlsfolge aber Null ergibt, wird sie gesetzt.

Bit 2 IRQ-Flag Sie ermöglicht oder verhindert Interrupts. Sie spielt in den späteren Kursteilen eine größere Rolle.

Bit 3 Dezimal-Flag Von ihr hängt es ab, ob eine Rechenoperation im Dezimalmodus ausgeführt wird. Normalerweise ist dieses Bit gelöscht und alle Rechnungen werden binär ausgeführt.

Bit 4 Break-Flag Wenn das Programm auf einen BRK-Befehl trifft, wird diese Flagge gesetzt.

Bit 5 ist nicht belegt.

Bit 6 V-Flag ("Overflow") Dieses Bit wird nur im Zweierkomplement-Modus benötigt. In diesem Modus wird das 7. Bit einer Zahl als Vorzeichen aufgefaßt. Es wird automatisch gesetzt, wenn ein Überlauf eingetreten ist, d.h. wenn der Zahlenbereich überschritten wurde und nun das höherwertigste Bit gesetzt wurde, obwohl es eigentlich das Vorzeichen enthalten sollte.

Bit 7 Negativ-Flag Wenn das Ergebnis einer Operation größer als 127($7F) ist, wird diese Flagge gesetzt. Wie oben erwähnt, gibt es einen Modus, in dem Zahlen, die größer als $7F sind (7. Bit gesetzt), als Negativ angesehen werden.

Das fast jeder Befehl auf die Flaggen einwirkt, soll uns der altbekannte LDA-Befehl zeigen.

Beispiel:
```
LDA #00
```

In den Akku wird der Wert $00 geladen. Da das Ergebnis dieser Aktion null ist, wird die Zero-Flagge gesetzt.
Bei
```
LDA #FF
```

hingegen bleibt die Zero-Flagge gelöscht. Da aber ein größerer Wert als $7F geladen wurde, wird nun die Negativ-Flagge gesetzt.
Wir benötigen in diesem Kursteil lediglich die Flaggen C, Z, N und V. Die anderen Flaggen sollten hier nur kurz erwähnt werden. Da Sie nun die Bedeutung der Flaggen kennen, wird Ihnen die Registeranzeige Ihres Maschinensprache-Monitors " N V - B D I Z C" sicherlich etwas mehr sagen.

Die Vergleichsbefehle:

CMP (CoMPare to accu) Dieser Befehl ermöglicht einen Vergleich zwischen dem Inhalt des Akkus und einem beliebigen Wert (oder dem Inhalt einer Speicherstelle). Die Register werden dadurch verglichen, daß der adressierte Wert vom Inhalt des Akkus abgezogen wird. Dabei werden (e nach Ergebnis) die Flaggen C, Z und N verändert. Der Akkumulatorinhalt bleibt unberührt.

Beispiel:
```
LDA #$05
CMP #$01
```

Es wird also der adressierte Wert ($01) vom Inhalt des Akkus ($05) abgezogen. Unsere Flaggen zeigen folgende Inhalte:
Carry-Flagge: gesetzt, da bei dieser "Subtraktion" kein Unterlauf aufgetreten ist.
Zero-Flagge: gelöscht, da die Differenz ($04) größer als Null ist.
Negativ-Flagge: gelöscht, da $04 kleiner als $7F ist.

Der CMP-Befehl erlaubt folgende (schon bekannte) Adressierungsarten:
Unmittelbare Adressierung, z.B: CMP #$10
Absolute Adressierung, z.B: CMP $2000

CPX / **CPY**
(ComPare to X)
Die Vergleichsbefehle CPX und CPY (ComPare to Y) entsprechen dem CMP-Befehl, nur das anstelle des Akkus mit den angegebenen Indexregistern verglichen wird. Da die Vergleichsbefehle nur die Flaggen verändern, benötigen wir aber noch Befehle, die auf den Inhalt dieser Flaggen reagieren.

Die Verzweigbefehle:

BEQ
(Branch on Equal)
Verzweige zur angegebenen Adresse, falls die Zero-Flagge gesetzt ist.

BNE
(Branch on Not Equal)
Verzweige, falls die Zero-Flagge gelöscht ist.

BCC
(Branch on Carry Clear)
Verzweige, falls die Carry-Flagge gelöscht ist.

BCS
(Branch on Carry Set)
Verzweige, falls die Carry-Flagge gesetzt ist.

BMI
(Branch on Minus)
Verzweige, falls die Negativ-Flagge gesetzt ist.

BPL
(Branch on PLus)
Verzweige, falls die Negativ-Flagge gelöscht ist.

BVC
(Branch on oVerflow Clear)
Verzweige, falls die V-Flagge gelöscht ist.

BVS
(Branch on oVerflow Set)
Verzweige, falls die V-Flagge gesetzt ist.

Wie Sie sehen, richten sich jeweils zwei dieser Befehle (gegensätzlich) nach einer Flagge. Wenn die Bedingungen für eine Verzweigung nicht gegeben ist, wird dieser Befehl einfach vom Programm übergangen.
Hier liegt eine deutliche Parallele zur IF ... Then … - Anweisung in BASIC. Bei den bedingten Verzweigungen gibt es noch eine weitere Besonderheit; Sehen wir uns zunächst den Befehl

```
. 2000   D0 03   BNE $2005
```

etwas näher an. Wie Sie sehen, handelt es sich um einen 2- Byte-Befehl, obwohl alleine die Angabe der Sprungadresse schon 2 Bytes benötigen müßte. Diese speicherplatzsparende Besonderheit wird durch die relative Adressierung ermöglicht. Es wird also die Sprungadresse nicht direkt angegeben, sondern nur die Entfernung zum Sprungziel.
Wenn Sie jede Sprungadresse selbst ausrechnen müßten, würde dies zu einer wüsten Rechnerei führen. Da uns der Assembler jedoch diese lästige Arbeit abnimmt, können wir dieses Thema getrost übergehen. Sie müssen nur wissen, daß Sie nicht weiter als 1**29 Bytes hinter** und **126 Bytes vor** Ihren Verzweigungsbefehl springen dürfen.
Dieser eingeschränkte Aktionsradius der relativen Adressierung fällt Ihnen bei der Programmierung kaum noch auf.
Jetzt, am Ende des dritten Teiles haben Sie schon die wichtigsten Grundlagen erlernt und sind in der Lage, kleinere Programme zu schreiben. Was Sie mit Ihrem Wissen anfangen können, zeigt Ihnen auch diesmal das Begleitprogramm zu diesem Kurs.

Laden Sie nun den "Assembler-Kurs 3" von Diskette und genießen Sie, was Sie bisher gelernt haben.

(wk)

Teil 4 – Magic Disk 04/90

Heute behandeln wir zwei neue Assemblerbefehle und die Erstellung eines Programm-ablaufplans. Auf Diskette wird außerdem noch die Zeropage-Adressierung erklärt.

JMP
(JuMP to adress)

Zuletzt wurde die bedingten Verzweigung erläutert. Der JMP-Befehl hängt nicht von solchen Bedingungen ab. Wenn das Programm diesen Befehl erreicht, wird sofort zur angegebenen Adresse verzweigt.

Beispiel: JMP $2000
springt zur Adresse $2000 Sicher haben Sie schon die Ähnlichkeit mit der GOTO-Anweisung in BASIC erkannt. Die Sprungadresse des JMP-Befehls kann auch in indirekter Form vorliegen, z.B.: JMP ($2000).

Eine Hexadezimalzahl besteht immer aus einem höherwertigen Byte (bei der Zahl $2000) ist dies $20) und einem niederwertigen Byte (hier:$00). Das höherwertige Byte entspricht also den ersten beiden Stellen, das niederwertige den letzten beiden Stellen der Hexzahl. Bei dem letzten Beispiel wird nun nicht zur Adresse $2000 verzweigt. Stattdessen werden die Speicherstellen $2000 und $2001 als "Sprungvektoren" behandelt. Enthält z.B. die Adresse $2000 das Byte $05(niederwertiges Byte) und $2001 den Wert $10 (höherwertiges Byte), so springt das Programm nach $1005. Diese Adressierungsart des JMP-Befehls wird jedoch nur selten verwendet.

JSR
(Jump to SubRoutine)

Ein Unterprogramm ist eine selbstständige Gruppe von Befehlen, die vom Hauptprogramm getrennt sind. Nach Aufruf des Hauptprogramms wird es abgearbeitet, bis ein JSR-Befehl erreicht wird (z.B.: JSR $2000). Nun wird in das Unterprogramm verzweigt. Es werden jetzt alle Befehle ausgeführt, bis das Programm auf ein RTS trifft. Daraufhin wird in das Hauptprogramm zurückgesprungen.

Selbstverständlich kann auch aus dem Unterprogramm ein weiteres Unterprogramm aufgerufen werden (Verschachtelung von Unterprogrammen). Die Kombination JSR ... RTS entspricht unseren bekannten BASIC-Befehlen GOSUB ... RETURN.
Der Vorteil eines Unterprogramms liegt darin, daß es von beliebig vielen Stellen aus dem Hauptprogramm aufgerufen werden kann. So müssen mehrmals benötigte Routinen nicht immer wieder neu geschrieben werden. Dadurch fällt weniger Programmierarbeit an und es wird Speicherplatz gespart.
Wie schon erwähnt, liegt einer der größten Nachteile der Assemblerprogrammierung in der Übersichtlichkeit längerer Programme. Ohne ein geeignetes Hilfsmittel wird das Programmieren zu einem reinen Glücksspiel (Fehlersuche in Assembler ist weniger spaßig!). Doch zum Glück gibt es die Programmablaufpläne. Bei diesen wird durch einfache Symbole der ganze Aufbau eines Programms verdeutlicht. Die vorhin eingeblendete Grafik verdeutlicht die Symbole.
Die Ablaufrichtung eines PAP geht immer von oben nach unten vor, wenn sie nicht durch Pfeile gekennzeichnet wird.

