Assembler ist keine Alchimie — Teil 8

Irgendwann brauchen Sie sie bestimmt, die Rechnung mit Fließkommazahlen. Auch den Umgang mit logischen Ausdrücken wie AND, ORA oder EOR und wie man mit dem Schiebe-Befehl ASL multipliziert, sollen Sie beherrschen. Das alles erfahren Sie im achten Teil dieses Kurses.

Inzwischen wissen Sie ja, daß alle Daten im Computer im Binärformat enthalten sind. Wie man eine normale, ganze Zahl zur binären umrechnet, wurde schon im Grafik-Kurs (64'er, Ausgaben 4 und 5 von 1984) gezeigt. Da aber viele Leser dieses Assemblerkurses die genannten Ausgaben nicht besitzen, soll doch nochmal vorgestellt werden, welcher Rechenweg der einfachste ist. Als Beispiel nehmen wir die Zahl 1985. Man teilt diese Zahl so lange durch 2, bis das Ergebnis 0 wird. Jedesmal notiert man sich den Rest, der entweder 0 oder 1 sein kann:

1985	:2=	992	Rest 1
992	:2=	496	Rest 0
496	:2=	248	Rest 0
248	:2=	124	Rest 0
124	:2=	62	Rest 0
62	:2=	31	Rest 0
31	:2=	15	Rest 1
15	:2=	7	Rest 1
7	:2=	3	Rest 1
3	:2=	1	Rest 1
1	:2=	0	Rest 1

Auch wenn Sie es noch nicht erkennen: Da steht schon das binäre Ergebnis. Von unten nach oben gelesen, ist das nämlich der Rest:

111 1100 0001

Nun reden wir ja von Fließkommazahlen. Also verändern wir unser Beispiel noch etwas. Jetzt soll uns die Zahl 1985,125 interessieren. In der Ausgabe 6/84 haben Sie gelernt, daß man das Komma verschieben kann, um daraus beispielsweise 1,985125x 10^3 zu machen. Wir wollen uns das Verschieben des Kommas aber für etwas später aufheben und zunächst einmal außer dem schon umgewandelten Vorkammateil nun auch den Nachkommateil, also die 0,125, ins Binärformat übertragen.

Genauso, wie wir vorhin eine Kettendivision durch 2 verwendet haben, gebrauchen wir nun eine Kettenmultiplikation mit 2. Der gesamte Nachkommateil wird dabei verdoppelt. Entweder ergibt sich dabei eine Vorkommastelle (das ist dann immer eine 1) oder das Ergebnis bleibt kleiner als 1. Wenn sich bei einem solchen Rechenschritt keine Vorkommastelle ergibt, schreibt man an die entsprechende Nachkommastelle der Binärzahl eine 0, andernfalls eine 1. Es wird so lange verdoppelt, bis keine Nachkommastellen mehr zur Verfügung stehen. Das klingt ziemlich umständlich. Am besten sehen Sie sich das jetzt mal an unserem Beispiel an:

0,125 x 2 = 0,250
1. Nachkommastelle:0

Beim ersten Verdoppeln hat sich keine neue Vorkommastelle ergeben, deshalb ist die erste Nachkommastelle der Binärzahl eine Null.

0,25 x 2 = 0,5
2. Nachkommastelle:0

Auch beim zweiten Verdoppeln ermitteln wir keine neue Vorkommastelle, wodurch sich wieder eine Null als Nachkommastelle ergibt.

0,5 x 2 = 1,0
3. Nachkommastelle:1

Hier hat sich nun eine Vorkommastelle beim Verdoppeln gebildet: Daher taucht als 3. Nachkommastelle unserer Binärzahl eine 1 auf. Gleichzeitig war das die letzte Nachkommastelle, denn unsere Ausgangszahl weist nach dem Komma nun nur noch eine Null auf.

Zur Übung wollen wir noch eine andere Zahl mit Nachkommastellen ins Binärformat überführen, nämlich 0,1.

0,1x2 = 0,2	**1. Nachkommastelle:0**
0,2x2 = 0,4	**2. Nachkommastelle:0**
0,4x2 = 0,8	**3. Nachkommastelle:0**
0,8x2 = 1,6	**4. Nachkommastelle:1**

Jetzt läßt man — das habe ich beim ersten Beispiel noch nicht erwähnt — diese neue Vorkommastelle einfach weg und rechnet wieder mit den Nachkommastellen weiter:

0,6x2 = 1,2	**5. Nachkommastelle:1**
0,2x2 = 0,4	**6. Nachkommastelle:0**
0,4x2 = 0,8	**7. Nachkommastelle:0**
0,8x2 = 1,6	**8. Nachkommastelle:1**
0,6x2 = 1,2	**9. Nachkommastelle:1**

Das kommt Ihnen sicherlich von der 5. Verdoppelung her bekannt vor. Es zeigt sich, daß diese Rechnung nie aufgeht, weil sich eine periodische Zahl ergibt:

0,000 1100 1100 1100...

Das kann Ihnen öfters bei der Zahlenumwandlung passieren, daß ein endlicher Dezimalbruch in einen unendlichen periodischen Binärbruch übergeht.

