Binäres Rechnen mit Papier und Bleistift

von Ulrich Stiehl, Heidelberg 1986

Die Computer-Zeitschrift *"Peeker – Magazin für Apple-Computer"*, die ich in den 4 Jahren von 1984 bis 1987 als Chefredakteur herausgegeben habe, ist heute nach Jahrzehnten natürlich nur noch von historischem Interesse.

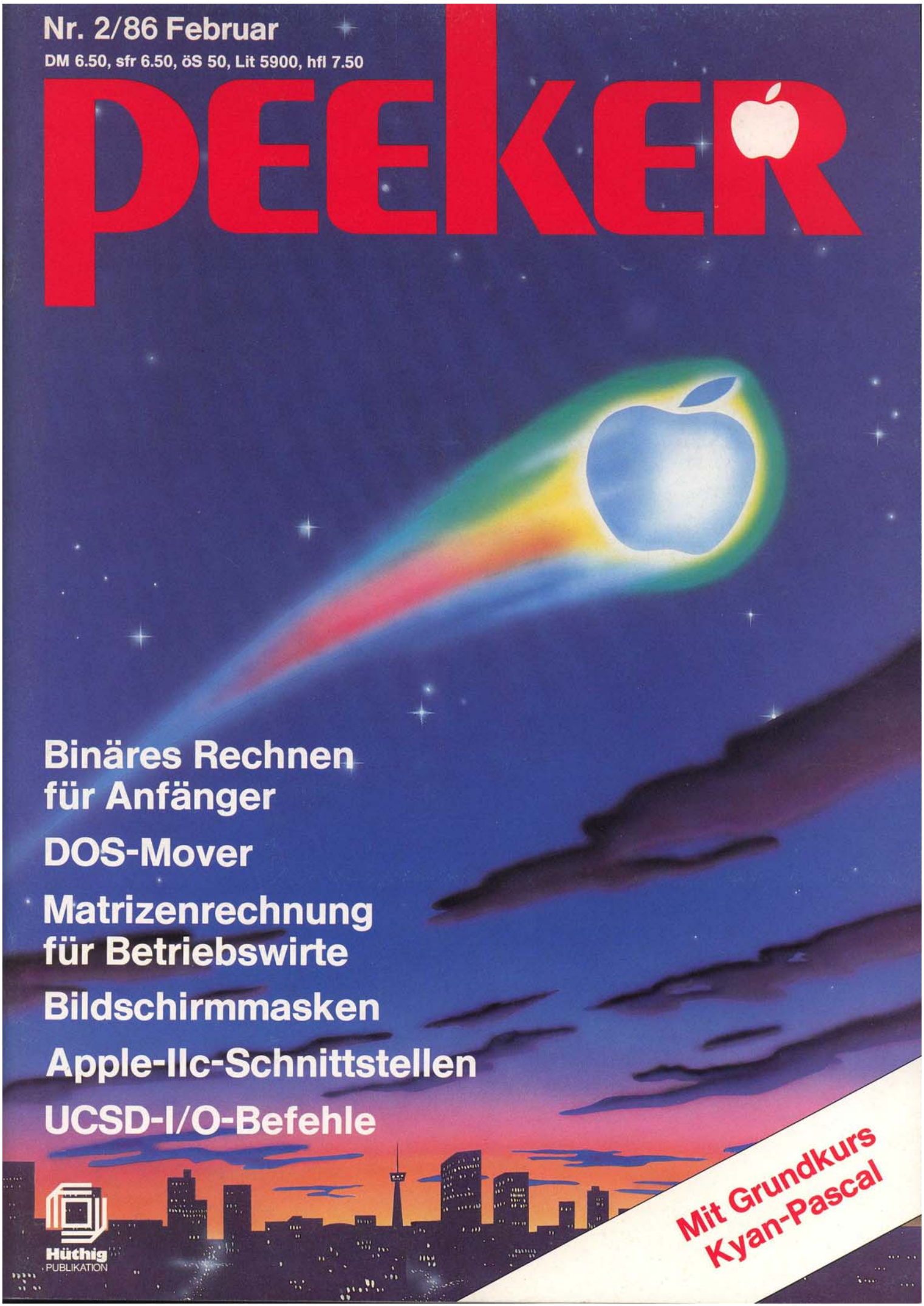
Das Magazin enthielt aber auch didaktisch aufbereitete Grundlagenbeiträge für den Informatik-Unterricht, die selbst heute noch nicht überholt sind.

Einen dieser Grundlagenbeiträge, der damals von zahlreichen Gymnasien angefordert wurde und als Zweiteiler in den Heften 2/86 und 4/86 erschien, habe ich hier für die vorliegende PDF-Datei mit 300 dpi gescannt.

http://www.sanskritweb.net/apple

Nr. 2/86 Februar
DM 6.50, sfr 6.50, öS 50, Lit 5900, hfl 7.50

peeker



Binäres Rechnen für Anfänger

DOS-Mover

Matrizenrechnung für Betriebswirte

Bildschirmmasken

Apple-IIc-Schnittstellen

UCSD-I/O-Befehle

Mit Grundkurs Kyan-Pascal

Hüthig
PUBLIKATION

Binäres Rech
mit Papier u

von Ulrich Stiehl

Teil 1: Addition und Subtraktion

Gliederung

1. Einleitung

Das binäre Rechnen mit natürlichen Zahlen ist leicht verständlich, *wenn* es nachvollziehbar erklärt wird. Die üblichen 6502-Lehrbücher von L.A.Leventhal, R.Hyde u.a. beschränken sich indes allesamt auf beiläufige Sätze in der Art „You can perform division on the computer just like you would perform division with pen and paper" und gehen dann sofort zur Tagesordnung über, indem sie Algorithmen präsentieren, die mit „paper and pencil" nur noch wenig gemein haben. Nirgendwo wird das binäre Rechnen einmal wirklich Schritt für Schritt auf das dezimale Rechnen mit Papier und Bleistift zurückgeführt. Folglich bleibt eine Lücke in der Beweisführung, und es ist von daher einsichtig, daß eine nicht geringe Zahl von Assemblerprogrammierern die binären Algorithmen „blind" anwendet.
Ein 6502-Profi wird möglicherweise jetzt naserümpfend ausrufen: „Ich gehöre nicht zu jenen ‚Blind'-Gängern!" Wirklich nicht? Dann prüfen Sie sich jetzt einmal selbst. Sie kennen das *binäre* Einer- und Zweierkomplement, z.B.
1001 → 0110 Einerkomplement
1001 → 0111 Zweierkomplement
Frage: Wie heißt das *dezimale* Zweierkomplement zu der Zahl 6789? Wenn Sie die Lösung spontan und ohne langes Knobeln niederschreiben können, dann sind Sie nicht so blind wie wir und brauchen den nachfolgenden Aufsatz nicht mehr zu lesen.

Im folgenden beschränken wir uns auf das schriftliche Rechnen (Addition, Subtraktion, Multiplikation und Division) mit **natürlichen Zahlen** (0, 1, 2, 3 usw.) im Dezimal- und Binärsystem. Somit werden negative Zahlen (-1, -2, -3 usw.) und Bruchzahlen (1/1, 1/2, 1/3 usw.) bzw. Dezimalzahlen im engeren Sinne (1.0, 0.5, 0.33333 usw.) ausgespart. Das schriftliche Rechnen im Dezimalsystem wird zwar als *praktische Fähigkeit* vorausgesetzt, doch werden wir zudem auf den zugrundeliegenden *theoretischen Algorithmus* eingehen, denn nur wenn man sich die Vorgehensweise beim schriftlichen Rechnen im Dezimalsystem *bewußt* macht, wird man die Rechenregeln auf das Binärsystem übertragen können.
Bei den einzelnen Abschnitten wird säuberlich zwischen Binärtheorie und 6502-Implementierung getrennt, so daß Nicht-Assemblerprogrammierer die 6502-Passagen überspringen können.