Laden Sie nun den ASSEMBLER-KURS 4 aus dem GAME-MENÜ. Dort erhalten Sie anhand von Beispielprogrammen weitere Informationen.

(rt/wk)

Teil 5 – Magic Disk 05/90

Das Thema, das wir diesmal behandeln, wird Sie sicherlich nicht in Verzückung geraten lassen, aber es ist leider notwendig: Das Rechnen in ASSEMBLER.
Wie Sie sicher schon ahnen, wird die Sache etwas komplizierter als in BASIC. Mit einer Zeile, wie C = A + B ist es in ASSEMBLER leider nicht getan. Um diesen Kursteil nicht unnötig zu komplizieren, werden wir uns heute nur mit der Addition und Subtraktion von ganzzahligen Werten (genannt: Integers) beschäftigen. Dabei kommt man allerdings um die Kenntnis der Flaggen des Statusregisters (bekannt aus Kursteil 3) nicht herum. Bevor wir munter addieren

und subtrahieren, sollten Sie vielleicht doch erstmal einen Blick auf die dualen Zahlendarstellungen des C64 werfen. Das Verlassen des erlaubten Zahlenbereiches kann einige unschöne Nebeneffekte haben.
Unser Computer kennt die ganzzahlige Werte von -128 bis +127 bei 8- Bit-Zahlen und von -32768 bis +32767 bei 16- Bit-Zahlen. Positive Dualzahlen sind für uns ja kein Problem, aber wie stellt man den negative Zahlen dar?
Das soll an dem Beispiel der Zahl -18 erklärt werden:
Die Zahl +18 wird bekanntlich als 00010010 kodiert. Um die negative Zahl zu erhalten, muß man das Komplement der Zahl bilden (d. h. die Zahl wird invertiert: aus jeder 0 wird eine 1 und umgekehrt) und +1 dazu addieren. Das Komplement von +18 ist 11101101 . Nun muß noch +1 dazu addiert werden und man kommt zu 11101110, was der Zahl -18 entspricht.
Das äußerste rechte Bit (Bit 7) nennt man das Vorzeichenbit, wobei 0 eine positive und 1 eine negative Zahl kennzeichnet. Diese Darstellungsform, bei der die negativen Zahlen umgewandelt werden, die positiven aber wie gewohnt aussehen, nennt man ZWEIERKOMPLEMENT.

ADC
(ADd with Carry)

Wenn zwei Zahlen addiert werden, dann muß die eine im Akkumulator stehen, die zweite kann unmittelbar oder als Inhalt einer Speicherstelle gegeben sein. Das Ergebnis der Rechen-operation steht wieder im Akkumulator. Dies gilt auch für die Subtraktion. Das Ergebnis sollte aus dem Akkumulator schnell in eine Speicherstelle geschrieben werden, da es sonst beim weiteren Programmablauf verlorengehen könnte.

Unser neuer Befehl ADC addiert aber nicht nur zwei Zahlen, sondern auch noch den Wert des Carry-Bits (was uns bei 8- Bit-Additionen gar nicht recht sein kann) . Daher sollte vor dem ADC zuerst der Befehl CLC stehen, mit dem das Carry-Bit gelöscht wird. Nun erhalten wir auf jeden Fall des richtige Ergebnis.
Dualzahlen werden fast wie Dezimalzahlen addiert:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 und zusätzlich enthält hier das Carry-Bit den Übertrag 1, der bei der nächsten Stelle nicht berücksichtigt wird.

Ein kleines Beispiel:

```
CLC
LDA #$5D          01011101    (Dez. 93)
ADC #$0E        + 00001110    (Dez. 14))
STA $1500         01101011    (Dez. 107)
```

Das Ergebnis #$6 B (=107) steht jetzt im Akkumulator und in der Speicherstelle $1500.
Ein weiteres Rechenbeispiel:

Dezimal	**Binär**
115	01110011
+ 14	+ 00001110
129	10000001

Hier ist offensichtlich etwas schiefgegangen. Das Ergebnis hätte +129 lauten sollen; herausgekommen ist aber 10000001, was der Zahl -127 im Zweierkomplement entspricht. Das diese Rechnung nicht korrekt ist, liegt am Verlassen des erlaubten Zahlenbereiches. Der Computer gibt uns hier zwar keine Fehlermeldung aus, er warnt uns jedoch vor dem Ergebnis,

indem er die V-Flagge setzt. Diese Overfolw-Flagge wird immer dann gesetzt, wenn ein unbeabsichtigter Übertrag in das Vorzeichenbit stattfand. Die gesetzte N-Flagge (Vorzeichen-flagge) sagt uns, daß das Ergebnis eine negative Zahl ist. Die V-Flagge wird eigentlich automatisch gesetzt oder gelöscht. Dennoch gibt es den Befehl CLV, mit dem man die V-Flagge auch selbst wieder löschen kann, und somit das Warnsignal unterdrückt.
Bisher wurde nur die Addition von 1-Byte-(8-Bit) Zahlen geschildert.
Die Addition von 2-Byte-(16-Bit) Zahlen wird noch etwas komplizierter, da der Akkumulator nur 8 Bits aufnehmen kann.
Des Rätsels Lösung: Man addiert zunächst nur die niederwertigen Bytes der 16- Bit-Zahlen. Sollte dort ein Übertrag über das 8 . Bit hinaus vorliegen, so gelangt dieser ins Carry-Bit. Nun werden die beiden höherwertigen Bytes addiert, und der Inhalt des Carry-Bits hinzugezählt. Das Abspeichern der beiden Bytes muß ebenfalls getrennt erfolgen. Erwähnt sei noch, daß bei 16- Bit-Zahlen das Bit 15 das Vorzeichen enthält.

Beispiel für eine 16-Bit-Addition:
$1000/$1001 enthalten die erste Zahl $1100/$1101 enthalten die zweite Zahl $1200/$1201 sollen das Ergebnis enthalten Das Programm dafür sieht folgendermaßen aus:

```
CLC
LDA $1000
ADC $1100       ;Addition des niederwertigen
STA $1200       ;Bytes
LDA $1001
ADC $1101       ;Addition des höherwertigen
STA $1201       Bytes ohne CLC
```

SBC (SuBtract with Carry) Dieser Befehl, wer hätte es gedacht, subtrahiert zwei Zahlen voneinander.

Das Dumme ist nur: Unser Prozessor kann überhaupt nicht subtrahieren, er kann nur addieren! Um dennoch Zahlen voneinander abziehen zu können, wendet er einen Trick an. Eine der beiden Zahlen wird einfach zu einer negativen Zahl gemacht und dann werden beide Zahlen addiert. Diese Umwandlung nimmt uns der Befehl SBC jedoch schon ab. Es wird zu dem ersten Operanden nicht nur das Zweierkomplement der zweiten Zahl, sondern auch noch das Komplement des Carry-Bits addiert. D.h. wenn das Carry-Bit das Ergebnis bei der Subtraktion nicht verfälschen soll, muß es zuvor gesetzt sein. Dazu dient der Befehl **SEC** (SEt Carry), der vor jedem SBC stehen sollte.

Ein Beispiel:

```
SEC
LDA #$7D
SBC #$0A
STA $1500
```

Per Hand sieht die Rechnung so aus:

Dezimal	**Binär**
125	01111101
- 10	+ 11110110
115	(1) 01110011

Das Ergebnis $73 (115) steht im Akku und in der Speicherstelle $1500. Wie Sie sehen, erhalten wir das richtige Ergebnis. Das Vorzeichen der Zahl ist positiv, da die N-Flagge nicht gesetzt ist. Das gesetzte "9. Bit" steht im Carry und wird bei der 8- Bit-Subtraktion nicht benötigt. Auch bei der Subtraktion signalisiert die V-Flagge ein unbeabsichtigtes Verlassen des Zahlenbereichs. Der SBC-Befehl hat also große Ähnlichkeit mit dem ADC, und so ist es nicht verwunderlich, daß

eine 16- Bit-Subtraktion einer 16- Bit-Addition auch sehr verwandt ist.

Beispiel für eine 16-Bit-Subtraktion:
$1000/$1001 enthalten die erste Zahl $1100/$1101 enthalten die zweite Zahl $1200/$1201 sollen das Ergebnis enthalten Das Programm sieht folgendermaßen aus:

```
SEC
LDA $1000
SBC $1100       ;Subtraktion der
STA $1200       ;niederwertigen Bytes
LDA $1001
SBC $1101       ;Subtraktion der höher-
STA $1201       ;wertigen Bytes ohne SEC
```

Abschließend möchte ich noch zeigen, daß es dem Programmierer selbst überlassen bleibt, ob er die Zahlen wirklich als vorzeichenbehaftet ansieht, wie es eigentlich vorgesehen ist, oder ob er die Zahlen als vorzeichenlos betrachtet.
Dazu ein Beispiel:

Dezimal	**Binär**
- 6	11111010
+ 2	+ 00000010
- 4	(1) 11111100

Das Ergebnis ist bei vorzeichenbehafteten Zahlen richtig, da -6($ FA) und -4($ FC) im Zweierkomplement gegeben sind. Nun vergessen Sie mal das Zweierkomplement und betrachten Sie die Zahlen als vorzeichenlos. Dann hätten wir $FA+$02=$FC (also dezimal 250+2=252) gerechnet. Auch dieses Ergebnis ist korrekt. Dieses Beispiel zeigt, daß es am Programmierer liegt, welches der beiden Ergebnisse er beabsichtigt.

BCD-Darstellung (Binary Coded Decimals)
Da die Befehle ADC und SBC sowohl im Binär-, als auch im Dezimalmodus arbeiten, muß es wohl Befehle geben, die zwischen den beiden Modi umschalten.
Wenn es nicht vom Programm geändert wurde, ist automatisch der Binärmodus angeschaltet.