Kehren wir zurück zu unserem ersten Beispiel, 1985,125. Die ganze Umwandlung (Vorkomma- und Nachkommaanteil) führte zu:

111 1100 0001,001

Der dritte Schritt der Verwandlung von der Dezimalzahl zum Binärformat (nach 1.=Vorkommaanteil umwandeln, 2.= Nachkommaanteil umwandeln) ist das sogenannte Normalisieren. Das ist einfach das Verschieben des Kommas nach links (wie in unserem Beispiel) oder rechts, so lange, bis vor dem Komma nur noch Nullen stehen und direkt hinter ihm eine 1. In der Ausgabe 2 (1985) haben wir gelernt, daß für jede Stelle, die das Komma nach links wandert, der Exponent um 1 höher wird. Unser Exponent ist im Moment noch Null (2° ist ja 1). Um also nach der Regel zu normalisieren, wird das Komma um 11 Stellen nach links verschoben. Der Exponent ist dann 1(dez) und unsere Zahl erscheint im neuen Gewand:

0.1111 1000 0010 01 E +1011

E +1011 heißt dabei Exponent, und wird im Binärformat dargestellt (10011 (bin.) ≙ 11 (dez.)). So weit, so gut. Alles bisher unternommene hat Allgemeingültigkeit. Von nun an aber müssen wir uns spezialisieren auf den Commodore 64 (im VC 20 und einigen anderen Computern ist es aber auch so). Der Exponent kann ja — je nach dem, ob das Komma nach links oder nach rechts zum Normalisieren verschoben wurde — positiv sein (wie bei unserem Beispiel) aber auch negativ. Im Commodore 64 wird zum Exponenten die Zahl 128 addiert. Das ist dann Schritt 4, der im Beispiel zu 138 führt, womit wir schon das Exponentenbyte fertig haben:

Exponent: dez.139 bin.1000 1011 hex.8B

Hätten wir einen negativen Exponenten erhalten, zum Beispiel 20, dann stünde im Exponentenbyte nun dez.108, beziehungsweise dasselbe im Binärformat.

Der Rest unserer Zahl, also die Mantisse, wird nun Schritt 5 unterzogen. Zunächst läßt man das Komma weg. Die Binärzahl wird dann auf 4 Byte linksbündig aufgeteilt. In unserem Beispiel erhalten wir so:

1111 1000	0010 0100
Byte 1	Byte 2
0000 0000	0000 0000
Byte 3	Byte 4

Wie Sie sehen, werden die unbenutzten Bits mit Nullen aufgefüllt. Was nun noch nicht berücksichtigt wurde, ist das Vorzeichen der Mantisse. Es ist im Beispiel noch nicht zu erkennen, ob wir +1985,125 oder -1985,125 vorliegen haben. Das gehen wir nun im letzten Schritt (Nummer 6) an. Im Commodore 64 gibt es zwei Möglichkeiten der Speicherung von Fließkommazahlen. Für Schritt 6 muß man sich entscheiden, wo man die Zahl haben will.

Im 6. Teil dieser Serie ist schon mal der FAC erwähnt worden, der Fließkomma-Akkumulator 1, welcher die Speicherstellen dez. 97 bis 102 ($61 bis $66) belegt. Ein zweiter Fließkomma-Akkumulator, AFAC oder ARG genannt, belegt die Plätze dez. 105 bis 110 ($69 bis $6E). Diese Akkumulatoren haben für die Fließkommarechnungen eine ähnliche Bedeutung wie der Akku für die 1-Byte-Rechnungen. Dort werden fast alle Ergebnisse abgelegt oder Zahlen abgerufen. Wir sehen, daß wir darin 6 Byte zur Verfügung haben. In Byte 97 liegt der Exponent in der von uns ermittelten Form. Byte 98 bis 101 sind die vier Mantissenbytes. Was ist in Byte 102? Das Vorzeichen! Bit 7 dieses Bytes ist 0, wenn eine positive, und 1 wenn eine negative Zahl vorliegt. Das galt für den FAC, wie Sie aus den Speicherstellen schon gesehen haben. Für den ARG ist das aber ganz genauso. Sehen wir uns nun in Bild 1 unsere Beispielzahl im FAC und im ARG nochmal an.

Im Bild ist auch angedeutet, daß die restlichen 7 Bit (Bits 0 bis 6) des Vorzeichenbytes keine Rolle spielen. Sie werden später direkt in diese Akkumulatoren hineinsehen und allerlei Bit-Müll darin finden. Lediglich Bit 7 ist für uns von Bedeutung.

Eigentlich ist das ja eine ganz schöne Verschwendung, von einem Byte wie diesem Vorzeichenbyte lediglich ein einziges Bit zu nutzen. Wenn eine beliebige Fließkommazahl irgendwo im Computer abgespeichert

FAC	$ 61 dez 97	62 98	63 99	64 100	65 101	66 102
ARG	$ 69 dez. 105	6A 106	6B 107	6C 108	6D 109	6E 110
INHALT BINÄR	1000 1011	1111 1000	0010 0100	0000 0000	0000 0000	0... ...
HEX.	8B	F8	24	00	00	
DEZ.	139	248	36	0	0	
BYTE Nr.	1	2	3	4	5	6
ERLÄUTERUNG	Exponent	MANTISSE				Vorzeichen

Bild 1. So sieht die Zahl 1985,125 komplex im FAC und ARG aus

wird, dann gilt ein anderes Format, das MFLPT-Format (von **M**emory-**FL**oating **P**oin**T**). Man speichert hier nur in 5 Byte. Das Vorzeichenbyte fällt weg. Wie aber merkt sich der Computer das Vorzeichen? Das ist ganz schlau eingefädelt: Es gibt nämlich in den 5 Byte (1 Exponentenbyte + 4 Mantissenbyte) ein überflüssiges Bit. Sie werden sich sicher erstaunt fragen, wo?