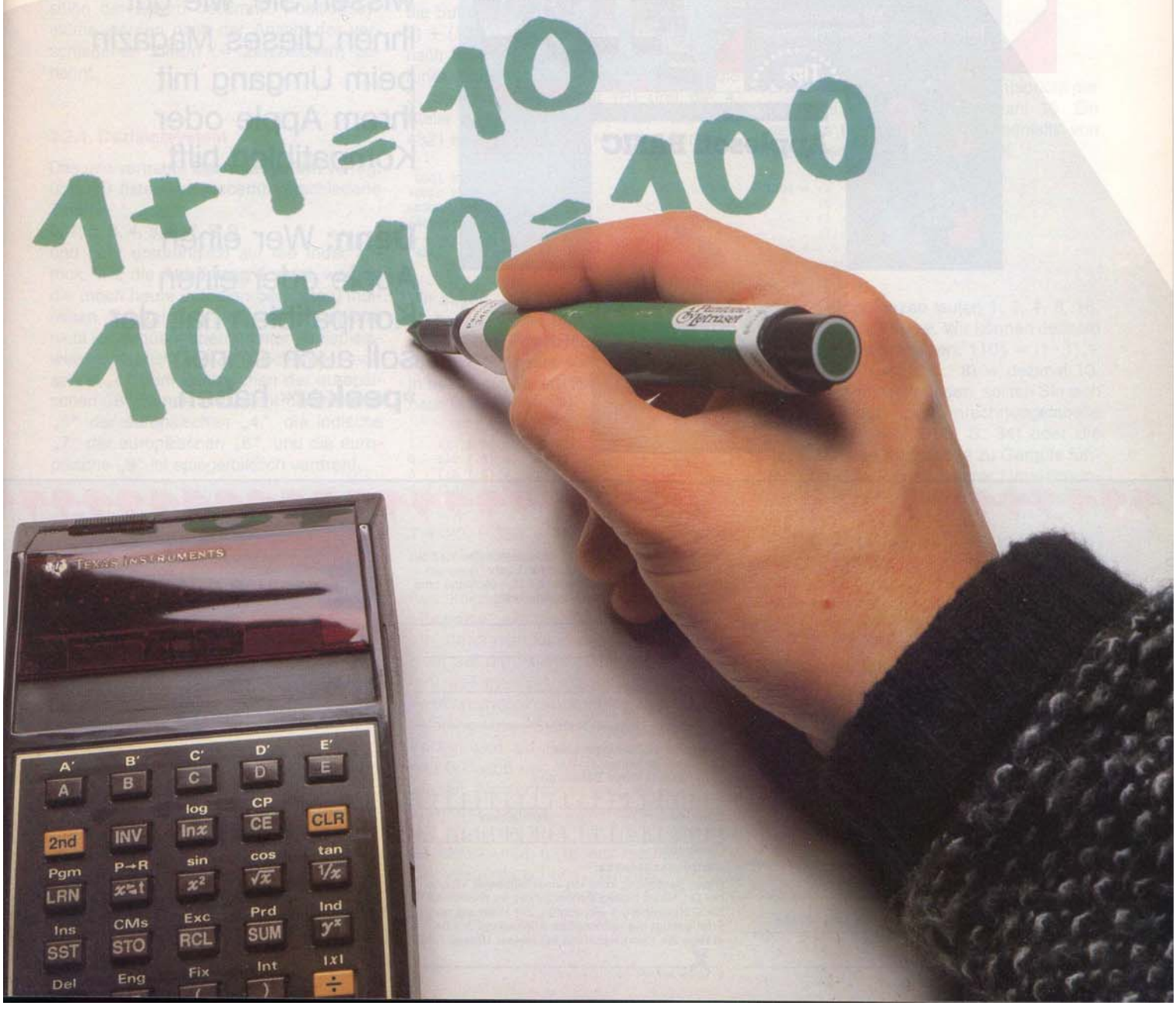
2. Zahlensysteme

Für jede Zahl könnte man theoretisch ein eigenes Zeichen erfinden, z.B. „●" usw. Da es jedoch unendlich viele Zahlen gibt, müßte man sich unendlich viele Zeichen ausdenken. Manche Naturvölker haben sich deshalb auf einige wenige Zahlen beschränkt. Demgegenüber haben Kulturvölker die **Ziffer** eingeführt, die sich als nicht-zusammengesetzte, „einziffrige" Zahl mit eigenem Zahlzeichen definieren läßt, z.B. „●". Zusammengesetzte, „mehrziffrige" Zahlen werden dann durch Verkettung der vorhandenen Ziffern gebildet, z.B. „●●●". Wenn sich der *Wert* einer mehrziffrigen Zahl aus der Summe der Ziffernwerte ergibt, spricht man vom Additionssystem. Wenn hingegen der Wert einer mehrziffrigen Zahl zusätzlich von der Stelle oder Position der Ziffern innerhalb der Ziffernkette abhängt, spricht man vom Positionssystem.

2.1. Additionssysteme

Die römische Zahl
XIII = 13
ist ein Beispiel für ein **Additionssystem**, denn es gilt
X + I + I + I = 13.
(M = 1000, D = 500, C = 100, L = 50, X = 10, V = 5, I = 1.)
Allerdings enthält das römische Zahlensystem bereits Elemente des Positionssy-

nnen
d Bleistift



stems. So hängt bei der Zahl
MCMLXXXVI = 1986
der Wert „CM" = 900 von der Position oder Stellung der Ziffer „C" (= 100) *vor* der Ziffer „M" (= 1000) ab, denn es gilt
CM = -100 + 1000 = 900
MC = 1000 + 100 = 1100
Ähnlich ist es mit dem französischen Zahlwort
quatre-vingts = 80, d.h. 4 · 20
gegenüber dem Zahlwort
vingt-quatre = 24, d.h. 20 + 4.

Wie ersichtlich, gibt es bei älteren Zahlensystemen etliche Sonderfälle und Ungereimtheiten.

2.2. Positionssysteme

Beim reinen **Positionssystem** wird der Wert einer Zahl konsequent durch die Position der Ziffern bestimmt. Positionssysteme werden nach der Anzahl der verschiedenen Ziffern (= Zahlzeichen) benannt.

2.2.1. Dezimalsystem

Das uns vertraute **Dezimalsystem** verfügt über 10 (lateinisch decem) verschiedene Ziffern
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
und geht ursprünglich auf die Inder zurück. Wie die **Abbildung 1** zeigt, wurden die (noch heute in Indien benutzten) indischen Zahlzeichen von den Europäern nicht konsequent übernommen. Beispielsweise entspricht das Zahlzeichen der indischen „4" dem Zahlzeichen der europäischen „8"; ferner entspricht die indische „5" der europäischen „4", die indische „7" der europäischen „6", und die europäische „9" ist spiegelbildlich verdreht.

Abbildung 1: Indische Zahlzeichen, wie sie heute in der Devanagari-Schrift benutzt werden (Hindi u. a.).

Da das Dezimalsystem das uns Europäern zunächst einzig vertraute Zahlensystem ist, verwundert es nicht, daß für den Begriff „Zahl im Dezimalsystem" kein eigenes Fachwort geprägt wurde. Wir sprechen zwar von Binärzahl (= „Zahl im Binärsystem") und Hexadezimalzahl (= „Zahl im Hexadezimalsystem"), aber der Terminus „Dezimalzahl" oder „dekadische Zahl" bedeutet normalerweise nicht „Zahl im Dezimalsystem", denn dafür haben wir bereits den allgemeinen, aber hier mißverständlichen Ausdruck „Zahl". Wir definieren deshalb **Dezimalzahl** (a) im weiteren Sinne als „Zahl im Dezimalsystem" und (b) im engeren Sinne als „Zahl im Dezimalsystem mit Dezimalpunkt bzw. -komma", z.B. 123.456 oder 123,456. In diesem Beitrag werden wir „Dezimalzahl" stets in dem erweiterten Sinne benutzen.