CLD schaltet den Binärmodus an
SED schaltet auf Dezimalmodus um

Bei diesen "binärkodierten" Dezimalzahlen gibt es nur die Ziffern 0 bis 9 . Es wird also jede Dezimalstelle für sich getrennt kodiert. Um diese 10 Ziffern kodieren zu können, benötigt man 4 Bits, mit denen bis zu 16 Kombinationen möglich wären. Sechs der Kombinationsmöglichkeiten bleiben daher ungenutzt. Die Dezimalzahl 0 wird als 0000, die 9 als 1001 kodiert. Die BCD-Darstellung der Zahl 128 lautet z.B.:

0001	0010	1000	**binär**
1	2	8	**dezimal**

Da jede Ziffer 4 Bits in Anspruch nimmt, fasst man je zwei BCD-Ziffern in einem Byte zusammen. Die Zahl 128 liegt folgendermaßen im Speicher: 0000000100101000. Die BCD-Zahlen seien hier jedoch nur am Rande erwähnt, da Sie nur sehr selten verwendet werden. Rechnen werden wir nicht mit diesen Zahlen.
Das richtige Verständnis für die diesmal erlernten Befehle werden Sie erst durch eigenes Ausprobieren erhalten. Also nichts wie hingesetzt und losprogrammiert. Auch diesmal gibt es selbstverständlich wieder ein Begleitprogramm zu dem Kurs auf Diskette. Es nennt sich

"ASSEMBLER-KURS 5" und enthält neben einer Joystickabfrage in ASSEMBLER auch noch einige nützliche Anwendungen der neuen Befehle.

Zusammenfassung der Befehle:

ADC (ADd with Carry)	Zwei Zahlen werden zusammen mit dem Inhalt des Carry-Bits addiert. Das Ergebnis steht wieder im Akkumulator. Dieser Befehl gilt sowohl für die binäre, als auch für die dezimale (BDC) Addition. Der ADC-Befehl kann die Flaggen N, V, Z und C beeinflussen.
SBC (SuBtract with Carry)	Zwei Zahlen werden subtrahiert, indem das Zweierkomplement der zweiten Zahl zu der ersten dazu addiert wird. Das Ergebis der Operation steht im Akkumulator. Auch SBC kann sowohl im Binär-, als auch im Dezimalmodus arbeiten.
CLC (CLear Carry)	Löscht die Übertragungsflagge. Dieser Befehl sollte immer vor einem ADC benutzt werden.
SEC (SEt Carry)	Setzt die Übertragsflagge. SEC sollte vor jedem SBC stehen.
CLD (CLear Decimal mode)	Löscht die Dezimalflagge und schaltet somit den Binärmodus ein.
SED Decimal mode)	Setzt die Dezimalflagge auf 1. Dadurch werden alle Rechen-(SEt operationen im Dezimalmodus ausgeführt.
CLV (CLear oVerflow flag)	Löscht die Überlaufflagge.

(Ralf Trabhardt/wk)

Kurs Teil 6 – Magic Disk 06/90

Diesmal geht es um die logischen Verknüpfungen AND, ORA und EOR, von denen Sie vielleicht einige schon von BASIC aus kennen. Außerdem werden Sie die Befehle BIT und NOP kennenlernen.

AND (AND with accumulator)	Die logischen Befehle verknüpfen den Akkumulatorinhalt bitweise mit den angegebenen Daten. Die AND-Verknüpfung läuft nach folgendem Muster ab:

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

Wie Sie sehen, erhält man als Ergebnis nur dann eine "1", wenn beide Operanden an dieser Stelle das Bit auf 1 gesetzt haben.

Beispiel für eine AND-Verknüpfung

```
LDA #$9A
AND #$D1
STA $1000
```

Der Akku wird mit dem Wert $9A (=144) geladen und mit der Zahl $D1(=209) UND-Verknüpft. Das Ergebnis der Verknüpfung steht wieder im Akku und kann von dort in Speicherstelle $1000 abgespeichert werden. Dasselbe Ergebnis erhielte man selbstverständlich auch durch:

```
LDA $1001
AND $1002
STA $1000
```

wenn vorausgesetzt wird, daß in Speicherstelle $1001 der Wert $9A und in $1002 die Zahl $D1

steht.
Führen wir nun die bitweise Berechnung durch, um zu sehen, welcher Wert in Speicherstelle $1000 abgelegt wird:

$9A	10011010
AND $D1	11010001
	10010000

Wenn Sie genau hinsehen, wird Ihnen auffallen, daß überall dort, wo in dem zweiten Operanden eine 0 auftaucht, auch das Ergebnis eine 0 enthält. Da haben wir auch schon das häufigste Einsatzgebiet des AND-Befehls gefunden, nämlich das gezielte Löschen von Bits. Ein weiteres Beispiel soll dies verdeutlichen. Hier verknüpfen wir eine beliebige Zahl mit dem Wert $F0 (=240). Für jede Bitposition an der ein X steht, kann entweder eine 1 oder eine 0 eingesetzt werden. Dies spielt für unser Beispiel keine Rolle:

$9A	XXXXXXXX
AND $F0	11110000
	XXXX0000

An unserem Ergebnis sieht man, daß die untersten vier Bits gelöscht wurden, während die obersten Bits ihren alten Wert beibehalten haben.
Der AND-Befehl beeinflußt die Flaggen N (wenn im Ergebnis Bit 7 gesetzt ist) und Z (falls das Ergebnis gleich Null ist).

ORA (inclusivE OR with Accumulator)

Die Verknüpfung verläuft nach folgendem Schema:

0 ORA 0 = 0 0 ORA 1 = 1 1 ORA 0 = 1 1 ORA 1 = 1	Man erhält als Ergebnis eine "1", wenn das Bit des ersten Operanden oder das Bit des zweiten Operanden an dieser Stelle auf 1 gesetzt ist.

Ansonsten ist ORA dem AND-Befehl sehr ähnlich. Er beeinflußt sogar dieselben Flaggen (N-Flag und Z-Flag). Ein Beispiel für eine ORA-Verknüpfung:

```
LDA #$9A
ORA #$D1
STA $1000
```

Auch hier wollen wir die Verknüpfung bitweise nachvollziehen:

$9A	10011010
ORA $D1	11010001
	11011011

Im Unterschied zu AND dient der ORA-Befehl dem gezielten Setzen von Bits. Als Beispiel möchte ich wieder ein beliebiges Byte mit dem Wert $F0 ODER-Verknüpfen:

$9A	XXXXXXXX
ORA $F0	11110000
	1111XXXX

Das Ergebnis zeigt, daß die obersten vier Bits auf 1 gesetzt wurden, während die anderen gleichgeblieben sind. Bei der ODER-Verknüpfung sind es jetzt die Einsen des Operanden $ F0, die eine Veränderung des Akkuinhaltes bewirken.
Vergleichen Sie das Ergebnis mit dem Beispiel der AND-Verknüpfung, so stellen Sie fest, daß dort die Nullen des zweiten Operanden für das Ergebnis entscheidend waren.

EOR (Exclusive OR with accumulator)

Die Verknüpfung verläuft nach folgendem Schema:

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

Man erhält an einer Bit-Stelle nur dann eine "1", wenn das Bit des ersten Operanden oder das Bit des zweiten Operanden an dieser Position auf 1 gesetzt ist.

Schaut man sich die Verknüpfungstabelle des EXKLUSIV-ODER an, so fällt ein Unterschied zum ORA direkt ins Auge: Nur wenn genau ein Operand das Bit auf 1 hat, wird auch im Ergebnis dieses Bit gesetzt, aber nicht, wenn beide Operanden eine gesetztes Bit haben. Ein Beispiel für eine EOR-Verknüpfung:

```
LDA #$9A
EOR #$D1
STA $1000
```

Bitweise ergibt sich folgendes Ergebnis:

```
   $9A    10011010
EOR $D1   11010001
          01001011
```

Aus diesem Ergebnis läßt sich auf Anhieb kein besonderer Sinn für diesen Befehl erkennen. Aber was passiert, wenn der zweite Operand mit dem ersten operanden übereinstimmt? Richtig, das Ergebnis der EOR-Verknüpfung wäre Null (d.h. das Zero-Flag wäre gesetzt). EOR kann also für Vergleiche benutzt werden, denn bei Ungleichheit der Operanden erhielte man ein Ergebnis ungleich Null.
Der EOR-Befehl hat aber noch ein weiteres Einsatzgebiet. Um dies zu veranschaulichen, verknüpfen wir den Akkuinhalt $9A mit $FF:

```
   $9A    10011010
EOR $FF   11111111
          01100101
```

Und siehe da, wir erhalten als Ergebnis das Komplement des vorherigen Akkuinhaltes. Man muß also, um das Komplement einer Zahl zu bekommen, diese mit EOR $FF verknüpfen.

BIT (BIT Test)

Eigentlich hat der Bit-Befehl nicht sehr viel mit den logischen Verknüpfungen zu tun. Man kann ihn eher zu den Vergleichsbefehlen zählen.

Mit BIT führt man eine UND-Verknüpfung zwischen dem Akkuinhalt und der angegebenen Speicherstelle durch. Dabei wird aber der Akkumulatorinhalt nicht verändert. Das Ergebnis dieser Verknüpfung wird in der Zero-Flagge festgehalten, da diese auf 1 gesetzt ist, falls das Ergebnis 0 war. Andernfalls ist das Zero-Flag gelöscht.
Die Adressierung des BIT-Befehles kann nur absolut erfolgen:

```
LDA #$9A
BIT $1500
```

Bei diesem Beispiel wollen wir annehmen, daß in Speicherstelle $1500 der Wert $D1 abgelegt ist. Dann lautet die logische Verknüpfung wie beim AND-Befehl:

```
      $9A    10011010
AND $D1      11010001
             10010000
```

Der Akkumulatorinhalt wird diesmal jedoch nicht verändert, und das Z-Flag ist gelöscht, da das Ergebnis ungleich 0 ist.

NOP
(No OPeration)
Der NOP-Befehl tut nämlich nichts. Aber wozu kann man einen Befehl gebrauchen, der nur zwei Taktzyklen abwartet und ein Byte Speicherplatz beansprucht, aber sonst keine Wirkung hat? Man nimmt ihn als Platzhalter innerhalb des ASSEMBLER-Programmes.