Erinnern Sie sich doch bitte zurück an den Schritt 3, das Normalisieren. Dort wurde so verfahren, daß rechts vom Komma eine 1 steht. Wenn da aber immer und ganz grundsätzlich diese 1 steht, dann muß man sie sich eigentlich gar nicht mehr besonders merken. Man kann — vorausgesetzt, man berücksichtigt diese 1 im Bit 7 des ersten Mantissen-Bytes immer bei den Rechnungen — das Bit für andere Zwecke verwenden: Also als Vorzeichenbit. Taucht hier also eine 0 auf, dann liegt eine positive Zahl vor, ist es aber eine 1, dann signalisiert diese eine negative Zahl. Für das MFLPT-Format muß in unserem Beispiel also Bit 7 des ersten Mantissenbytes gelöscht werden (1985,125 ist ja nun mal positiv) und die komplette Zahl sieht im MFLPT-Format so aus:

```
1000 1011    0111 1000    0010 0100    0000 0000    0000 0000
  Byte 1       ↑ Byte 2     Byte 3       Byte 4       Byte 5
Exponent     M    A    N    T    I    S    S    E
```

Der Pfeil weist auf das Vorzeichenbit. Man spricht hier auch vom »gepackten« Format. Damit das alles nun nicht nur graue Theorie bleibt und Sie auch aus eigenem Erleben diese Zahlenformate sehen können, wollen wir hier ein kleines Testprogramm ausprobieren. Es wird Ihnen auch später noch gute Dienste leisten können, wenn Sie mal irgendwelche Zahlen in das FLPT- (also FAC oder ARG) oder ins MFLPT-Format umrechnen müssen. Zu Fuß ist das ja — wie Sie nun wissen — ganz schön haarig! Wie so oft, besteht auch dieses Programm aus einem Basic-Teil, der die Benutzerführung übernimmt und zwei kleinen Maschinenroutinen, die per USR-Vektor angesprungen werden. In diesen Assembler-Programmteilen sind zwei Interpreter-Routinen verborgen, die nützlich und daher erklärenswert sind. Als Listing 1 ist das Basic-Aufrufprogramm abgedruckt.

Es fragt zunächst mal, ob der SMON eingeladen ist. Der wird nämlich aus dem Programm heraus angesprungen. Wird die Frage mit »J« beantwortet, dann zeigt sich ein kleines Menü, andernfalls ist das Programm beendet: Der SMON muß erst eingeladen werden.

Das Menü bietet 3 Optionen: Eine Zahl kann im FAC (Option 1), im ARG (Option 2) oder im MFLPT-Format ab Speicherstelle $6800 (Option 3) betrachtet werden.

Für Option 1 wird der USR-Vektor auf die Einsprungadresse des SMON gestellt und dann mittels USR-Kommando die Zahl Z in den FAC übergeben. Es schaltet sich dann der SMON ein, der nun mittels des Kommandos M 0061 den Inhalt des FAC als Hex-Zahlen zeigt.

Option 2 richtet zunächst den USR-Vektor auf ein kleines Assembler-Programm ab $6000, welches den FAC-Inhalt in den ARG schiebt, dann den USR-Vektor auf den SMON richtet und schließlich auch diesen einschaltet. Auch hier wird mit dem M-Kommando dann per M 0069 der ARG-Inhalt sichtbar. Option 3 richtet den USR-Vektor auf eine Maschinenroutine, die bei $6100 beginnt. Dort wird der FAC-Inhalt nach $6800 und folgende Speicherstellen verschoben und zwar ins MFLPT-Format. Anschließend erfolgt dann wieder das Ausrichten des USR-Vektors auf den SMON, Anschalten des SMON, wo man durch M 6800 den Inhalt ansehen kann.

Folgende Vorgehensweise empfehle ich Ihnen:

1. Einladen des SMON

2. Eintippen der beiden kleinen Assembler-Routinen mit Hilfe des SMON und Abspeichern (man kann einfach mit dem SMON-Kommando S»Programmname«,6000,610A speichern).

2a. Wenn Sie die beiden Routinen schon gespeichert vorliegen haben, dann laden Sie sie jetzt ein. Jedenfalls sollten Sie nach dem Laden beider Assembler-Programme (SMON und die beiden Routinen) ein NEW eingeben, so daß alle Zeiger zurückgestellt werden.

3. Erst jetzt Laden oder Eintippen des Basic-Aufrufprogrammes.

Wenn Sie nun das Testprogramm starten und zum Beispiel unsere Zahl 1985,125 eingeben, werden Sie folgendes finden:

Option 1:
M0061
:0061 8B F8 24 00 00 78 00 00

Option 2:
M0069
:0069 8B F8 24 00 00 78 D4 CE

Option 3:
M6800
:6800 8B 78 24 00 00 FF FF FF

Die Bytes, welche zu unserer Zahl gehören, sind unterstrichen. Sie können jeweils nach RUN/STOP noch mit dem SMON-Kommando $8B (oder eine andere Sie interessierende Hexzahl) eine Ausgabe im Binär- und im Dezimalformat erreichen.

So, nun aber endlich zu den beiden Assembler-Routinen. Zur Option 2 gehört das folgende, bei $6000 beginnende Programm:

```
5 REM*** TEST FUER FLPT UND MFLPT ***                    <162>
10 POKE 52,96:POKE 56,96:CLR:PRINT CHR$(14
   7)CHR$(17)CHR$(17)                                    <215>
15 PRINT"IST DER SMON EINGELADEN?":INPUT"J
   /N";A$:IF A$="N"THEN END                              <254>
20 PRINT CHR$(17)CHR$(17)"{3SPACE}FLPT IN
   FAC"TAB(25)"1":PRINT                                  <003>
30 PRINT"{3SPACE}FLPT IN ARG"TAB(25)"2":PR
   INT                                                   <030>
40 PRINT"{3SPACE}MFLPT AB $6800"TAB(25)"3"
   :PRINT:PRINT                                          <077>
50 GET A$:IF A$<"1"OR A$>"3"THEN 50                      <211>
60 PRINT"AUSWAHL",A$:PRINT:PRINT:INPUT"GEB
   EN SIE EINE ZAHL EIN";Z                               <107>
65 PRINT CHR$(147)                                       <142>
70 ON VAL(A$)GOTO 100,200,300                            <233>
100 REM***** FAC *****                                   <097>
110 POKE 785,0:POKE 786,192:REM USR-VEKTOR
    AUF SMON = $C000                                     <091>
120 A=USR(Z)                                             <205>
200 REM***** ARG *****                                   <213>
210 POKE 785,0:POKE 786,96:REM USR-VEKTOR
    AUF $6000                                            <011>
220 A=USR(Z)                                             <049>
300 REM***** MFLPT *****                                 <227>
310 POKE 785,0:POKE 786,97:REM USR-VEKTOR
    AUF $6100                                            <114>
320 A=USR(Z)                                             <150>
400 REM*********************************                 <138>
410 REM NACH MELDUNG DES SMON MIT DEN                    <042>
420 REM KOMMANDOS                                        <221>
430 REM (1)    M 0061                                    <212>
440 REM (2)    M 0069                                    <231>
450 REM (3)    M 6800                                    <241>
460 REM DEN MONITOR EINSCHALTEN. DIE                     <137>
470 REM EINGEGEBENE ZAHL IST DANN ALS                    <148>
480 REM HEX-BYTES SICHTBAR.                              <135>
490 REM*********************************                 <228>
```