Bei Positionssystemen entspricht die Anzahl der Zahlzeichen auch gleichzeitig der sog. Basis. Betrachten wir hierzu folgende Aufstellung:

3210 Exponent
4321 Dezimalzahl

Die Dezimalzahl 4321 läßt sich auch als die Summe von (1 · 1) + (10 · 2) + (100 · 3) + (1000 · 4) schreiben, d.h. von rechts nach links repräsentiert die 1. Stelle die Einer (E), die 2. Stelle die Zehner (Z), die 3. Stelle die Hunderter (H) und die 4. Stelle die Tausender (T) usw. Die Zahl 4321 entsteht mithin durch die Addition

```
 0001 E
+0020 Z
+0300 H
+4000 T
-------
 4321
```

Wir sagen, daß Dezimalzahlen die **Basis** 10 haben, wobei unter Basis die Grundzahl in der Potenzrechnung gemeint ist (Basis ↑ Exponent = Potenz). Es gilt dann in bezug auf unsere Zahl 4321 von rechts nach links:

```
1 · (10 ↑ 0) =    1 (E)
2 · (10 ↑ 1) =   20 (Z)
3 · (10 ↑ 2) =  300 (H)
4 · (10 ↑ 3) = 4000 (T)
```

1 + 20 + 300 + 4000 = 4321. Die Zehnerpotenzen lauten 1, 10, 100, 1000 usw.

Aus philosophischer Sicht sind die obigen „Beweise" allerdings ein Hysteron-Proteron, denn man kann das Positionssystem nicht auf die Potenzrechnung zurückführen, weil die Potenzrechnung bereits das Positionssystem voraussetzt. Zur Axiomatisierung des Systems der natürlichen Zahlen wird auf die einschlägigen Werke von G.Peano und Whitehead/Russell verwiesen. Als Kurzdarstellung sei empfohlen „Handbuch philosophischer Grundbegriffe", München 1974, S.1775 ff. Uns geht es jedoch hier nicht um mathematische Grundlagenforschung, sondern um das praktische („vorphilosophische") Verständnis.

2.2.2. Binärsystem

Nachdem wir nunmehr das Dezimalsystem an Beispielen erläutert haben, fällt es uns leicht, andere Positionssysteme zu definieren. Im **Binärsystem** gibt es 2 Zahlzeichen – 0 und 1 (oder L) –, und die Basis ist 2. Die Zahlen im Binärsystem oder Dualsystem heißen Binärzahlen oder Dualzahlen („binär" von lateinisch „bis" = zweimal; „dual" von lateinisch „duo" = „zwei"), und eine einzelne Stelle einer x-stelligen Binärzahl wird als **Bit** bezeichnet (von englisch „*b*inary dig*it*" = „bit" = „Binärziffer"). Betrachten wir hierzu die folgende Aufstellung:

3210 Exponent
1111 Binärzahl

Auch hier können wir wieder von rechts nach links eine Addition vornehmen:

```
1 · (2 ↑ 0) = dezimal 1
1 · (2 ↑ 1) = dezimal 2
1 · (2 ↑ 2) = dezimal 4
1 · (2 ↑ 3) = dezimal 8
```

1 + 2 + 4 + 8 = 15. Somit entspricht der Binärzahl 1111 die Dezimalzahl 15. Ein weiteres Übungsbeispiel – ebenfalls von rechts nach links ausgerechnet:

```
1101 = (1 · (2 ↑ 0)) +
       (0 · (2 ↑ 1)) +
       (1 · (2 ↑ 2)) +
       (1 · (2 ↑ 3))
1101 = 1 + 0 + 4 + 8
1101 = dezimal 13
```

Die Zweierpotenzen lauten 1, 2, 4, 8, 16, 32, 64, 128, 256 usw. Wir können deshalb auch verkürzt schreiben: 1101 = (1 · 1) + (0 · 2) + (1 · 4) + (1 · 8) = dezimal 13. Bevor Sie nun weiterlesen, sollten Sie sich eine binär-dezimale Umrechnungstabelle (z.B. aus Peeker 7/85, S. 34) oder die nachstehende Kurztabelle zu Gemüte führen und eine Reihe weiterer Umrechnungen mit Papier und Bleistift vornehmen.

Eine Binärzahl läßt sich also in eine Dezimalzahl umrechnen, indem man von rechts nach links die der Position entsprechende Potenz entweder mit 0 oder mit 1 malnimmt, womit das Ergebnis entweder 0 ist oder der Positionspotenz entspricht. Umgekehrt läßt sich eine Dezimalzahl in eine Binärzahl umwandeln, indem man fortlaufend durch 2 teilt und den jeweiligen Rest (0 oder 1) von oben nach unten oder von rechts nach links notiert. Beispiel:

```
20 : 2 = 10 Rest 0 (0.)
10 : 2 =  5 Rest 0 (1.)
 5 : 2 =  2 Rest 1 (2.)
 2 : 2 =  1 Rest 0 (3.)
 1 : 2 =  0 Rest 1 (4.)
```

43210 Exponent
10100 Binärzahl

2.2.3. Hexadezimalsystem

Neben dem Dezimal- und Binärsystem ist noch das **Hexadezimalsystem** gebräuchlich. Hier gibt es 16 Zahlzeichen – 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F –, und die Basis ist 16. Neben der Bezeichnung „Hexadezimalzahl" oder kurz „Hexzahl" („hex" griechisch „sechs", „decem" lateinisch „zehn") gibt es noch den Terminus „Sedezimalzahl" („sedecim" bzw. „sex-decim" lateinisch „sechzehn"), der sich jedoch nicht durchgesetzt hat. Manche Puristen wollen indessen nur die reinlateinische Bezeichnung „sedezimal" gelten lassen. Es sind dies dieselben Puristen, die vom rein-lateinischen „Ipsomobil" oder vom rein-griechischen „Autokinetikon" sprechen, während der Rest der Menschheit weiterhin beim griechisch-lateinischen „Automobil" oder kurz „Auto" bleibt.

Die nachfolgende Kurztabelle dient zum bequemen Umrechnen zwischen den drei Positionssystemen – binär, dezimal, hexadezimal:

```
0000 = 00 = 0
0001 = 01 = 1
0010 = 02 = 2
0011 = 03 = 3
0100 = 04 = 4
0101 = 05 = 5
0110 = 06 = 6
0111 = 07 = 7
1000 = 08 = 8
1001 = 09 = 9
1010 = 10 = A
1011 = 11 = B
1100 = 12 = C
1101 = 13 = D
1110 = 14 = E
1111 = 15 = F
```

Hexadezimalzahlen wurden in der EDV deshalb eingeführt, weil die Speicherstelle, die ein Mikroprozessor bearbeiten kann, in der Regel das x-fache einer 4stelligen Binärzahl und somit das x-fache einer 1stelligen Hexzahl enthält. Im einzelnen unterscheidet man:

Nibble, z.B. 1010 = A
4stellige Binärzahl = 1stellige Hexzahl

Byte, z.B. 10101010 = AA
8stellige Binärzahl = 2stellige Hexzahl

Wort, z.B. 1010101010101010 = AAAA
16stellige Binärzahl = 4stellige Hexzahl

Da der 6502-Prozessor hinsichtlich der Binärmathematik nur über Additions- und Subtraktionsbefehle verfügt, die exakt jeweils 1 ganzes Byte bearbeiten (= 8stellige Binärzahl = 8 Bits = 2stellige Hexzahl), sind hier die Operanden (Summand, Minuend, Multiplikator, Divisor usw.) einer binären Rechenoperation stets ein x-faches eines Bytes. Beispiele: 8-mal-8-Bit-Multiplikation, 16-durch-8-Bit-Division usw.