Wenn man bei seinem Programm plötzlich feststellt, daß man noch einen Befehl vergessen hat, muß man normalerweise den gesamten Programmcode im Speicher verschieben. Hat man aber z.B drei NOP's hintereinander geschrieben, so kann man diese einfach mit einem JMPoder JSR-Befehl überschreiben. Durch einen solchen "Notausstieg" hat man die Möglichkeit, weitere Befehle einzufügen.
Wie immer finden Sie auf der Diskette ein Begleitprogramm mit vielen nützlichen Tips. Starten Sie dazu das Programm "ASSEMBLER-KURS 6" aus dem Gamesmenü.

Ralf Trabhardt/(wk)

Teil 7 – Magic Disk 07/90

Bevor wir uns den ROM-Routinen des C64 zuwenden, werden noch die vier Bit-Verschiebebefehle ASL, LSR, ROL und ROR erklärt. Diese Befehle benötigen wir erst im Kursteil 9 bei der Multiplikation und Division von Ganzzahlen.
Trotzdem sollen diese Befehle schon jetzt behandelt werden, da Sie in der Folge 9 genug Probleme mit dem Verständnis der Fließkomma-Zahlen haben werden und nicht noch zusätzlich mit neuen Befehlen konfrontiert werden sollen. Die Verschiebebefehle

ASL
(Arithmetic Shift Left)
Bei diesem Befehl wird der gesamt Akkuinhalt, oder der Inhalt der angegebenen Speicherstelle bitweise nach links verschoben. Das Bit 7 wird in das Carry-Bit ausgelagert, in Bit0 des Bytes wird eine 0 eingefügt.

```
          7 6 5 4 3 2 1 0
Carry <= * * * * * * * * <= 0
```

Ein einmaliges Linksverschieben entspricht einer Verdoppelung der ursprünglichen Zahl. Beispiel:

```
LDA #$19
ASL
STA $1000
```

$19 entspricht der binären Bitfolge: 00011001. Ein ASL führt nun zu 00110010($32) und einer 0 im Carry-Bit. Das Ergebnis der Verschiebung wird in der Speicherstelle $1000 abgelegt. Aber neben der Möglichkeit der Verdoppelung einer Zahl kann man mit ASL auch bequem die Bits einer Speicherstelle über das Carry-Bit testen. Es werden das Negativ und das Zero-Flag

beeinflußt. Der Befehl ASL bietet folgende Adressierungsmöglichkeiten:

ASL	Akku-orientiert
ASL $1000	absolut
ASL $1000,X	absolut X-indiziert
ASL $FA	Zeropage-absolut
ASL $FA,X	Zeropage-abs. X-indiziert

LSR
(Logical Shift Right)

LSR bildet das Gegenstück zu ASL. Das betroffene Bit wird nun nach rechts geschoben. Dabei gelangt das bit 0 in das Carry-Bit und in Bit7 wird eine 0 eingesetzt.

```
    7 6 5 4 3 2 1 0
0 => * * * * * * * * => Carry
```

Beispiel:

```
LDA #$19
LSR
STA $1000
```

Aus der Bitfolge 00011001 wird durch LSR ein 00001100. Außerdem enthält das Carry-Bit jetzt die links aus dem Byte geschobene 1. Das Ergebnis wird wieder in der Speicherstelle $1000 abgelegt.
LSR hat in diesem Fall die dezimale Zahl 25($19) ganzzahlig halbiert. Das gesetzte Carry-Bit signalisiert dabei, daß ein Rest aufgetreten ist (25:2=12 Rest 1). LSR erlaubt die selben Adressierungsarten und beeinflußt die gleichen Flaggen wie der ASL-Befehl.

ROL
(ROtate Left)

Der ROL-Befehl hat große Ähnlichkeit mit dem ASL, nur daß das Byte jetzt nicht mehr nur verschoben wird, sondern um 1 Bit rotiert. Wie auch bei ASL landet das Bit7 im Carry. Das Carry-Bit wird anschließend in Bit0 übertragen. Somit ist das Bit 7 schließlich in Bit 0 gelangt.

```
  7 6 5 4 3 2 1 0
Carry <= * * * * * * * * <= Carry
```

Beispiel:

```
LDA #$B1
ROL
STA $1000
```

In den Akku wird die Bitkombination 10110001 geladen. Nach dem ROL befindet sich der Wert 01100011($63) im Akku, der in die Speicherstelle $1000 abgelegt wird. Das Carrybit enthält ebenfalls eine 1.

ROR
(ROtate Right)

Wie nicht anders zu erwarten war, ist ROR wieder das Gegenstück zu ROL. Jetzt wird das Byte nicht um ein Bit nach links, sondern nach rechts rotiert. Es befindet sich daher Bit0 im Carry. Der Inhalt des Carry-Bits rotiert nun in Bit7 hinein.

```
  7 6 5 4 3 2 1 0
Carry => * * * * * * * * => Carry
```

Beispiel:

```
LDA #$B1
ROR
STA $1000
```

Auch für ROR wird das Beispiel durchgeführt. Aus 10110001 wird nach ROR die Bitfolge

11011000. Das Carry-Bit ist auch in diesem Beispiel gesetzt. Die zuletzt besprochenen Befehle lassen dieselben Adressierungsarten wie der ASL-Befehl zu, und auch sie verändern die Flaggen N und Z.

ROM-Routinen

Unsere Assembler-Programme stehen immer im RAM (Random Access Memory), dem Lese-/Schreib-Speicher. In das ROM (Read Only Memory) können wir nicht hineinschreiben, es kann nur gelesen werden.
Folglich sind die oben erwähnten ROM-Routinen einige gegebene Unterprogramme, die wir in unsern eigenen Assembler-Programmen verwenden können. Der Sinn der Benutzung dieser ROM-Routinen soll Ihnen am Beispiel des "Bildschirmlöschens" verdeutlicht werden:

Bisher benötigten wir eine Schleife, die den gesamten sichtbaren Bildschirmbereich mit dem SPACE-Zeichen " " vollgePOKEd hat. Es war also ein erheblicher Programmieraufwand erforderlich. Nun geht das Ganze aber viel kürzer. Wir springen einfach mit JSR ein ROM-Unterprogramm an, das das Bildschirmlöschen für uns übernimmt. Ein "Clear Screen" wird jetzt einfach mit JSR $E544 ausgeführt.
Übrigens werden alle ROM-Routinen mit einem JSR angesprungen, da diese allesamt mit RTS enden.
Einige ROM-Routinen benötigen vor ihrem Aufruf gewisse Informationen, ohne die sie nicht fehlerfrei arbeiten können. Zum Beispiel braucht die Routine zum Setzen des Cursors die Zeilenposition im X-Register und die der Spalten im Y-Register.

```
LDX #$00
LDY #$08
JSR $E50C
```

Dieses Programm setzt den Cursor in der Zeile 0 auf die Spalte 8. Das folgende Programm gibt den ASCII-Text an der aktuellen Cursorposition aus

```
LDA #$00
LDY #$C1
JSR $AB1E
```

Der auszugebende Text für diese Routine muß im Speicher an der Stelle $C100 beginnen, und mit einem Byte $00 enden.
Als Zeiger auf diese Textstelle fungieren hierbei der Akku (niederwertiges Byte der Adresse) und das Y-Register (höherwertiges Byte). Wie Sie einen ASCII-Text direkt im Speicher ablegen können, verrät Ihnen das Handbuch Ihres Assemblers. Oft gibt es dafür einen Befehl der Art

```
C100   .ASC "beispieltext"
```

Der Text liegt dann folgendermaßen im Speicher:

```
C100   42 45 49 53 50 49 45 4C          beispiel
C108   54 45 58 54 00 00 00 00          text
```

Eine weitere wichtige Routine ist die Tastaturabfrage JSR $FFE4. Hierbei steht der ASCII-Code der gedrückten Taste im Akkumulator. Das Beispielprogramm:

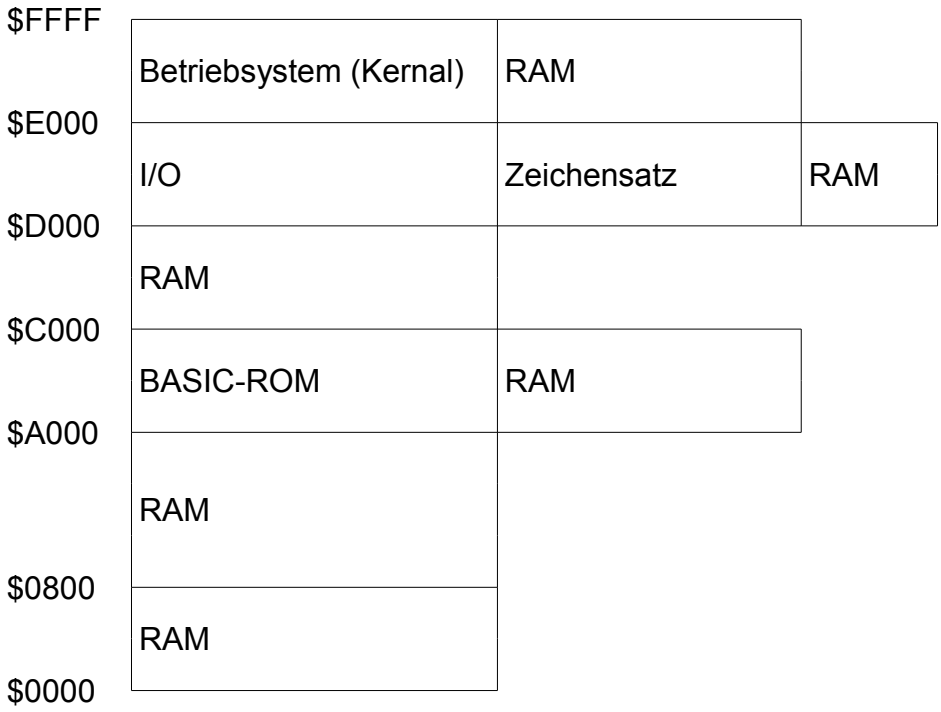
```
C000   JSR $FFE4
C003   BEQ $C000
```