© 64'er

Listing 1. Testprogramm für die beiden kleinen Assembler-Routinen. Die Bedienung ist im Artikel erklärt.

6000 JSR BC0C

$BC0C ist die erste Interpreter-Routine, die wir uns zunutze machen. Sie schiebt den Inhalt vom FAC in den ARG. Mehr dazu später.

```
6003   LDA   #00
6005   STA   0311
6008   LDA   #C0
600A   STA   0312
```

Damit haben wir den USR-Vektor auf $C000 gestellt.

600D JMP C000

Das war das Einschalten des SMON. Im Grunde genommen könnten wir uns das Stellen des USR-Vektors ersparen

Es ist aber sinnvoll — vor allem bei langen Programmen — wenn verstellte Vektoren nach Beendigung des Programmes auf einem definierten Wert stehen.

Nun noch die Routine für Option 3:

```
6100   LDX   #00
6102   LDY   #68
6104   JSR   BBD4
```

Auch das ist wieder eine Interpreter-Routine: Sie schiebt den FAC-Inhalt in einen Speicherbereich, dessen Startbyte durch die beiden Index-Register angegeben wird (X-Register für LSB, Y-Register für MSB, hier also 6800). Dabei wird die Zahl vom FLPT-Format in das MFLPT-Format umgewandelt.

Das Prográmmchen schließen wir ab mit einem Sprung zum Rest der ersten Routine:

6107 JMP 6003

Sehen Sie sich mal einige Zahlen im Fließkomma-Format an. Fast alle Operationen mit Zahlen vollführt unser Computer mit diesen Fließkommazahlen. Das ist dann beispielsweise der Grund dafür, daß aus einer Basic-Zeile wie der folgenden:

IF INT(X*10)=INT(ABS(X*10))THEN ...

auch bei positiven X-Werten (wo man mathematisch Gleichheit feststellt) manchmal die Bedingung als nicht erfüllt erkannt wird. X wird sofort als Fließkommazahl in den FAC gelegt, mit einer Fließkomma-Zehn multipliziert, der ABS-Wert wird ebenfalls per Fließkomma-Arithmetik ermittelt und so weiter. Dabei treten häufig Rundungsprobleme auf, wenn ein Zwischenergebnis mehr als 32 signifikante binäre Nachkommastellen aufweist (wie wir es ja zum Beispiel beim periodischen Binärbruch gesehen haben, der sich aus der simplen Dezimalzahl 0,1 ergibt). Das Rechnen mit Fließkommazahlen im Computer öffnet zwar einen ungeheuren Zahlenraum für unsere Anwendungen, es geht aber viel langsamer als die 2-Byte-Arithmetik. Immerhin müssen hier jedesmal 6 Byte (beziehungsweise 5 bei MFLPT) berücksichtigt werden. Ich glaube aber kaum, daß wir jemals in die Verlegenheit kommen werden, beispielsweise eine Fließkomma-Addition programmieren zu müssen. Eben weil unser C 64 fast alle Zahlenoperationen mit Fließkomma-Formaten durchführt, sind nahezu alle Eventualitäten schon als fertige abrufbare Programme im Interpreter enthalten. Wir müssen nur wissen, wie unsere Zahlen aussehen (das haben Sie nun ja gelernt) und wo und wie man sie für Operationen bereithält und wo und wie man die entsprechenden Routinen finden kann. Einen der wichtigsten Wege, unsere Zahlen ans Maschinenprogramm zu übergeben, haben Sie schon kennengelernt: Das Argument der USR-Funktion landet automatisch im FLPT-Format im FAC.

Die beiden ersten Interpreter-Routinen

Von nun an sollen nach und nach Interpreter-Routinen vorgestellt werden. Das ist allerdings nicht so einfach wie bei der Kernal-Sprungtabelle. Es gibt für die letzteren viele recht gut dokumentierte Listen. Für die Interpreter-Routinen ist kaum Literatur vorhanden. Will man ähnlich erfassen wie die Kernal-Routinen, dann muß man ROM-Listings wälzen und vor allem probieren, probieren ... Falls Sie also mal einen Fehler in der Beschreibung feststellen oder Dinge, die ich leer lassen muß, weil mir dazu die Erleuchtung noch nicht gekommen ist, selbst schon kennen, dann schreiben Sie mir. Gemeinsam haben wir vielleicht die Chance, auch die letzte im Interpreter versteckte Nuß noch zu knakken!

Nun also zur ersten schon verwendeten Routine:

Name	**MOVAF**
Zweck	Übertragen des FAC in den ARG
Adresse	$BC0C, dez. 48140
Vorbereitung	Wert in FAC
Speicherstellen	$61-66 FAC $69-6E ARG $6F, $70
Register	Akku, X-Register
Stapelbedarf	4

Diese Routine ist deswegen so wichtig, weil viele Rechenoperationen, die zwei Zahlen verknüpfen, zwischen dem FAC und dem ARG abgewickelt werden. Wenn Sie unser kleines Testprogramm mal mit der Option 2 laufen lassen und hinterher nicht nur mit M0069 in den ARG, sondern auch mit M0061 in den FAC hineinsehen, dann stellen Sie fest, daß der FAC-Inhalt noch immer vorhanden ist.