3. Addition

3.1. Dezimale Addition

Einstellige Dezimalzahlen können wir „im Kopf" addieren, d.h. wir wissen sofort, daß z.B. 2 + 2 = 4 ist. Mehrstellige Dezimalzahlen werden schriftlich addiert, indem man die beiden Summanden oder Addenden rechtsbündig untereinanderschreibt und von rechts nach links die Einer, Zehner, Hunderter usw. zur Summe zusammenzählt. Beispiel:

```
  1234 Summand
+ 2345 Summand
------------
  3579 Summe
```

Dabei kann ein **Übertrag** von der Einer- zur Zehnerposition, von der Zehner- zur Hunderterposition usw. stattfinden. Beispiel:

```
  1234 Summand
+ 5778 Summand
   11  Übertrag
------------
  7012 Summe
```

4 + 8 ergibt 12 (= 10 + 2). Da jedoch die Einerposition nur 1 Ziffer aufnehmen kann, tragen wir bei der Einerposition 2 ein und übernehmen die 1, die für die 1 von 10 steht, in die Zehnerposition. Danach addieren wir in der Zehnerspalte 3 + 7 + 1 = 11 usw. Dies alles ist Ihnen seit Ihrer Kindheit bereits bestens vertraut.

Es gibt folgende Grundregeln, die sowohl für Dezimal- als auch Binärzahlen gelten:

1. Wenn man 2 Summanden addiert, ist der Übertrag in die nächste Position höchstens 1. Dies gilt nicht, wenn man mehr als 2 Summanden auf einmal addiert.

2. Wenn man 2 x-stellige Summanden addiert, ist das Ergebnis höchstens (x+1)-stellig. Dies gilt ebenfalls nicht, wenn man mehr als 2 Summanden auf einmal addiert. Beispiel für den ungünstigsten Fall:

```
  9999 Summand
+ 9999 Summand
 1111  Übertrag
------------
 19998 Summe
```

3.2. Binäre Addition

Bei der binären Addition müssen wir uns zunächst mit den Rechenregeln vertraut machen. Da wir später bei der binären Multiplikation auch mit mehr als 2 Summanden rechnen werden, geben wir auch deren Additionsregeln an:

```
0 + 0 = 0 Übertrag 0
1 + 0 = 1 Übertrag 0
0 + 1 = 1 Übertrag 0
1 + 1 = 0 Übertrag 1
1 + 1 + 1 = 1 Übertrag 1
1 + 1 + 1 + 1 = 0 Übertrag 1 + 1
1 + 1 + 1 + 1 + 1 = 1 Übertrag 1 + 1
usw.
```

Man präge sich ein, daß im Binärsystem 1 + 1 nicht etwa 2 ergibt, denn die 2 existiert hier gar nicht als Zahlzeichen. 1 + 1 ergibt mithin 10 oder 0 Übertrag 1.

Bitte rechnen Sie nun mit Papier und Bleistift folgende Übungen durch:

```
  0000 Summand (00)
+ 1111 Summand (15)
-------------------
  1111 Summe   (15)

  1010 Summand (10)
+ 0101 Summand (05)
-------------------
  1111 Summe   (15)

  0111 Summand  (07)
+ 0011 Summand  (03)
   111 Übertrag (1 )
-------------------
  1010 Summe    (10)

  1111 Summand  (15)
+ 0111 Summand  (07)
  1111 Übertrag (1 )
-------------------
 10110 Summe    (22)

  1111 Summand  (15)
+ 1111 Summand  (15)
  1111 Übertrag (1 )
-------------------
 11110 Summe    (30)
```

Dieses letzte Beispiel zeigt, daß die Addition von 2 x-stelligen Binärzahlen höchstens zu einem (x+1)-stelligen Ergebnis führt. Dies gilt nicht, wenn 3 oder 4 Binärzahlen auf einmal addiert werden:

```
   1110 Summand  (14)
+  1111 Summand  (15)
+  1100 Summand  (12)
   111  Übertrag (1 )
  111   Übertrag
-------------------
 101001 Summe    (41)

   1111 Summand  (15)
+  1111 Summand  (15)
+  1111 Summand  (15)
+  1111 Summand  (15)
   1111 Übertrag (2 )
   1111 Übertrag
  1111  Übertrag
-------------------
 111100 Summe    (60)
```

Wie ersichtlich, muß man bei der Addition von mehr als 2 Summanden mehr als eine Übertragszeile einrichten.

3.3. 6502-Addition

Während wir bei der schriftlichen Addition Binärstelle für Binärstelle, also Bit für Bit, zusammengezählt haben, erfolgt die 6502-Addition aus der Sicht des Programmierers stets in 8-Bit-Schüben, d.h. byteweise. Im einfachsten Fall werden 2 Bytes zusammengezählt.

Der Übertrag bzw. das Übertrag-Bit wird im Status-Register des 6502-Prozessors als sog. **Carry-Flag** (= Übertrag-Speicherstelle) registriert. Das Carry-Flag C kann mit **CLC** (= Clear Carry) auf 0 und mit **SEC** (= Set Carry) auf 1 gesetzt werden. Der Additionsbefehl **ADC** addiert den Inhalt der Speicherstelle M (= Memory) zu dem Inhalt des Akkumulators A und legt das Ergebnis (vorübergehend) im Akkumulator A ab. Unter **Vor-Übertrag** versteht man den Inhalt des Carry-Flags *vor* der Addition und unter **Nach-Übertrag** den Inhalt des Carry-Flags *nach* der Addition. Es gibt bei anderen Prozessoren einen Additionsbefehl, der nur den Nach-Übertrag registriert, der bei der Addition von A und M entsteht, symbolisch
A + M → A + C.

Der 6502-Befehl ADC (= Add *with* Carry) berücksichtigt jedoch nicht nur den Nach-Übertrag, sondern auch stets den Vor-Übertrag, symbolisch
A + M + C → A + C.

Deshalb muß eine Addition immer mit CLC eingeleitet werden. Beispiele:

Vorher:
C = 0
A = 00000001
M = 00000001
Befehl:
ADC M
Nachher:
C = 0
A = 00000010
Kommentar: Hier stört das fehlende CLC nicht, weil C bereits vorher *zufällig* 0 war. Darauf kann man sich jedoch nicht verlassen.

Vorher:
C = 1
A = 00000001
M = 00000001
Befehl:
ADC M
Nachher:
C = 0
A = 00000011
Kommentar: Hier fand eine fehlerhafte Addition statt (1 + 1 = 3!), weil der Vor-Übertrag nicht durch CLC gelöscht wurde (effektiv 1 + 1 + 1 = 3).

Vorher:
C = 1
A = 00000001
M = 00000001
Befehl:
CLC
ADC M
Nachher:
C = 0
A = 00000010
Kommentar: Richtige Addition.

Wenn wir für eine 8-und-8-Bit-Addition als Ergebnis nur 1 Byte reservieren, so können wir einen möglichen **Überlauf** (= Nach-Übertrag 1) durch folgende Befehlssequenz abfangen:

```
CLC
ADC M
BCS FEHLER
BCC ENDE
```

Grund: CLC löscht den Vor-Übertrag. Wenn dann *nach* der Addition von A und M der Nach-Übertrag 1 beträgt, so muß die Summe von A + M größer als 11111111, d.h. größer als dezimal 255 gewesen sein. Im Falle eines Überlaufs (Nach-Übertrag 1) wird der Sprung BCS (Branch on Carry set) ausgeführt, andernfalls der Sprung BCC (Branch on Carry clear).