überprüft, ob irgendeine Taste gedrückt wurde. In BASIC hätten wir

```
10 GET A$
20 IF A$ = "" THEN 10
```

geschrieben. Natürlich kann ich Ihnen in diesem Kurs nicht alle verfügbaren ROM-Routinen

vorstellen. Wenn ich aber Ihr Interesse daran geweckt habe, dann sollten Sie sich ein Buch zulegen, das "dokumentierte ROM-Listings" enthält. Aber wo genau liegt eigentlich der ROM-Bereich im Speicher?
Wir haben bis jetzt von $AB1E bis $E50C aufgerufen. Bei $C000-$CFFF liegt aber ein RAM-Bereich, den wir schon oft genutzt haben. Es muß daher mehr als nur einen ROM-Bereich geben.
In der Tat gibt es 3 unterschiedliche ROM-Bereiche. Von $A000-$BFFF liegt das ROM des BASIC-Interpreters. Dieser Interpreter sorgt dafür, daß Sie gleich nach dem Einschalten des Computers das BASIC verfügbar haben.
Zwischen $D000-$DFFF befinden sich die I/ O-Bausteine, die für die Ein- und Ausgaben verantwortlich sind.
Ab $E000 bis zum Ende des Speichers $FFFF liegt das Betriebssystem-ROM. Speicher-belegungsplan des C64:



Der mit "Zeichensatz" gekennzeichnete Bereich enthält den Zeichensatz-Generator (character generator), den ich im Kursteil 8 mit der Erstellung selbstdefinierter Zeichen näher erläutern werde.
Wenn Sie den Speicherbelegungsplan ansehen, stellen Sie fest, daß einige Bereiche mit derselben Adresse mehrfach belegt sind. So liegt z.B. von $E000 bis $FFFF nicht nur das Betriebssystem,"darunter" befindet sich auch noch ein RAM-Bereich. Ähnlich verhält es sich mit dem Bereich von $D000-$DFFF, der sogar dreifach belegt ist.
Aber wie kann man zwischen den mehrfach belegten Bereichen auswählen?
Dazu dient die Speicherstelle $01, der sogenannte Prozessorport. Je nachdem, welcher Wert in dieser Speicherstelle steht, verändern sich die Zugriffsmöglichkeiten auf den Speicher.
Als Assembler-Programmierer könnten wir zum Beispiel mit

```
LDA #$36
STA $01
```

das gesamte BASIC abschalten und das darunterliegende RAM für unsere Zwecke nutzen. Wir hätten somit einen durchgehenden RAM-Speicher von $0800 bis $CFFF geschaffen. Leider können wir aber nicht mehr auf die ROM-Routinen des BASIC-Interpreters zugreifen.

```
LDA #$37
STA $01
```

stellt den Ausgangszustand wieder her und schaltet das BASIC-ROM an.

$01	**$A000-$BFFF**	**$D000-$DFFF**	**$E000-$FFFF**
$37	BASIC	I/O	ROM
$36	RAM	I/O	ROM
$35	RAM	I/O	RAM
$34	RAM	RAM	RAM
$33	BASIC	Charset	ROM
$32	RAM	Charset	ROM
$31	RAM	Charset	RAM
$30	RAM	RAM	ROM

Diese Tabelle des Prozessorports zeigt Ihnen die Möglichkeiten auf. Auf den ersten Blick wirkt das mit der Mehrfachbelegung und dem Umschalten zwischen Bereichen alles etwas umständlich. Anders geht es aber nicht.
Mit unserem maximal 2 Byte langem Adressteil (z.B. LDA $FFFF) können wir nur den Bereich von $0000-$FFFF, also 64 KByte adressieren. Würde man alle mehrfach belegten Bereiche hintereinander schreiben, käme man aber auf einen Speicherbereich von 64 KByte + 24 KByte =88 KByte.
Als praktischen Anschauungsunterricht laden Sie nun am Besten das Beispielprogramm aus dem Menü: ASSEMBLER-KURS 7.

(rt/wk)

Kurs Teil 8 – Magic Disk 08/90

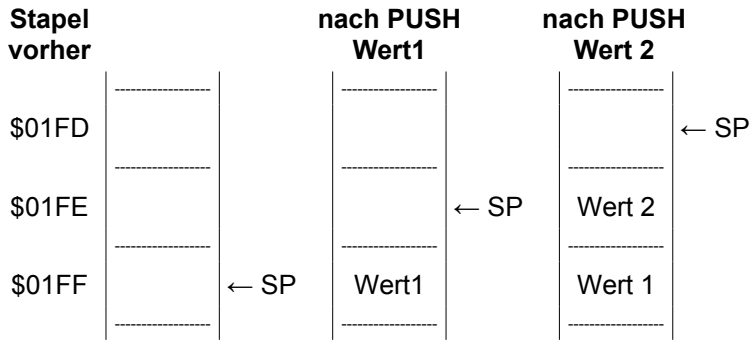
Der Stapel
Der Stapel ist der Speicherbereich von $0100 bis $01FF, der direkt von der CPU verwaltet wird. Der Zugriff auf diese "Page 1" ist daher sehr schnell.
Ein Stapel wächst von oben nach unten, d. h. in unserem Fall, daß zuerst ein Byte in $01FF, das nächste in $01FE usw. abgelegt wird. Bemerkenswert ist beim Stapel, daß er nach dem LIFO-Prinzip (last in first out) arbeitet. Das zuletzt auf den Stapel gebrachte Byte muß also als erstes wieder heruntergeholt werden, wenn die darunterliegenden Bytes gelesen werden sollen. Wenn Sie Schwierigkeiten haben, sich das LIFO-Prinzip zu verdeutlichen, dann denken Sie sich den Stapel doch einfach als einen Stapel von Getränkekisten. Dann wird Ihnen klar, daß Sie keinesfalls eine der unteren Kisten herausziehen können. Die zuletzt auf den Stapel gebrachte Kiste, das ist die Kiste ganz oben auf dem Stapel, muß als erste entfernt werden, um an die darunterliegende Kiste heranzukommen. Aber woher weiß der Computer, welcher Wert auf dem Stapel der oberste ist?
Dazu benötigen wir einen Zeiger (Pointer), der auf die jeweilige Spitze des Stapels zeigt, den sogenannten Stapelzeiger (Stackpointer). Unser Stapelzeiger ist 8 Bits breit, also eigentlich um ein Byte zu klein für die Adressierung des Stapelbereichs ($01FF bis $0100). Sicher fällt Ihnen auf, daß das höherwertige Byte ($01) über dem gesamten Stapelbereich gleich bleibt. Es ist daher unnötig, diesen Wert immer im Stapelzeiger mit abzuspeichern. Stattdessen begnügt man sich damit, daß der Stapelzeiger das niederwertige Byte der Spitze des Stapels enthält. Die effektive Adresse, auf die der Zeiger deutet, kann man sich folglich mit $0100+ (Inhalt des Stapelzeigers) selbst errechnen. Gültige Werte sind für den Stapelzeiger zwischen $00 und

$FF.
So, jetzt wissen wir, wie ein Stapel funktioniert, aber wozu man einen Stapel benötigt, das wissen wir noch nicht. Es gibt drei wichtige Anwendungsgebiete für einen Stapel:

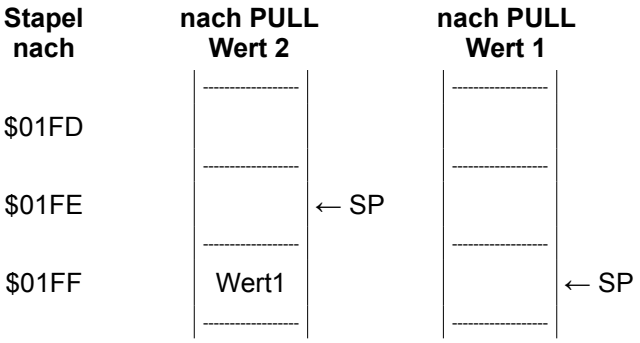
1. Er übernimmt die Verwaltung der Rücksprungadressen bei Unterprogrammaufrufen. Diese Eigenschaft wird später noch näher erläutert werden.
2. Zur Zwischenspeicherung von Daten bei Interrupts.
3. Kurzzeitige Sicherung von Daten.

Diesen letzte Punkt wollen wir uns nun genau ansehen.

Die Stapeloperationen
Im Grunde genommen benötigt man nur zwei verschiedene Stapelbefehle, nämlich einen, um etwas auf den Stapel zulegen (PUSH) und einen, um einen Wert vom Stapel zu holen (PULL, POP).



Wenn noch keine Stapeloperationen durchgeführt wurden, zeigt der Stapelzeiger SP auf die Speicherstelle $01FF. Nun wird WERT1 auf den Stapel gelegt, und zwar an der Adresse, auf die der Stapelzeiger weist. Nach diesem Schreibzugriff auf den Stapel wird der Stapelzeiger um 1 erniedrigt und zeigt so auf die nächste freie Speicherstelle des Stapels ($01FE) . Eine weiters PUSH-Anweisung legt den WERT2 auf die Spitze des Stapels (Top of Stack), was der Position des Stapelzeigers entspricht. Anschließend erniedrigt sich der Stapelzeiger wieder um 1.



Während bei PUSH-Befehlen des Stapels zu beobachten war, daß sich der Stapelzeiger erst nach dem Schreiben in die Stapelspeicherstelle erniedrigt hat, stellt man nun fest, daß bei PULL-Anweisungen der Stapelzeiger vor dem Lesezugriff inkrementiert wird. Das ist auch nötig, da der Stapelzeiger immer auf die nächste FREIE Speicherstelle des Stapels zeigt. Wird der Stapelzeiger zuvor inkrementiert, dann weist er nun auf das zuletzt auf den Stapel geschobene Element. Dieser Wert kann nun vom Stapel geholt werden. Das obige Beispiel verdeutlicht, daß der zuerst auf den Stapel gelegte Wert (Wert1) erst als letzter vom Stapel geholt werden kann.