Allerdings ist das nicht immer der Fall. **MOVAF** rundet nämlich — wenn nötig — vorher noch den FAC-Inhalt, der dann natürlich anders aussieht.

Fast noch häufiger benutzt man die zweite Interpreter-Routine:

Name	**MOVMF**
Zweck	Übertragung von FAC in Speicher unter Umrechnung ins MFLPT-Format
Adresse	$BBD4 dez. 48084
Vorbereitung	Wert in FAC Zieladresse in X- und Y-Register (X = LSB, Y = MSB)
Speicherstellen	$61-66 FAC $70, $22, $23
Register	Akku, X- und Y-Register
Stapelbedarf	4

Außer den unter »Speicherstellen« genannten sind natürlich auch noch die Zieladresse und deren vier nachfolgende Bytes in die Routine einbezogen (das MFLPT-Format besteht ja aus 5 Byte). $22/23 ist ein für die Operation verwendeter Zeiger.

MOVMF wird häufig dann verwendet, wenn Werte, aus welchen Gründen auch immer, außerhalb der Fließkomma-Akkumulatoren gelagert werden müssen.

Es wird Ihnen vielleicht aufgefallen sein, daß im Gegensatz zur Beschreibung der Kernal-Routinen — die Rubrik »Fehler« fehlt. Der Grund ist, daß es keine solchen Sicherungen bei den Interpreter-Routinen gibt. Was passieren kann, ist unter bestimmten Bedingungen das Ansteuern von normalen Basic-Fehlermeldungen, die aber nicht immer den tatsächlichen Zustand wiedergeben. Wenn Ihnen mal bei der Programmierung mit Interpreter-Routinen Zweifel aufkommen, dann verfolgen Sie lieber den Programmweg mittels eines ROM-Listing und schalten Sie eigene Fehler-Routinen ein. Das war aber nur für die Fortgeschrittenen gesagt. Wir werden uns erst nach und nach dahin vortasten. Zunächst fehlen uns ja noch ein paar Assembler-Kenntnisse. Mit dem nächsten Abschnitt soll das besser werden.

Assembler-Befehle zum Beherrschen von Bits

Bevor wir damit anfangen, muß ich Ihnen noch sagen, daß Sie sich Zeit lassen sollten. Das Pensum, das ich Ihnen heute zumute, ist wirklich ganz gewaltig. Zunächst sehen wir uns die logischen Befehle AND, ORA und EOR an, dann Bit-Schiebereien mittels ASL. LSR, ROL und ROR sollen uns dann in der nächsten Ausgabe beschäftigen.

Fangen wir also mit AND an. AND verknüpft den Akku-Inhalt Bit für Bit mit dem angegebenen Wert nach den Regeln der logischen UND-Verknüpfung. Die Adressiermöglichkeiten dieses Befehls sind allerlei:

AND 6000	**absolut**
AND FE	**Zeropage absolut**
AND #07	**unmittelbar**
AND 6000,X	**absolut-X-indiziert**
AND 6000,Y	**absolut-Y-indiziert**
AND (FA,X)	**indiziert-indirekt**
AND (FB),Y	**indirekt-indiziert**
AND FE,X	**Zeropage-absolut-X-indiziert**

Damit haben wir eine ganze Menge an Möglichkeiten. Erinnern Sie sich noch an die Regeln einer UND-Verknüpfung? Wenn nicht, dann sehen Sie sich nochmal die Tabelle 1 an.

AND	0	1
0	0	0
1	0	1

Tabelle 1. Wahrheitstabelle zur AND-Verknüpfung

Sie erkennen, daß zwei miteinander AND-verknüpfte Bits nur dann als Ergebnis 1 haben, wenn in beiden Bits der Wert 1 steht. Man kann mittels AND ganz gezielt Bits löschen. Nehmen wir mal als Beispiel an, wir wollten geshiftete Zeichen (das sind die mit den Codes größer als 128) in normale Zeichen umwandeln. Dazu bringen wir die Zeichencodes in den Akku und löschen Bit 7. Übrig bleibt dann der Code für das ungeshiftete Zeichen. Für das Löschen von Bit 7 brauchen wir eine sogenannte UND-Maske, die dafür sorgt, daß alle anderen Bits unverändert bleiben. An den Stellen muß in dieser Maske also eine 1 stehen (denn 0 AND 1 ergibt 0, 1 AND 1 ergibt 1). Lediglich Bit 7 der Maske muß 0 sein. Die Maske muß also heißen:

0111 1111 $7F dez. 127

Nehmen wir an, im Akku befände sich der Code für ein geshiftetes A, also dez. 193 (binär 1100, 0001, $C1), dann ergibt die AND-Verknüpfung mit der Maske:

Akku	**1100**	**0001**	**Shift A**
Maske	**0111**	**1111**	
AND			
Jetzt im Akku	**0100**	**0001**	

Normales A (Code dez. 65, $41)

Man kann also, je nach Wahl der Maske, beliebige Bits löschen.

AND ist, je nach der gewählten Adressierungsart, ein 2- oder 3-Byte-Befehl. Weil das Ergebnis im Akku steht, können Flaggen beeinflußt werden. Die N- und die Z-Flagge reagieren auf das Ergebnis.

Im Gegensatz zu Basic, wo es nur eine ODER-Verknüpfung gibt, nämlich OR, existieren im Assembler zwei davon. Man unterscheidet ein »inklusives« und ein »exklusives« ODER. Die inklusive ODER-Verknüpfung des Akku mit den angegebenen Daten geschieht mit dem Assembler-Befehl ORA. ORA entspricht dem Basic-Befehl OR. Alle Adressierungsarten, die dem AND-Befehl offenstehen, können auch auf ORA angewendet werden. Wenn man Bits ORA-verknüpft, findet man folgende Ergebnisse:

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

Auch hier ist eine sogenannte Wahrheitstabelle recht einprägsam (siehe Tabelle 2).

