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 $1{,}985125 \times 10^3$ 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