Die PUSH-Befehle

PHA Mit diesem Befehl wird der Akkuinhalt auf den Stapel geschoben.

(PusH Accumulator)	Anschließend wird der Stapelzeiger um 1 erniedrigt. Der Akkuinhalt bleibt dabei, ebenso wie die Flaggen, unverändert.
PHP (PusH Processor status)	Anstelle des Akkuinhaltes wird jetzt das gesamte Statusregister des Prozessors mit allen Flaggen auf den Stapel gelegt. Danach wird der Stapelzeiger um 1 erniedrigt.

Die PULL-Befehle

PLA (PuLl Accumulator)	Diese Anweisung ist das Gegenstück zu PHA. Der Stapelzeiger wird zuerst um 1 erhöht. Nun wird der Inhalt der Speicherstelle, auf die der Stapelzeiger deutet, in den Akku eingelesen. Neben dem Akkuinhalt können sich auch die Flaggen N und Z ändern.
PLP (PuLl Processor status)	Der Stapelzeiger wird inkrementiert und der aktuelle Stapelwert wird in das Prozessor-Statusregister übertragen. Dabei ändern sich selbstverständlich alle Flaggen.

Zusätzlich zu den Stapeloperationen gibt es auch noch zwei Transferbefehle, die sich direkt auf den Stapelzeiger beziehen.

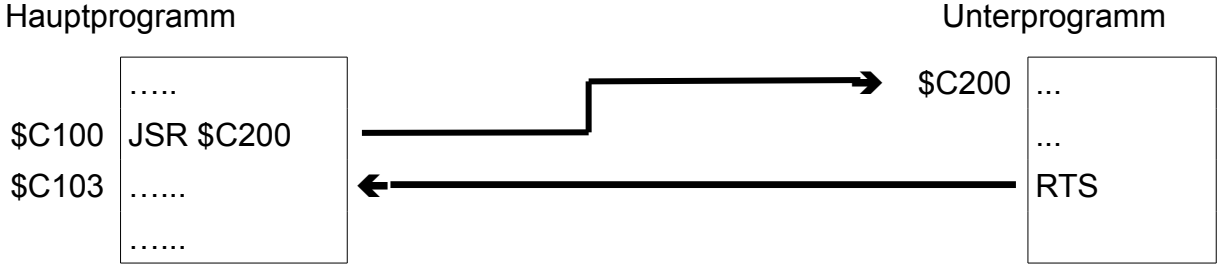
Die Stapel-Transferbefehle

TSX (Transfer Stackpointer to X)	Wie der Name der Anweisung schon aussagt, überträgt der Befehl den Stapelzeiger in das X-Register. Somit kann die Adresse, auf die der Stapelzeiger weist, ermittelt werden. Das X-Register kann dementsprechend Werte zwischen $00 und $FF enthalten.
TXS (Transfer X to Stackpointer)	Dieser gegenteilige Befehl zu TSX ermöglicht eine direkte Einfluß-nahme des Programmierers auf den Stapelzeiger, indem der Wert des X-Registers in den Stapelzeiger übertragen wird.

Alle oben genannten Befehle können nur impliziert adressiert werden, d.h. sie haben keinen Adressteil.

Der Einsatz des Stapels bei Unterprogrammen

Bisher haben wir es als selbstverständlich erachtet, daß unser Assembler-Programm nach einem RTS bei der nächsten Anweisung nach dem JSR im aufrufenden Programm fortfährt. Das Programm hat sich also die Stelle, von der aus der Unterprogrammsprung erfolgte, gemerkt. Womit wir wieder beim Stapel wären.
Der Prozessor muß also vor jeder Verzweigung in ein Unterprogramm die momentane Adresse auf den Stapel speichert, was der Adresse die Instruktion nach dem JSR-Befehl entspricht. Diese Adresse muß nach einer speziellen Regel "gestackt" werden, da unser Stapel bekanntlich immer nur ein Byte aufnehmen kann, eine Adresse jedoch aus 2 Bytes besteht. Zuerst wird das höherwertige Byte mit einem PUSH auf den Stapel gelegt und anschließend der Stapelzeiger um 1 vermindert. Erst jetzt wird das niederwertige Byte auf den Stapel gebracht. Erneut muß der Stapelzeiger dekrementiert werden.



Der Stapel nach JSR $ C200:

Stapel	Wert	
$01FD		← SP
$01FE	$03	
$01FF	$C1	

Nach einem RTS-Befehl läuft der umgekehrte Prozess ab. Zuerst wird der Stapelzeiger um 1 erhöht und das niederwertige Byte vom Stapel geholt. Im Anschluß daran holt man das höherwertige Byte vom Stapel und der Stapelzeiger wird wieder erhöht. nun kann die aktuelle Adresse wieder zusammengesetzt werden und das Hauptprogramm fährt mit seiner Abarbeitung an der gewünschten Stelle fort.
Es fällt uns auf, daß diese abgelegten Rücksprung-Adressen denselben Stapelbereich belegen, den auch der Programmierer zur vorübergehenden Datenspeicherung benutzen kann. Da der Speicherbereich des Stapels begrenzt ist, müssen wir auch die Struktur dieser automatisch ablaufenden Prozesse der Unterprogramm-Verwaltung kennen.

(rt/ wk)

Teil 9 – Magic Disk 09/90

Multiplikation und Division
Die Addition und Subtraktion von ganzen Zahlen mit ADC und SBC war noch recht einfach. Leider gibt es aber keine Assemblerbefehle wie MULT oder DIV, die unsere Berechnungen übernehmen könnten.
Wenn wir eine Multiplikation durchführen wollen, dann müßten wir eigentlich ein Programm dafür schreiben, was allerdings eine ziemliche Quälerei wäre, da wir das Problem über Additionen lösen müßten.
Glücklicherweise gibt es auch für dieses Problem eine geeignete ROM-Routine ($B357). Der Faktor muß dafür in den Speicherstellen $28 und $29, der zweite Faktor in $71/$72 bereitgestellt werden. Das Ergebnis dieser 16-Bit-Multiplikation erhalten wir im X-(niederwertiges Byte) und Y-Register (höherwertiges Byte).
Dummerweise existiert keine entsprechende Routine für die Division. Hier tritt außerdem noch das Problem auf, daß beim Dividieren zweier ganzer Zahlen in den seltensten Fällen das Ergebnis eine ganze Zahl sein dürfte. Jetzt kommen wir um die gebrochenen Zahlen nicht mehr herum.

Die Fließkommazahlen (Floating Point,FLP)
Gebrochene Zahlen werden im allgemeinen als Fließkommazahlen bezeichnet. Fließkommazahlen bestehen immer aus drei Teilen: der Mantisse, der Basis und dem Exponenten. Betrachten wir zunächst eine dezimale FLP-Zahl: 4500 ist darstellbar als 4.5*10↑3, was der Darstellung 4.5 E3 entspricht (4.5 ist die Mantisse,3 der Exponent und 10 die Basis). Die Zahl 0.045 ließe sich auch als 4.5*10↑-2(4.5 E-2) schreiben. Beachten Sie bitte, daß beide Zahlen auf dieselbe Mantissenform gebracht wurden und sich lediglich noch im Exponenten unterscheiden. Das haben wir durch ein Links- bzw. Rechtsverschieben des Dezimalpunktes erreicht.
Bei dezimalen FLP-Zahlen ist das alles noch recht einfach, aber wie kann man binäre FLP-Zahlen in ein einheitliches Format bringen, so daß alle Zahlen dieselbe Anzahl von Bytes haben?
Betrachten wir das Problem anhand eines Beispiels:

Die Dezimalzahl 50.125 soll in eine binäre FLP-Zahl umgewandelt werden. Die Umwandlung erfolgt in 5 Schritten:

1. Vorkommateil umwandeln
   Der Vorkommateil wird als Integerzahl behandelt und wie gewohnt umgerechnet

   ```
   50 : 2 = 25   Rest: 0      niederw. Bit
   25 : 2 = 12   Rest: 1
   12 : 2 =  6   Rest: 0
    6 : 2 =  3   Rest: 0
    3 : 2 =  1   Rest: 1
    1 : 2 =  0   Rest: 1
   ```

   Das Ergebnis :110010

2. Nachkommateil umwandeln
   Die Stellen nach dem Binärpunkt haben die Wertigkeiten 2↑-1,2↑-2 usw.

   Bei der Berechnung muß daher die Dezimalzahl immer mit 2 multipliziert werden, bei auftretenden Vorkommastellen wird das jeweilige Bit gesetzt.

   ```
   0.125 * 2 = 0.25    Vorkommastelle: 0
   0.25  * 2 = 0.5     Vorkommastelle: 0
   0.5   * 2 = 1       Vorkommastelle: 1 n.Bit
   ```

   Das Ergebnis: .001

3. Normalisierung der Mantisse
   Unter Normalisierung versteht man das Anpassen der Mantisse an ein bestimmtes Format. Der Binärpunkt muß soweit verschoben werden, bis er links genau neben der höchstwertigen binären 1 steht.

   Vorherige Mantisse: 110010.001
   Normalisierte Mantisse: 0.110010001 * 2↑6

   In unserem Bespiel mußte der Binärpunkt um 6 Stellen nach links verschoben werden, was durch eine Multiplikation mit 2↑6 ausgeglichen werden muß, damit das Ergebnis nicht verfälscht wird.
   Die Mantisse ist nun in der richtigen Form und wir haben auch unseren binären Exponenten (69 gefunden. Die binäre Basis ist logischerweise 2.

4. Umwandlung des Exponenten
   Zu unserem Exponenten 6 wird nun noch die Zahl 128 hinzuaddiert, damit es möglich ist, auch negative Exponenten, die bei der Normalisierung der Mantisse entstehen können, darzustellen. Bei der Rückumrechnung (Binär -> Dezimal) ist daher darauf zu achten, daß vom binären Exponenten 128 abzuziehen ist.