Ein vollständiges Additionsbeispiel für die 8-Bit-Summanden M1 und M2 sieht im Falle einer 8-Bit-Summe S so aus:

```
CLC
LDA M1
ADC M2
BCS FEHLER
STA S
BCC ENDE
```

Gestattet man eine 16-Bit-Summe SL und SH (Low und High Byte = niederwertiges und höherwertiges Byte), so wird ein nunmehr zulässiger Überlauf im höherwertigen Byte der Summe abgefangen. SH enthält nach der Addition entweder 0 (kein Überlauf) oder 1 (Überlauf). Einen größeren Wert als 1 kann SH niemals enthalten, weil der Nach-Übertrag höchstens 1 ist:

```
LDA #0    ;SH auf
STA SH    ;0 setzen.
CLC       ;Vor-Übertrag = 0
LDA M1    ;1. Summand
ADC M2    ;2. Summand
STA SL    ;SL-Summe
BCC ENDE  ;kein Nach-Übertrag.
INC SH    ;SH auf 1 erhöhen
BCS ENDE  ;wegen Nach-Übertrag.
```

Wenn die Summanden mindestens je 2 Bytes umfassen, so muß man vom niederwertigen (Low) Byte zum höherwertigen (High) Byte – also quasi von rechts nach links – addieren. Nehmen wir an, der 1. Summand belege die Speicherstellen M1L und M1H und der 2. Summand M2L und M2H. Als Summe dieser 2-Byte- oder 16-Bit-Addition sei ein 3-Byte- oder 24-Bit-Ergebnis SL, SM, SH (Low/Middle/High Byte) zugelassen. Die Addition sieht dann wie folgt aus:

```
LDA #0    ;SH auf
STA SH    ;0 setzen.
CLC       ;Vor-Übertrag = 0
LDA M1L   ;Low Bytes
ADC M2L   ;der Summanden.
STA SL    ;SL-Summe
LDA M1H   ;High Bytes
ADC M2H   ;der Summanden.
STA SM    ;SM-Summe
BCC ENDE  ;kein Nach-Übertrag.
INC SH    ;SH auf 1 setzen
BCS ENDE  ;wegen Nach-Übertrag.
```

Betrachten wir diese Addition an einem konkreten Zahlenbeispiel:

```
  High     Low
  11111111 11111111 Summand M1
+ 11111111 11111111 Summand M2
  1        1        0 Übertrag
-------------------
  11111111 11111110 Summe S
```

Da wir die Addition mit CLC einleiten, ist der Vor-Übertrag auf 0 gesetzt. Die Addition der beiden niederwertigen Bytes der Summanden läuft also auf
11111111 + 11111111 + 0
hinaus, wodurch ein Nach-Übertrag von 1 entsteht, der zugleich zum Vor-Übertrag bei der nachfolgenden Addition der beiden höherwertigen Bytes der Summanden wird, so daß für diese gilt:
11111111 + 11111111 + 1
Dies führt zu einem Nach-Übertrag von 1 in das 3. Byte der Summe, die nunmehr effektiv so aussieht:

```
High     Middle   Low
00000001 11111111 11111110
```

Wie dieses ungünstigste Zahlenbeispiel zeigt, sind Vor- und Nach-Übertrag jeweils höchstens 1, so daß problemlos eine beliebig genaue binäre Addition implementiert werden kann.

4. Subtraktion

4.1. Dezimale Subtraktion

Während man bei der Addition die beiden Summanden vertauschen kann, muß man bei der Subtraktion die Reihenfolge beachten, denn 7 - 2 entspricht nicht 2 - 7. Bei der Gleichung
7 - 2 = 5
ist die 7 der Minuend, die 2 der Subtrahend und die 5 die Differenz. Der Minuend ist also die Zahl, von der der Subtrahend abgezogen wird. Während man die Subtraktion einstelliger Dezimalzahlen „im Kopf" erledigt, muß man bei mehrstelligen Zahlen Minuend und Subtrahend rechtsbündig untereinanderschreiben. Danach wird von rechts nach links entweder die

Subtrahendenziffer von der jeweiligen Minuendenziffer abgezogen (= **norddeutsches Wegnehmen**) oder die Differenz zwischen Minuendenziffer und Subtrahendenziffer zur Subtrahendenziffer hinzugezählt (= **süddeutsches Ergänzen**).
Wenn kein **Borgen** (= negativer Übertrag = „Vortrag" = „Unterlauf" = „der Borg") stattfindet, d.h. wenn die Minuendenstellen nicht kleiner als die jeweiligen Subtrahendenstellen sind, sind beide Verfahren gleichwertig. Beispiel ohne „Borgen":

```
 5678 Minuend
-1234 Subtrahend
------------------
 4444 Differenz
```

Norddeutsch: 8 weniger 4 = 4, 7 weniger 3 = 4, 6 weniger 2 = 4 und 5 weniger 1 = 4.
Süddeutsch: 4 bis 8 = 4, 3 bis 7 = 4, 2 bis 6 = 4 und 1 bis 5 = 4.

Norddeutsches Borgen:

```
  111  Übertrag
 7234 Minuend
-5678 Subtrahend
------------------
 1556 Differenz
```

1. Stelle: 4 - 8 geht nicht, also 14 - 8 = 6, 1 geborgt;
2. Stelle: 3 - 1 = 2, 2 - 7 geht nicht, also 12 - 7 = 5, 1 geborgt;
3. Stelle: 2 - 1 = 1, 1 - 6 geht nicht, also 11 - 6 = 5, 1 geborgt;
4. Stelle: 7 - 1 = 6, 6 - 5 = 1; fertig.
Beim norddeutschen Verfahren wird also der geborgte Übertrag von der *nächsten Minuendenstelle abgezogen.*

Süddeutsches Borgen:

```
 7234 Minuend
-5678 Subtrahend
  111  Übertrag
------------------
 1556 Differenz
```

1. Stelle: 8 bis 4 geht nicht, also 8 bis 14 = 6, 1 geborgt;
2. Stelle: 7 + 1 = 8, 8 bis 3 geht nicht, also 8 bis 13 = 5, 1 geborgt;
3. Stelle: 6 + 1 = 7, 7 bis 2 geht nicht, also 7 bis 12 = 5, 1 geborgt;
4. Stelle: 5 + 1 = 6, 6 bis 7 = 1; fertig.
Beim süddeutschen Verfahren wird also der geborgte Übertrag zur nächsten *Subtrahendenstelle hinzugezählt.*

4.2. Binäre Subtraktion

Zunächst müssen wir uns mit den Rechenregeln der binären Subtraktion vertraut machen, wobei wir uns auf den einfachsten Fall (1 Minuend minus 1 Subtrahend) beschränken, weil die kombinierte Subtraktion (1 Minuend minus *mehrere* Subtrahenden) „zuviel Kopf nimmt" (Adam Ries) und deshalb hier unnötig verwirren würde.

1. – 3. Regel

```
 0   1   1
-0  -1  -0
---------
 0   0   1
```

Dies sind die drei einfachen Regeln, weil sie zu keinem negativen Übertrag führen.