ORA	0	1
0	0	1
1	1	1

Tabelle 2. Wahrheitstabelle zur ORA-Verknüpfung

Während man mit AND gezielt Bits löschen kann, ist es mit ORA möglich, Bits zu setzen. Auch dazu verwendet man eine Maske, die an allen Stellen, an denen Bits unverändert bleiben sollen, eine 0, sonst aber eine 1 enthält. Nehmen wir nochmal das Beispiel von vorhin und wandeln nun das ungeshiftete Zeichen in ein geshiftetes um. Wir müssen also Bit 7 wieder setzen: Da muß in der Maske dann eine 1 stehen. Alle anderen Bits bleiben unverändert, wenn die Maske dort

eine Null aufweist. Die Maske muß daher heißen:
1000 0000 $80 dez. 128

Im Akku soll das ungeshiftete B stehen (Code dez. 66, $42, bin. 0100 0010). Die Rechnung sieht dann so aus:

```
Akku      0100  0010  Code für B
Maske     1000  0000
  ORA
Jetzt im
Akku      1100  0010
```

Code für geshiftetes B.

Je nach Art der Maske kann man also ein oder mehrere Bits setzen. Im Beispiel ist auch der Einfluß dieses Befehls auf die Flaggen zu erkennen. Der Akku-Inhalt vor der ORA-Operation hatte kein Bit 7, also keine gesetzte N-Flagge. Danach ist Bit 7 gesetzt und die N-Flagge zeigt eine 1. Außer der N-Flagge kann — ebenso wie beim AND-Befehl — auch noch die Z-Flagge reagieren. ORA ist je nach Adressierungsart ein 2- oder 3-Byte-Befehl.

Während zwei Bits in der ORA-Verknüpfung eine 1 ergeben, wenn sie beide gesetzt sind oder eines von beiden, schließt die EOR-Verknüpfung den ersten Fall aus. EOR ist die exklusive ODER-Verknüpfung. Sie läßt sich sprachlich erfassen im »entweder ... oder ...«, also beispielsweise: Beim Roulette fällt die Kugel entweder auf Rouge oder auf Noir, beides zusammen ist nicht möglich. Die Regeln bei EOR sind also:

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

Eine Wahrheitstabelle dazu sehen Sie in Tabelle 3.

EOR	0	1
0	0	1
1	1	0

Tabelle 3. Wahrheitstabelle zur EOR-Verknüpfung

Wozu verwendet man EOR? Es fällt Ihnen vielleicht auf, daß wir die aus Basic bekannte NOT-Funktion nicht in Assembler vorliegen haben. Obwohl EOR einige viel weitergehendere Verwendungsmöglichkeiten aufweist als NOT (aber auf Boolesche Algebra wollen wir hier nicht eingehen), kann man es mit gleicher Wirkung einsetzen. Wir haben beispielsweise in den ersten Folgen dieses Kurses negative Zahlen durch Komplementieren erzeugt. Dabei sollte jedes Bit in sein Gegenteil verkehrt werden. Das wäre die Aufgabe einer NOT-Funktion. Durch ein EOR FF können wir dasselbe erreichen. Sehen wir uns wieder ein Beispiel an. Im Akku steht dez. 15 ($0F, bin. 0000 1111):

```
Akku      0000  1111
Maske     1111  1111   =$FF
  EOR
Jetzt im
Akku      1111  0000
```

Einerkomplement von dez. 15.

Auch EOR kann alle Adressierungsarten verkraften, die die beiden anderen logischen Assembler-Befehle erlauben. Je nach der gewählten Art liegt dann ein 2- oder 3-Byte-Befehl vor. Auch hier werden die Z- und die N-Flagge beeinflußt.

Das waren also die logischen Befehle. Leider ist hier nicht der geeignete Ort, die Vielseitigkeit, die damit möglich ist, deutlich zu machen. Wenn Sie sich dafür interessieren, sollten Sie mal etwas über Boolesche Algebra lesen oder eine Einführung in die mathematische Logik.

Um dieses Thema abzuschließen, soll noch erwähnt werden, daß der Basic-Interpreter so eingerichtet ist, daß er immer dann, wenn die Richtigkeit einer Aussage zu überprüfen ist, mit —1 antwortet bei wahrer Aussage, dagegen mit 0 bei falscher. Auf diese Weise kommen diese merkwürdigen Basic-Programmzeilen ins rechte Licht, in denen Sequenzen auftauchen wie:
C=A—161—33*(A< 255)—64*(A < 192)—32*(A <160)+32*(A < 96)—64*(A < 64).

Jedesmal, wenn zum Beispiel A < 64 ist, tritt anstelle der Klammer ein —1 auf. Übrigens ist diese Formel eine schöne kurze Möglichkeit, ASCII-Code (hier A als Variable) in den Bildschirmcode umzurechnen (der Bildschirmcode steht dann in der Variablen C).

Kommen wir nun zur zweiten Gruppe von Assembler-Befehlen, die Bit-Manipulationen erlauben: den Verschiebe-Befehlen. Fangen wir dabei mit ASL an, was vom englischen »arithmetic shift left« kommt. Zu deutsch heißt das dann »arithmetisches nach links schieben«. Davon sind wir aber auch noch nicht schlauer. Sehen wir uns an, was dieser Befehl tut (Bild 2).