   ```
       6
   + 128
   -----
     134  (binär: 10000110)
   ```

   Ergebnis: Mantisse: 0.110010001
   Exponent: 10000110

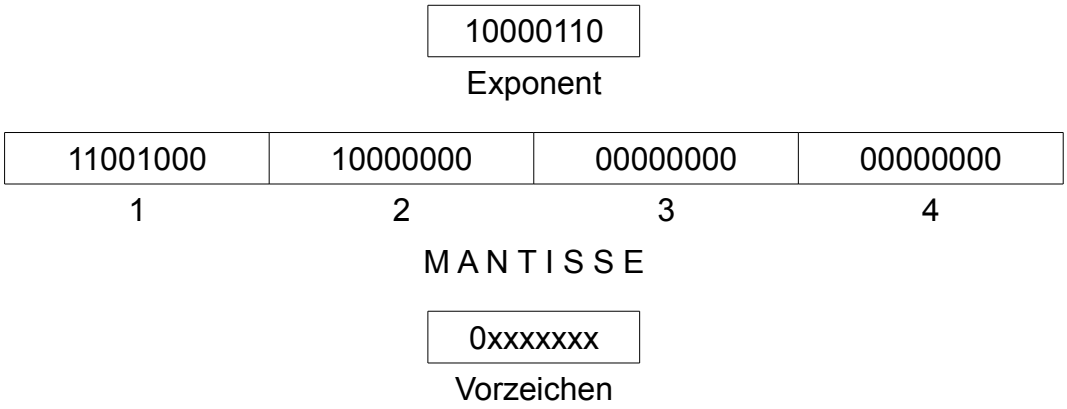
   Nun muß nur noch festgelegt werden, in wie vielen Bytes diese Zahl abgespeichert werden soll.

5. Floatingpoint-Zahlen des C64 Dieser Computer kennt zwei verschiedene Fließkommaformate, die sich hauptsächlich durch die Art der Speicherung des Vorzeichens unterscheiden. Die eine wird nur innerhalb der beiden Fließkommakkumulatoren verwendet, die andere beim Abspeichern der Ergebnisse im Arbeitsspeicher.

a) In den beiden Fließkommaakkumulatoren (abgekürzt: FAC und ARG) werden alle Berechnungen von Fließkommazahlen mit Hilfe von ROM-Routinen durchgeführt. Der FAC belegt in der Zeropage den Bereich von Speicherstelle $61 bis $66, ARG von $69 bis $6E. Welche Aufgabe die einzelnen Bytes haben, können Sie der nachfolgenden Tabelle entnehmen.

	FAC	ARG
Exponent	$61	$69
Mantisse 1	$62	$6A
Mantisse 2	$63	$6B
Mantisse 3	$64	$6C
Mantisse 4	$65	$6D
Vorzeichen	$66	$6E

Schauen wir nun, wie die FLP-Zahl aus unserem Beispiel im FAC oder ARG abgelegt wird. Der Exponent erhält 8 Bits, während sich die Mantisse auf 32 Bits ausbreiten darf. Die normalisierte Mantisse wird aber ab dem Binärpunkt linksbündig in die zur Verfügung stehenden Bytes eingetragen. Ist die Mantisse länger als 4 Bytes, so wird der überschüssige Teil einfach weggelassen. So entstehen u. a. die heißgeliebten Rechenfehler in einem Computer. Zusätzlich gibt es noch ein Byte, das das Vorzeichen enthält. Von diesem Byte ist jedoch nur das Bit 7 von Bedeutung (0 -> positiv ;1 -> negativ), die anderen Bits sind uninteressant. Format der Fließkommaakkumulatoren:



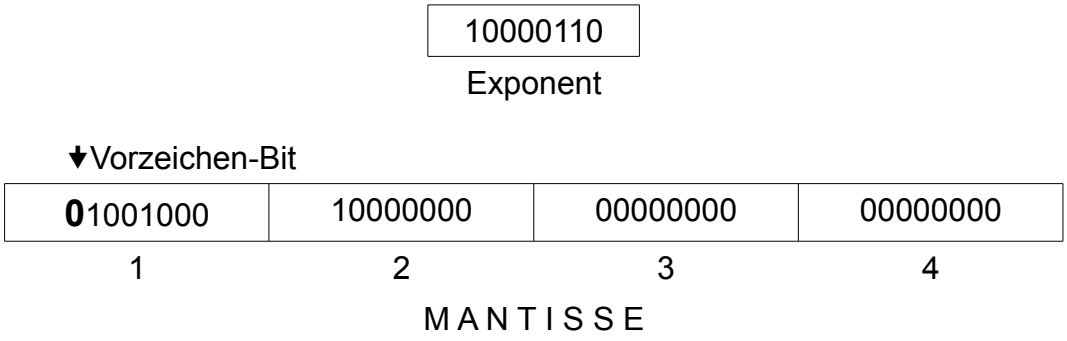
In hexadezimaler Form ergibt das: 86 C8 80 00 00 00

Sicher halten Sie die Benutzung eines ganzen Bytes für nur ein Vorzeichenbit für Verschwendung. In den Fließkommaakkumulatoren stört das jedoch niemanden, da es ohnehin nur zwei davon gibt. Außerdem erleichtert das getrennte Vorzeichen die Berechnung bei der Fließkommaarithmetik.
Hier nun eine kleine Auswahl der möglichen Rechenoperationen:

Rechenoperation		**Routine**
Addition	FAC : = ARG + FAC	$B86A
Subtraktion	FAC : = ARG - FAC	$B853
Multiplikation	FAC : = ARG * FAC	$BA2B
Division	FAC : = ARG / FAC	$BB12
Potenzierung	FAC : = ARG ↑ FAC	$BF7B

Vor dem Aufruf der Routinen müssen FAC und ARG natürlich die gewünschten Zahlen enthalten.
Wie Sie sehen, enthält der FAC immer das Ergebnis der Berechnungen. Spontan fallen uns zwei Probleme auf: Wie bekomme ich die Zahlen in den FAC oder ARG und wie kann ich das Ergebnis abspeichern, damit es erhalten bleibt? Auch dafür stehen uns wieder einige ROM-Routinen zur Verfügung, die Sie gleich noch kennenlernen werden. Zunächst jedoch zurück zu dem zweiten Fließkomma-Format von dem die Rede war.

b) Es handelt sich um das Format, in dem die Fließkommazahlen im Speicher abgelegt werden, entweder als Ergebnis einer Berechung, oder als Operanden für die Rechenoperationen in FAC und ARG. Ein Format, das sich auf den Arbeitsspeicher bezieht, sollte möglichst kurz und speicherplatzsparend sein. Ein eigenes Byte für ein Vorzeichen ist jetzt völlig ausgeschlossen. Die ganze Fließkommazahl wird nun durch einen kleinen Trick auf 5 Bytes beschränkt.
Wir wissen, daß rechts neben dem Binärpunkt der normalisierten Mantisse eine 1 steht (deshalb haben wir den Punkt ja dorthin gesetzt). Wenn uns sowieso bekannt ist, daß es sich um eine 1 handelt, dann brauchen wir sie doch nicht mehr mit abzuspeichern. Diese 1 muß lediglich bei allen Berechnungen berücksichtigt werden. Diese Bits nennt man Hidden Bits (versteckte Bits). Da das äußerst links stehende Bit der Mantisse nun frei geworden ist, können wir dort unser Vorzeichen unterbringen. Tatsächlich benötigen wir jetzt nur noch 5 Bytes für die Zahlendarstellung. Format im Arbeitsspeicher:



Hexadezimal erhält man: 86 48 80 00 00

ROM-Routinen zur Handhabung der Fließkommazahlen Innerhalb dieses Kursteiles kann ich Ihnen leider nur eine kleine Auswahl dieser ROM-Routinen vorstellen.

$B3A2	Wandelt eine Vorzeichenlose ganze Zahl (0...255), die im Y-Register steht, in eine FLP-Zahl im FAC um.
$BBA2	Lädt FAC mit einer FLP-Zahl aus dem Arbeitsspeicher, auf die der Zeiger Akku/Y-Register weist und wandelt sie in das benötigte 6-Byte-Format um.
$BA8C	Lädt ARG mit einer FLP-Zahl aus dem Arbeitsspeicher, auf die der Zeiger Akku/Y-Register weist und wandelt sie in das benötigte 6-Byte-Format um.
$BC0C	Kopiert den Inhalt des FAC in den ARG.

$BBFC	Kopiert den Inhalt des ARG in den FAC.
$B7F7	Wandelt den Inhalt des FAC in eine vorzeichenlose Integerzahl (0...65535) in den Speicherzellen $14/|$15 und Y-Register/ Akkumulator um.
$BDDD	Wandelt den Inhalt von FAC in eine ASCII-Zeichenkette und legt sie im Puffer am $0100 ab.
$BBD4	Wandelt den FAC (Rechenergebnis)| in eine 5-Byte-FLP-Zahl und legt sie im Speicher under der Adresse ab, die der Zeiger X-Register / Y-Register bildet.

Die USR-Funktion (User callable machine language SubRoutine)
Bisher waren wir es gewohnt, unsere ASSEMBLER-Programme von BASIC aus mit SYS-Befehlen aufzurufen. Die Parameterübergabe konnte nur durch das POKEn der Werte an die jeweilige Adresse erfolgen.
Wenn der zu übergebende Wert jedoch eine FLP-Zahl ist, dann müßte die Zahl in BASIC zerlegt werden, um sie dann stückweise in den FAC zu POKEn. Das wäre doch sehr umständlich. Speziell für das Bearbeiten von Fließkommazahlen gibt es daher die USR-Funktion. Der Vorteil dieser Funktion ist, daß der Inhalt der angegebenen Variablen automatisch in den FAC übertragen wird. Aber woher weiß der Computer , wo unsere ASSEMBLER-Routine beginnt? Bei der Ausführung von USR wird der indirekte Sprung JMP($0311) ausgeführt. Die Einsprungadresse des ASSEMBLER-Programms muß als Zeiger in den Adressen $0311 (niederwertiges Byte) und $0312 (höherwertiges Byte) vor dem Aufruf bereitgestellt werden. Unter anderem enthält das Programm "ASSEMBLER-KURS 9" auch ein Beispiel für die Benutzung von USR.
Im nächsten (und letzten) Kursteil geht es dann um den Umgang mit den Interrupts. Mich interessiert sehr Ihre Meinung, wie Ihnen der Kurs gefallen hat. Welche Themen wurden zu kurz behandelt, wo sehen Sie noch Probleme, was fanden Sie besonders gut?