4. Regel

```
 0
-1
---
 1
```

Hier muß geborgt werden, indem anstelle des effektiven Minuenden 0 der Schein-

Minuend 10 benutzt wird, wobei die „1" der „10" von der nächsten Stelle geborgt wird, also so:

```
 10 (2)
- 1 (1)
--------
  1 (1)
```

Je nachdem, ob die nächste Minuendenstelle 1 oder 0 beträgt und ob bei der Subtraktion der momentanen Stelle bereits ein negativer Übertrag der vorangehenden Stelle zu berücksichtigen ist, sind einige Spezialfälle zu unterscheiden, wobei wir uns aus Platzgründen auf das süddeutsche Verfahren (Subtrahendenstelle + Übertrag statt Minuendenstelle - Übertrag) beschränken wollen:

```
 321 1., 2., 3. Stelle
 100 Minuend
-011 Subtrahend
  11 Übertrag
-----------------
 001 Differenz
```

1. Stelle: 1 bis 0 geht nicht, also 1 bis 10 = 1, 1 geborgt;
2. Stelle: 1 + 1 = 10, 10 bis 0 geht nicht, also 10 bis 10 = 0, 1 geborgt;
3. Stelle: 0 + 1 = 1, 1 bis 1 = 0; fertig.

```
 321 1., 2., 3. Stelle
 110 Minuend
-011 Subtrahend
  11 Übertrag
-----------------
 011 Differenz
```

1. Stelle: 1 bis 0 geht nicht, also 1 bis 10 = 1, 1 geborgt;
2. Stelle: 1 + 1 = 10, 10 bis 1 geht nicht, also 10 bis 11 = 1, 1 geborgt;
3. Stelle: 0 + 1 = 1, 1 bis 1 = 0; fertig.

Wie ersichtlich, kann die Auflösung der Minuendenstelle höchstens zu 10 (dezimal 2) oder 11 (dezimal 3) führen („Zwei-Drei-Regel"). Die nach dem süddeutschen Verfahren vorgenommene Zusammenfassung von Subtrahend + Übertrag (= Gesamtsubtrahend) kann 1 (dezimal 1) oder 10 (dezimal 2) ergeben. Daraus lassen sich alle denkbaren Kombinationen ableiten. Versuchen Sie nun, die folgenden Beispiele mit Papier und Bleistift zu lösen.

```
 1101 (13)
-1010 (10)
   1
---------
 0011 (03)

 1100 (12)
-1011 (11)
   11
---------
 0001 (01)

 1000 (08)
-0001 (01)
  111
---------
 0111 (07)
```

Wer sich bei den obigen Aufgaben unterfordert fühlen sollte, probiere einmal die folgende kombinierte Subtraktion. Der Umweg über die dezimale Umrechnung wäre gemogelt!

```
 11111110 Minuend
-00000001 Subtrahend 1
-00000010 Subtrahend 2
-00000101 Subtrahend 3
-00001011 Subtrahend 4
-00010111 Subtrahend 5
-00101111 Subtrahend 6
-01011111 Subtrahend 7
          Übertrag
-------------------------
          Differenz (???)
```

4.3. 6502-Subtraktion

Die 6502-Subtraktion läßt sich als Umkehrung der 6502-Addition begreifen:
– Dem ADC-Additionsbefehl entspricht der Befehl **SBC** in der Subtraktion.
– Vor der Addition muß das Carry-Flag gelöscht (CLC), vor der Subtraktion gesetzt werden (SEC).
– Nach der Addition bedeutet ein ausgeführter BCC, daß kein Additionsüberlauf auftrat, nach der Subtraktion bedeutet ein ausgeführter BCS, daß kein Subtraktionsüberlauf („Unterlauf") auftrat.
Die 8-minus-8-Bit-Subtraktion sieht gegenüber der entsprechenden Addition mithin so aus:

```
Subtraktion          Addition
-------------------------------
SEC                  CLC
LDA MINUEND          LDA SUMMAND1
SBC SUBTRAHEND       ADC SUMMAND2
BCC FEHLER           BCS FEHLER
BCS ENDE             BCC ENDE
```

Eine 16-minus-16-Bit-Subtraktion, bei der ML/MH, SL/SH und DL/DH für die jeweils nieder/höherwertigen Bytes des Minuenden, des Subtrahenden und der Differenz stehen, würde folgendermaßen codiert:

```
SEC
LDA ML      ;Low Byte
SBC SL
STA DL
LDA MH      ;High Byte
SBC SH
STA DH
BCC FEHLER
BCS ENDE
```

Bislang haben wir die Subtraktion einfach als Gegenstück zur Addition mit umgekehrten Vorzeichen betrachtet. Wenn wir jedoch ein konkretes Zahlenbeispiel unter die Lupe nehmen, kommt der „Blackout":

```
  10000000 00000000 Minuend
- 01111111 11111111 Subtrahend
  11111111 1111111  Übertrag
-----------------------------
  00000000 00000001 Differenz
  High     Low
```

Das obige Beispiel entspricht noch der klassischen süddeutschen Methode. Wir beginnen jedoch die Subtraktion durch SEC mit einem Vor-Übertrag! Danach müßte es „richtig" heißen:

```
  10000000 00000000 Minuend
- 01111111 11111111 Subtrahend
             1 Vor-Übertrag
  11111111 1111111  Übertrag
-----------------------------
  00000000 00000000 Differenz (?)
  High     Low
```

Dies kann offenbar nicht mit mit rechten Dingen zugehen, denn 32768 - 32767 ist immer noch 1 und nicht 0! Des Rätsels Lösung ist die sog. Komplement-Addition, denn die SEC-SBC-Befehlssequenz bedeutet eigentlich SEC-ACC („ACC" = Add Complement with Carry"). Wenn Ihnen dies alles zu verwirrend erscheint, so überspringen Sie am besten den nächsten Abschnitt, da er für die Multiplikation und Division nicht benötigt wird.

4.4. Komplement-Addition

4.4.1. Dezimale Komplement-Addition

Die Subtraktion „nimmt viel Kopf", wenn mehrere Minuendenstellen hintereinander durch Borgen aufgelöst werden müssen. Deshalb haben findige Mathematiker die Subtraktion auf die sog. komplementierte Addition zurückgeführt. Betrachten wir hierzu die folgenden Ziffernpaare:

0 – 9
1 – 8
2 – 7
3 – 6
4 – 5

Die 9 bezeichnen wir als **Einerkomplement** zur 0, die 8 als Einerkomplement zur 1 usw. Umgekehrt ist auch die 0 das Einerkomplement zur 9, die 1 das Einerkomplement zur 8 usw.
Im Dezimalsystem ist die Basis b = 10. Wenn wir z als *Ziffer* und z' als Komplement-Ziffer definieren, dann gilt:

z' = (b - 1) - z

Beispiel:
b = 10, z = 0, ergo
z' = (10 - 1) - 0
z' = 9 - 0
z' = 9

Das Einerkomplement Z' der *Zahl* Z wird gebildet, indem jede einzelne *Ziffer* z der Zahl Z zu z' komplementiert wird. Beispiele:

12345 Z
87654 Z'

43210 Z
56789 Z'

Das sog. **Zweierkomplement** Z'' zur Zahl Z wird gebildet, indem das Einerkomplement Z' um 1 erhöht wird. Beispiele:

```
 12345 Z
 87654 Z'
+    1
---------
 87655 Z''

 43210 Z
 56789 Z'
+    1
---------
 56790 Z''
```

Man hat nun herausgefunden, daß man die normale Subtraktion zweier x-stelliger Zahlen M (= Minuend) minus S (= Subtrahend), d.h.
M - S,
wenn gilt
M > = S,
auf die Komplement-Addition
M + S''
zurückführen kann, wenn man den sich dabei ergebenden Übertrag Ü in die (x+1)te Stelle – hier in die 5. Stelle – „unter den Tisch fallen läßt". Beispiele:

```
  1234 M       1234 M
- 1234 S     + 8766 S''
     Ü         1111  Ü
--------     ---------
    0000         0000

  1234 M       1234 M
- 1122 S     + 8878 S''
     Ü         1111  Ü
--------     ---------
    0112         0112
```

Wenn S weniger Stellen als M hat, so muß man S mit führenden Nullen auffüllen:

```
  1234 M       1234 M
- 0022 S     + 9978 S''
     Ü         1111  Ü
--------     ---------
    1212         1212
```

Obwohl wir uns in diesem Aufsatz auf die natürlichen, sprich positiven Ganzzahlen beschränken wollen, müssen wir auch den Fall M < S betrachten, da er weiter unten bei der 16-minus-16-Bit-Subtraktion im niederwertigen Byte vorkommen kann. Beispiel:

```
  1234 M       1234 M
- 1235 S     + 8765 S''
  111  Ü           Ü
 1           0     N
--------     ---------
  9999         9999
  0001         0001 ''
- 0001        -0001
```

Wenn bei der konventionellen Subtraktion der Nach-Übertrag in die (x+1)te Stelle N = 1 und bei der Komplement-Addition N = 0 ist, so muß von dem Ergebnis das Zweierkomplement gebildet und das Vorzeichen umgekehrt werden.
(Bei der konventionellen Subtraktion könnte man auch b ↑ x, d.h. konkret 10 ↑ 4 = 10000 vom Ergebnis 9999 abziehen, also 9999 - 10000 = -1. In der Praxis schreibt man jedoch nicht 1234 - 1235, sondern 1235 - 1234 = +1, und ändert im nachhinein das Vorzeichen in -1.)