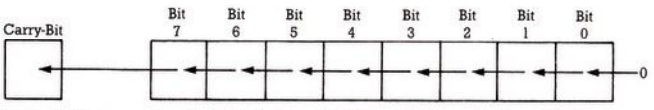

Bild 2. Wirkung des ASL-Befehls: Arithmetisches Linksschieben

Der gesamte Inhalt des Akku beziehungsweise der Speicherstelle (je nach Adressierung) wird um eine Bit-Position nach links verschoben. Das vorherige Bit 7 wandert in die Carry-Flagge, alle anderen Bits erhalten eine um 1 höhere Position, das freigewordene Bit 0 wird mit einer 0 aufgefüllt. Toll! Aber was soll das? Zur Erklärung machen wir nochmal einen kurzen Ausflug zu unserem normalen dezimalen Zahlensystem. Nehmen wir mal die Zahl 123. Bei der Einführung in die Fließkommazahlen hatten wir das Komma zu verschieben gelernt. 123 ist ja dasselbe wie 123,00. Wenn wir das Komma um eine Stelle nach rechts verschieben, erhalten wir 1230,0 (dabei lassen wir jetzt mal den Exponenten außer acht, der wäre ja —1, weil 123,00 = 1230,0 $x10^{-1}$). Man kann das Ganze auch andersherum sehen: Wir haben die Zahl 123 eine Stelle nach links verschoben und die freigewordene Stelle ganz rechts mit einer Null aufgefüllt. 1230,0 ist das Zehnfache von 123,00. Die Verschiebung um eine Stelle nach links hat also zur Multiplikation unserer Zahl mit der Basis unseres Zahlensystems (also 10) geführt. Eine zweimalige Linksverschiebung führt zu 12300, den 100fachen Wert unserer Ausgangszahl. Wir haben also die Zahl 123,00 mal 10 mal 10 genommen, das sind 10^2. Jede Linksverschiebung erhöht unseren Ausgangswert um eine Zehnerpotenz, oder — anders ausgedrückt — erhöht den Multiplikator um eine Zehnerpotenz und deshalb natürlich auch das Ergebnis (einmal linksschieben: Multiplikator = $10 = 10^4$, zweimal linksschieben: Multiplikator: 100 = 10^2 und so weiter).

Im Binärsystem, zu dem wir nun wieder zurückkehren, ist die Zahlenbasis die Zahl 2. Einmal linksschieben entspricht dann einer Multiplikation mit $2^1 = 2$. Das zweimalige Linksschieben führt zur Multiplikation mit $2^2 = 4$ und so weiter. Nehmen wir als Beispiel die Zahl 3, welche am Binärsystem 0000 0011 heißt:

Bis jetzt landete im Carry-Bit immer eine Null. Wenn wir nun nochmal linksschieben, finden wir darin eine 1, die offensichtlich als Bit 8 unseres Ergebnisses dienen muß:

7. ASL (1) 1000 0000 = (mit Carry als Bit 8) dez. 384 (2x192=384, 2^7x3=128x3=384

Daraus folgt, daß immer dann, wenn man sich nicht hundertprozentig sicher ist, eine Abfrage des Carry-Bits erfolgen sollte, sofern man ASL zum Rechnen einsetzt (BCC beziehungsweise BCS bieten sich da an). Dazu kommen wir noch. Sehen wir uns zunächst mal an, wie ASL adressiebar ist:

ASL
ohne Adresse, der Akkuinhalt wird nach links verschoben. Manchmal als eigene Adressierungsart bezeichnet.
ASL 6000
absolut
ASL FE
Zeropage-absolut
ASL 6000,X
absolut-X-indiziert
ASL FA,X
Zeropage-absolut-X-indiziert

Je nach Adressierung tritt ASL dann als 1-, 2- oder 3-Byte-Befehl auf. Die N-, die Z- und die Carry-Flagge werden beeinflußt. Das Ergebnis steht bei der ersten Adressierungsart (also ASL ohne Adresse) im Akku. In den anderen Fällen findet man es in der jeweiligen Speicherstelle.

Nun gut, werden Sie sagen, man kann also mittels ASL Zahlen mit 2, 4, 8, 16 32 etc. multiplizieren. Was aber, wenn man mal 40 nehmen will? Da gibt es einige Möglichkeiten, die ein bißchen den Erfindungsgeist ansprechen. Man kann ja, wenn irgendeine Zahl Z mal 40 gerechnet werden soll, dafür schreiben:
40xZ=(32+8)xZ=32xZ+8xZ

Schon haben wir wieder Multiplikatoren, die den Einsatz von ASL ermöglichen. Die beiden Zwischenergebnisse (als 32xZ und 8xZ) speichern wir irgendwo ab und zählen sie dann zusammen. Wenn Z zum Beispiel 3 wäre, könnte man das so programmieren:

```
6000    STA 6100
```

1. ASL	führt zu	0000 0110	= dez.6 (2^1x3=2x3=6)
2. ASL		0000 1100	= dez. 12 (2x6= 12, 2^2x3=4x3=12)
3. ASL		0001 1000	= dez.24 (2x12=24, 2^3x3=8x3=24)
4. ASL		0011 0000	= dez. 48 (2x24=48, 2^4x3=16x3=48)
5. ASL		0110 0000	= dez. 96 (2x48=96, 2^5x3=32x3=96)
6. ASL		1100 0000	= dez. 192 (2x96=192, 2^6x3=63x3+192)

Dabei sollte im Akku Z also die 3 stehen, die wir nun zwischengespeichert haben.

```
6003    ASL
6004    ASL
6005    ASL
6006    ASL
6007    ASL
```

Jetzt liegt im Akku der 32fache Wert von 3, also 96 vor und wir speichern dieses Zwischenergebnis ab.

```
6008    STA 6101
600B    LDA 6100
```

Wir haben nun den Wert 3 aus dem Zwischenspeicher $6100 wieder in den Akku geholt und schieben ihn 3mal nach links um den 8fachen Wert zu erhalten.

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.