Schreiben Sie mir doch bitte Ihre positive oder negative Kritik:

Ralf Trabhardt
Philippsbergstraße 45
6200 Wiesbaden

Also, bis zum nächsten Kursteil…

Kurs Teil 10 – Magic Disk 10/90

Wie versprochen, handelt der letzte Teil dieses Kurses von den Interrupts. Interrupt bedeutet soviel wie "Unterbrechung" des laufenden Programms. Der C64 reagiert auf verschiedene Arten von Interrupts, die man in Hard und Softwareinterrups unterteilen kann.

Hardware-Interrupts
Es gibt zwei unterschiedliche Arten von Hardware-Interrupts, den IRQ (interrupt request = Interrupt Anforderung) und den NMI (non maskable interrupt = Nicht maskierbarer Interrupt). Beide sind eigentlich Pins am Prozessor, an denen durch ein Impuls ein entsprechender Interrupt ausgelöst werden kann. Die Reaktion des Prozessors auf beide Interrupts ist etwas unterschiedlich.
Grundsätzlich sagt man, daß der NMI die höhere Priorität beider Interrupt-Arten hat und somit für die für das System wichtigen Aufgaben eingesetzt wird.

a. Ablauf eines NMI:

1. Der momentan zu bearbeitende Befehl des laufenden Programms wird noch ausgeführt.
2. Der Programmzähler (program counter), d. h. das Register, das immer auf den nachfolgenden Befehl eines Programms zeigt, wird auf den Stack geschoben. Dabei wird zuerst das höherwertige und dann das niederwertige Byte des Programmzählers abgelegt.
3. Das Statusregister, das alle Flags enthält, wird auf den Stack gebracht.
4. Der Inhalt der Adressen $FFFA (als niederwertiges Byte) und $FFFB (als höherwertiges Byte) wird zusammengesetzt und als neuer Programmzähler benutzt. Ab jetzt wird also das (unterbrechende) Programm an der Stelle ausgeführt, auf die der neue Programmzähler zeigt

b. Ablauf eines IRQ:

1. Zunächst scheint der IRQ genauso zu verlaufen, wie der NMI. Der augenblicklich zu bearbeitende Befehl des laufenden Programms wird noch vollständig abgearbeitet.
2. Anschließend erfolgt eine Überprüfung des Interrupt-Flags (Bit 2 des Statusregisters). Wenn das Bit gesetzt ist, dann wird die Interrupt-Anforderung einfach ignoriert, und der Prozessor fährt fort, als ob nichts geschehen wäre.Ist das Interrupt-Bit jedoch gelöscht, so wird der IRQ ausgeführt.
3. Der Programmzähler wird auf den Stack geschrieben.
4. Das Statusregister wird gestackt.
5. Nun wird das Interrupt-Flag gesetzt, so daß alle nachfolgenden Interrupt-Anforderungen ignoriert werden. Man bezeichnet diesen Vorgang als 'Sperren' des IRQ.
6. Der Inhalt der Adressen $FFFE und $FFFF wird zusammengesetzt und als neuer Programmzähler benutzt.

Der Hauptunterschied zwischen NMI und IRQ liegt darin, daß der IRQ maskiert (d.h. gesperrt) werden kann.
Wie oben bereits beschrieben, wird beim IRQ zuerst das I-Flag untersucht und erst daraufhin entschieden, ob ein Interrupt auszuführen ist, oder nicht. Dieses Interrupt-Flag des maskierten Interrupts IRQ kann vom Programmierer selbst direkt beeinflußt werden.

SEI (SEt Interrupt mask)	Mit dem Befehl SEI wird das I-Flag gesetzt und somit werden alle folgenden IRQs gesperrt.
CLI (CLear Interrupt mask)	Durch CLI kann dieser Zustand wieder aufgehoben werden, indem das I-Flag gelöscht wird.

Ein IRQ ist nun für den nachfolgenden Programmteil wieder zugelassen. Häufige Anwendung finden diese beiden Befehle auch beim Umschalten der Speicherkonfiguration durch den Prozessorport. Ohne das vorherige Abschalten der Interrupts mittels SEI kann es beim Ausschalten von ROM-Bereichen zum Systemabsturz kommen

Wodurch können die Interrupt IRQ und NMI ausgelöst werden? Selbstverständlich nur durch die Hardware des Computers:

IRQ: a) durch den Videocontroller VIC
b) durch den CIA-Baustein ($DC00)

NMI: a) durch den CIA-Baustein ($DD00)
b) durch die RESTORE-Taste

Der Software-Interrupt
Software-Interrupts werden durch den Befehl **BRK** (BReaK) ausgelöst. Das war einer der ersten Befehle, die wir in diesem Kurs kennengelernt haben. Nun wollen wir untersuchen, was genau passiert, wenn der Prozessor auf einen BRK-Befehl im Programm stößt:

1. Das Break-Flag (Bit 4 des Statusregisters) wird gesetzt.
2. Der Programmzähler wird auf den Stack gebracht.
3. Das Statusregister wird gestackt.
4. Nun wird das Interrupt-Flag gesetzt, so daß alle IRQs gesperrt werden.
5. Der Inhalt der Adressen $FFFE und $FFFF (dieselben Adressen wie beim IRQ) wird zusammengesetzt und als neuer Programmzähler benutzt.

Meistens wird der BRK-Interrupt von den Assembler-Monitoren dazu genutzt, um das laufende Programm abzubrechen und das Statusregister mit den aktuellen Flags anzuzeigen. Es wäre aber auch eine andere Nutzung dieses Interrupts möglich.
Der Software-Interrupt BRK wird also im Gegensatz zu den Hardware-Interrupts durch einen Assembler-Befehl ausgelöst. Egal, ob IRQ, NMI oder der Softwareinterrupt BRK. Eines haben alle gemeinsam. Sie lesen alle einen Zeiger aus ($FFFE / $FFFF bei IRQ, BRQ;$FFFA/ $FFFB bei NMI) der auf eine ROM-Routine weist.
Diese ROM-Routinen werden nun als Interruptprogramm abgearbeitet und führen am Ende einen unbedingten Sprung aus. Dieser unbedingte Sprung beruft sich auf einen Zeiger, der im RAM steht. Diese Zeiger sind:

für IRQ	$0314/$0315
für BRK	$0316/$0317
für NMI	$0318/$0319

Diese Zeiger sind schon mit einem bestimmten Wert initialisiert, so daß der Sprung ohne unser Eingreifen bestens funktioniert. Wie Sie schon ahnen werden, müssen wir diesen Zeiger lediglich auf unsere selbstgeschriebenen Routinen umbiegen, und schon haben wir eigene Interruptroutinen.

RTI – Das Ende der Interrupts:
Wenn das Programm durch einen Interrupt unterbrochen wurde und der Prozessor nun eine Interrupt-Routine bearbeitet, dann muß ihm auch irgendwie kenntlich gemacht werden, wann diese Unterbrechung beendet werden soll. Auch die Interruptroutinen benötigen daher ein Programmende. Für 'normale' Programme haben wir als Kennzeichnung für das Programmende immer den RTS-Befehl (ohne vorherigen JSR-Befehl) im Hauptprogramm benutzt, wodurch wir zurück ins BASIC kamen.
Der Befehl für das Ende unser selbstgeschriebenen Interrupt-Routine lautet **RTI** (ReTurn from Interrupt). Bei der Bearbeitung dieses Befehls läuft der umgekehrte Prozeß des Interrupt-Aufrufs ab.

1. Das Statusregister muß vom Stack geholt werden.
2. Der Programmzähler wird vom Stack geholt und wieder als aktueller Zähler benutzt.
3. An der Stelle, auf die der Programmzähler zeigt, wird das unterbrochene Programm fortgeführt.

Der Hauptunterschied zwischen RTS und RTI liegt darin, daß beim RTI zusätzlich zum Programmzähler auch noch das Statusregister vom Stack geholt werden muß. Ein Interrupt hat doch einige Ähnlichkeiten mit einem Unterprogrammaufruf.
Ein Unterprogramm wird jedoch von einer klar definierten Stelle im Hauptprogramm (durch den JSR-Befehl) aufgerufen. Die Interrupts NMI und IRQ werden jedoch hardwaremäßig ausgelöst, so daß die Unterbrechung des Hauptprogramms an jeder beliebigen Stelle stattfinden kann. Diese Eigenschaft erweckt den Eindruck, daß die Interruptroutine und das Hauptprogramm

parallel arbeiten, was aber natürlich nicht der Fall ist. In dem Begleitprogramm "ASSEMBLER-KURS10" werden wir eigene Interrupt-Routinen schreiben. Dabei geht es um die Benutzung des Systeminterrupts und der Interrupts durch den Videocontroller (Rasterzeilen-Interrupt). Dies war der letzte Teil des Assembler-Kurses. Ich hoffe, er konnte Ihnen einige Kenntnisse und Anregungen verschaffen.

rt/wk

Danksagung

Die Veröffentlichung der Magic Disk Kurse als PDF-Datei erfolgt mit schriftlicher Genehmigung der COMPUTEC MEDIA GmbH. Es dürfen keine kommerziellen Absichten verfolgt werden!

Mein Dank gilt ebenfalls Mirco Geldermann. Auf seiner Webseite www.magicdisk64.de ist es möglich die Diskettenimages der Magic Disk C64 Ausgaben offiziell herunterzuladen.