Der Sachverhalt wird deutlicher, wenn wir die Komplement-Addition ziffernweise vornehmen. Betrachten wir hierzu noch einmal den denkbar einfachsten Fall, bei dem bei der konventionellen Subtraktion gar kein Übertrag vorkommt:

```
              321 Stelle
  876 M       876 M
- 123 S     + 877 S''
       Ü     111  Ü
---------   ---------
  753         753
```

Die Komplement-Addition läßt sich von rechts nach links, d.h. von der 1. bis zur 3. Stelle, ziffernweise so durchführen:

```
  3     2     1   Stelle
  8     7     6   M-Ziffer
+ 9   + 8   + 7   S''-Ziffer
  1     1     1   Ü
---   ---   ---
  7     5     3
```

Hier ergab sich für jede Ziffer ein Übertrag von 1. Bei dem nächsten Beispiel erfolgt bei der konventionellen Subtraktion jedoch „zwischendrin" ein Übertrag, der sich gleichwohl nicht als Nach-Übertrag auf die (x+1)te Stelle auswirkt:

```
              123 Stelle
  806 M       806 M
- 123 S     + 877 S''
  1   Ü      1 1  Ü
---------   ---------
  683         683
```

Wenn wir die Komplement-Addition wieder ziffernweise von der 1. bis zur 3. Stelle durchführen, so wird der Übertrag in der 2. Stelle gleich 0, und wir dürfen dann in der 3. Stelle nur das *Einer*komplement und nicht mehr das Zweierkomplement bilden:

```
  3       2       1   Stelle
  8       0       6   M-Ziffer
+ 8(!) + 8     + 7   S'(')-Ziffer
  1       0       1   Ü
---     ---     ---
  6       8       3
```

Damit kommen wir endlich dem Wesen der Komplement-Addition auf die Spur:

1. Stelle: Wir *beginnen mit einem Vor-Übertrag von 1*, bilden das *Einer*komplement von der 1. Subtrahendenstelle **3**, d.h. 6, und zählen den Vor-Übertrag 1 hinzu, womit sich das *Zweier*komplement 7 ergibt. Dann addieren wir 6 (1. Minuendenstelle) und 7 (Zweierkomplement der 1. Subtrahendenstelle): 6 + 7 = 3, Übertrag 1. Wir tragen in der 1. Differenzstelle 3 ein und übernehmen den Übertrag 1 als Nach-Übertrag in die 2. Stelle.

2. Stelle: Wir bilden das Einerkomplement der 2. Subtrahendenstelle **2**, d.h. 7, und zählen den Vor-Übertrag 1 hinzu, womit sich das Zweierkomplement 8 ergibt. Dann addieren wir 0 (2. Minuendenstelle) und 8 (Zweierkomplement der 2. Subtrahendenstelle): 0 + 8 = 8, Übertrag 0. Wir tragen in der 2. Differenzstelle 8 ein und übernehmen den Übertrag 0 als Nach-Übertrag in die 3. Stelle.

3. Stelle: Wir bilden das Einerkomplement der 3. Subtrahendenstelle **1**, d.h. 8, und zählen den Vor-Übertrag 0 hinzu, womit sich effektiv das *Einer*komplement 8 ergibt. Dann addieren wir 8 (3. Minuendenstelle) und 8 (*Einer*komplement der 3. Subtrahendenstelle): 8 + 8 = 6, Übertrag 1. Da keine weitere Stelle mehr vorkommt, ist dies der Nach-Übertrag.

Wenn der Nach-Übertrag 1 ist, so können wir ihn unter den Tisch fallen lassen, denn die Differenz ist eine positive Zahl und damit richtig. Wenn der Nach-Übertrag 0 ist, so ist die Differenz eine negative Zahl und damit falsch. Merksatz zur Komplement-Addition:

Mit Vor-Übertrag 1 beginnen und nachher prüfen, ob Nach-Übertrag 1 ist. Dann ist Differenz korrekt.

4.4.2. Binäre Komplement-Addition

Für das Binärsystem gilt ebenfalls die Formel
z' = (b - 1) - z (siehe 4.4.1)
Bei der binären Basis b = 2 ergibt sich für die zwei Ziffern z:

z = 0, ergo
z' = (2 - 1) - 0
z' = 1 - 0
z' = 1
und
z = 1, ergo
z' = (2 - 1) - 1
z' = 1 - 1
z' = 0

Zunächst bringen wir einige Übungsbeispiele zum Zweierkomplement von mehrstelligen Binärzahlen Z:

```
 11110000 Z
 00001111 Z'
+00000001
----------
 00010000 Z''

 01010101 Z
 10101010 Z'
+00000001
----------
 10101011 Z''

 00000000 Z
 11111111 Z'
+00000001
----------
100000000 Z''
```

In diesem letzten Fall führt das Zweierkomplement zu einem Überlauf. Betrachten wir nunmehr das Subtraktionsbeispiel, mit dem wir den Abschnitt 4.3 abgebrochen haben (M = Minuend, S = Subtrahend, Ü = Übertrag):

```
   10000000 00000000 M (32768)
-  01111111 11111111 S (32767)
-------------------------------
   00000000 00000001   (00001)
```

Wir formen den Subtrahenden zum Zweierkomplement um

```
   01111111 11111111 S
   10000000 00000000 S'
   10000000 00000001 S''
```

und führen dann die Zweierkomplement-Addition durch

```
   10000000 00000000 M
+  10000000 00000001 S''
  1                  Ü
-------------------------
   00000000 00000001
```

Wenn wir statt dessen eine Einerkomplement-Addition mit einem Vor-Übertrag von 1 vornehmen, ergibt sich folgende Aufstellung:

```
   10000000 00000000 M
+  10000000 00000000 S'
  1                1 Ü
-------------------------
   00000000 00000001
```

Betrachten wir abschließend den Fall einer 16-minus-16-Bit-Subtraktion, bei der vom niederwertigen zum höherwertigen Byte (a) bei der konventionellen Subtraktion ein Übertrag von 1 und (b) bei der Komplement-Addition ein Übertrag von 0 erfolgt, wobei wir zunächst die konventionelle Subtraktion voranstellen (V = Vor-Übertrag, N = Nach-Übertrag):

```
   11111111 00000000 M ( 65280)
-  00001111 11111111 S ( 04095)
         1         0 V (     0) V=0!
      1111  1111111  Ü (   11 )
  0                 N (0      ) N=0!
-------------------------------------
   11101111 00000001   ( 61185)
```

Als Einerkomplement-Addition mit Vor-Übertrag 1 ergibt sich folgende Aufstellung:

```
   11111111 00000000 M  ( 65280)
+  11110000 00000000 S' ( 95904)
         0         1 V (      1) V=1!
   111               Ü  ( 11   )
  1                  N  (1     ) N=1!
-------------------------------------
   11101111 00000001    ( 61185)
```

4.4.3. 6502-Komplement-Addition

Wir können nunmehr die 6502-Subtraktion SEC SBC als SEC ACC („Add Complement with Carry") begreifen:

1. Durch SEC wird der Vor-Übertrag auf 1 gesetzt.

2. Wir laden den Akkumulator A mit dem Subtrahenden, „EORen" ihn zum Einerkomplement und zählen die Speicherstelle M (= Memory) hinzu. Da durch SEC das Carry-Flag auf 1 gesetzt wird, entspricht der SBC-Befehl praktisch einer Zweierkomplement-Addition.