Tabelle 4. Alles Wissenswerte der neuen Assembler-Befehle

```
600E    ASL
600F    ASL
6010    ASL
```

Nun erfolgt das Zusammenzählen beider Zwischenergebnisse. Dabei ist ja 8xZ noch im Akku.

```
6011    CLC
6012    ADC 6101
```

Damit ist die Aufgabe gelöst. Das Ergebnis steht im Akku und kann nun weiter verwendet werden.

Auf diese Weise kann man immer einen Mulitplikator in eine Zweierpotenz (2, 4, 8, 16,...) und weitere Summanden zerlegen. Dies ist allerdings eine zwar schnelle, aber doch recht eingeschränkte Art der Multiplikation. Außerdem haben Sie noch nicht erfahren, wohin man denn nun am besten mit BCC verzweigt, wenn die 8 Bits des Ergebnisses überlaufen.

Das alles aber, liebe Leser, erfahren Sie erst beim nächsten mal. Für heute war's lang und schwer genug! Wir werden dann einen eleganten Weg der 16-Bit-Multiplikation und Division erarbeiten. Und unser Programmprojekt soll natürlich endlich weiter gedeihen. Abschließend finden Sie in Tabelle 4 noch alles Wissenswerte zu den neuen Befehlen.

(Heimo Ponnath/gk)

Memory Map mit Wandervorschlägen

Weiter geht's mit der Erkundung der Speicherlandschaft. Die Adressen 57 bis 79 tragen zum Funktionieren eines Basic-Programms bei.

Heute sind eine Gruppe von Zeigern an der Reihe, die vom Betriebssystem des Computers während der Abarbeitung einzelner Programmzeilen verwendet werden.

Adresse 57 und 58 ($39 und $3A)

Nummer der laufenden Basic-Programmzeile

Diese Speicherzellen enthalten die Zeilennummer in Low-/High-Byte-Darstellung derjenigen Basic-Anweisung, welche gerade ausgeführt wird.
Ein kurzes Programm macht das deutlich:

```
10 PRINT "ZEILE 10", PEEK(57)+256*PEEK(58)
20 A=3:PRINT A,PEEK(57)+256*PEEK(58)
30 B=5:PRINT B,PEEK(57)+256*PEEK(58)
40 PRINT A*B,PEEK(57)+256*PEEK(58)
```

In jeder Zeile wird zuerst etwas gePRINTet, nämlich Text, Variable und ein Rechenresultat. Durch das Komma getrennt wird in der 2. Bildschirmhälfte (VC 20) beziehungsweise Bildschirmviertel (C 64) der Inhalt der Speicherzellen 57/58 ausgedruckt. Das Resultat zeigt in der Tat die jeweilige Zeilennummer an.

Die Basic-Befehle GOTO, GOSUB-RETURN, FOR-NEXT, END, STOP, CONT und die Betätigung der STOP-Taste während eines Programmlaufes verwenden alle den Inhalt dieser Speicherzellen, um entweder zu der laufenden Zeile zurückzufinden oder um die Unterbrechung mit BREAK IN... anzuzeigen. Auch die meisten Fehlermeldungen verwenden diese Zellen.

In vielen Basic-Erweiterungen und Programmierhilfen wird ein Befehl TRACE oder STEP angeboten, welcher ein schrittweises Abarbeiten eines Programms bei gleichzeitiger Anzeige der gerade aktiven Zeilennummer erlaubt. Dieses TRACE verwendet natürlich auch den Inhalt der Zellen 57/58.

Schließlich sei noch erwähnt, daß im direkten Modus, also bei direkt eingetippten Aktionen des Computers ohne Programmzeilen, in der Zelle 58 immer die Zahl 255 steht. Diejenigen Basic-Befehle, welche im direkten Modus nicht erlaubt sind (INPUT, GET, DEF) prüfen in Zelle 58, ob sie im direkten Modus oder während eines Programmlaufes aufgetreten sind.

Adresse 59 und 60 ($3B und $3C)

Zeilennummer der letzten Programmunterbrechung

Immer dann, wenn ein Programmablauf durch die Befehle END oder STOP oder aber mit der STOP-Taste abgebrochen wird, wird die Nummer der gerade ausgeführten Programmzeile nach 59/60 gebracht und bleibt dort so lange, bis eine neue Unterbrechung erfolgt.

Das läßt sich am besten mit der STOP-Taste und nachfolgendem CONT zeigen. Nehmen Sie bitte dazu das kleine Demo-Programm der Zellen 57/58 und ändern Sie alle PEEK-Adressen in 59/60 um. Fügen Sie außerdem noch eine Zeile 50 hinzu:

```
50 GOTO 10
```

Den dadurch erzeugten kontinuierlichen Lauf des Programms bremsen Sie dann mit der STOP-Taste und lassen ihn danach mit CONT weiterlaufen.

Auf der rechten Seite erscheint jetzt die Zeilennummer, bei der das Programm vorher unterbrochen worden ist.

Adresse 61 und 62 ($3D und $3E)

Zeiger auf die Adresse, ab welcher der Text der laufenden Basic-Zeile abgespeichert ist.

Die Abarbeitung der einzelnen Basic-Zeilen während eines Programmlaufs wird von einem kleinen Maschinencode-Programm, welches in den Speicherzellen 115 bis 138 steht (wir kommen noch dahin), gesteuert. In den Zellen 122/123 enthält es die Adresse, ab der die gerade bearbeitete Basic-Zeile gespeichert ist.

Sobald eine neue Basic-Zeile verarbeitet wird, holt das Betriebssystem diese Adresse aus 122/123 und speichert sie in den hier zur Diskussion stehenden Speicherzellen 61/62 ab, wie üblich als Low-/High-Byte.

Dasselbe geschieht bei jedem Befehl END, STOP, bei Fehlern mit dem Befehl INPUT und durch das Drücken der STOP-Taste. Der Befehl CONT hingegen schaut in 61/62 nach und bringt die darin befindliche Adresse zurück in die Speicherzellen 122/123 zur Fortsetzung