3. Wenn nach der SBC- bzw. „ACC"-Operation das Carry-Bit bzw. der Nach-Übertrag 1 ist, wurde die Subtraktion ohne negativen Überlauf ausgeführt.

Formelmäßig lassen sich die drei Schritte so zusammenfassen:
A ← A' + M + 1 oder
A ← A'' + M

Der Befehl
EOR #%11111111
invertiert das Bit-Muster von A und erzeugt damit ein Einerkomplement, z.B.:

10101110 Bit-Muster
11111111 EOR-Maske
01010001 Einerkomplement

Die 6502-Subtraktion
SEC
LDA MINUEND
SBC SUBTRAHEND
BCC FEHLER
BCS ENDE
läßt sich folglich so simulieren:
SEC
LDA SUBTRAHEND
EOR #%11111111
ADC MINUEND
BCC FEHLER
BCS ENDE

Man könnte also auf den Subtraktionsbefehl völlig verzichten. Das nachfolgende Demo, das auf der Peeker-Sammeldisk unter dem Namen **KOMPLEMENT.DEMO** enthalten ist, veranschaulicht dies:

```
10 PRINT CHR$(4)"BLOAD KOMPLE-
   MENT"
20 INPUT "Minuend:";M: POKE 768,M
30 INPUT "Subtrahend: ";S: POKE 769,S
40 CALL 770
50 PRINT M;"-";S;"=";M - S
60 PRINT : GOTO 20
```

Gibt man nach
RUN KOMPLEMENT.DEMO
2 und 1 für Subtrahend und Minuend ein, so wird
01=01
2 - 1 = 1
angezeigt. Gibt man jedoch z.B. 1 und 2 ein, so wird
-FF=-FF
1 - 2 = -1

angezeigt. Dabei steht -FF für die interne, d.h. (noch) nicht komplementierte -1.

```
1           ORG  $0300
2  *
3  * KOMPLEMENT
4  * ==========
5  *
6  MINUEND  HEX  60
7  SUBTRAH  HEX  30
8  *
9  COUT     EQU  $FDED
10 CROUT    EQU  $FD8E
11 PRBYTE   EQU  $FDDA
12 * Normale Subtraktion
13 NORMAL   SEC
14          LDA  MINUEND
15          SBC  SUBTRAH
16          PHA
17          BCS  NORMAL1
18          LDA  #"-"
19          JSR  COUT
20 NORMAL1  PLA
21          JSR  PRBYTE
22          LDA  #"="
23          JSR  COUT
24 * Zweierkomplement-Addition
25 KOMPLEM  CLC
26          LDA  SUBTRAH
27          EOR  #%11111111
28          ADC  #1
29          ADC  MINUEND
30          PHA
31          BCS  KOMPLEM1
32          LDA  #"-"
33          JSR  COUT
34 KOMPLEM1 PLA
35          JSR  PRBYTE
36          JMP  CROUT
```

Der Vollständigkeit halber sei abschließend erwähnt, daß die 6502-Subtraktion in verschiedenen Büchern durch die Formel

A ← A - M - (1 - C)

erklärt und in diesem Zusammenhang C als komplementiertes Borgen interpretiert wird. Wenn C = 1 ist, ist (1 - 1) = 0, und C wird nicht zum Subtrahenden hinzugezählt. Wenn C = 0 ist, ist (1 - 0) = 1, und C wird zum Subtrahenden hinzugezählt. Dies hat indes mit der Papier- und Bleistift-Methode wenig gemein.

Literatur

Klaus D. Sänger: Rechnen aufgefrischt, Buch-und-Zeit-Verlag, Köln 1982. (Behandelt in leichtverständlicher Form die dezimalen Papier- und Bleistift-Rechenverfahren.)

E.Pracht/K.Heidenreich: Elementare Zahlentheorie (UTB). Schöningh-Verlag, Paderborn 1978. (Enthält eine recht detaillierte Darstellung der Rechenverfahren für g-adische Stellenwertsysteme, z.B. Komplement-Addition auf S. 127f. Infolge des permanenten Wechsels zwischen exotischen Zahlensystemen mit den Basen 3, 7, 2, -2 (!) usw. für Nicht-Mathematiker schwer verständlich.)

R.Männel/M.Kohler: Algebra für Wirtschaftsschulen. 15. Aufl., Gehlen-Verlag, Bad Homburg 1980. (Eines der zahlreichen Schulbücher, das sich auch mit dem dualen Rechnen befaßt, hier S. 80ff. Lösungheft mit anfordern!)

A.Osborne/D.Bunnell: An Introduction to Microcomputers. Band 0. 3. Aufl., Osborne/McGraw-Hill-Verlag, Berkeley 1982. (Enthält auf S. 89-132 eine gute Einführung in das binäre Rechnen, allerdings beschränkt auf Addition und Subtraktion.)

Lance A.Leventhal/W.Saville: 6502 Assembly Language Subroutines. Osborne/McGraw-Hill-Verlag, Berkeley 1982. (Enhält auf S. 230-305 alle wichtigen 6502-Algorithmen für binäre Addition, Subtraktion, Multiplikation und Division, allerdings ohne nähere Erläuterungen.)

Hinweis: Der zweite Teil, der im Peeker 3/86 erscheinen wird, befaßt sich mit der Multiplikation und Division.

Nr. 4/86 April
DM 6.50, sfr 6.50, öS 50, Lit 5900, hfl 7.50

PEEKER

Binäres Rechnen
Multiplikation und Division

Harddisk Controller
Aufbau und Arbeitsweise

Sonderzeichen
auf dem Imagewriter

Kyan-Pascal
und Assembler

Zweistimmige Melodien
auf dem Apple

Hüthig
PUBLIKATION

Binäres Rech mit Papier un

Teil 2: Multiplikation und Division

von Ulrich Stiehl

Dieser zweite Teil setzt voraus, daß Sie den Stoff aus dem ersten Teil (Peeker 2/86) bereits kennen.

Gliederung

5. Multiplikation

5.1. Dezimale Multiplikation

Multiplizieren heißt malnehmen oder vervielfachen. Der Ausdruck
$7 * 5 = 35$
im Sinne von „siebenmal die Fünf" (und *nicht* „die Sieben fünfmal") läßt sich in die Addition
$5 + 5 + 5 + 5 + 5 + 5 + 5 = 35$
überführen, wobei 7 als Multiplikator MR (Malzahl), 5 als Multiplikand MD (malgenommene Zahl) und 35 als Produkt P (Ergebnis) bezeichnet werden. Bei der mündlichen Multiplikation wird *zuerst* der

nen
d Bleistift

Multiplikator, bei der schriftlichen Multiplikation *zuerst* der Multiplikand genannt, also

```
mündlich:     MR      *  MD   =   P
              7       *  5    =   35
schriftlich:  MD      *  MR   =   P
              98765   *  7    =   691355
```

Dies steht im Widerspruch zu mir bekannten Schulbüchern und beruht offenbar auf einem tradierten Mißverständnis, über das in der Zwischenzeit keiner mehr nachgedacht hat. Rechnen Sie einmal „98765 * 7" schriftlich aus und murmeln Sie das, was Sie dabei denken, vor sich hin. Sie sagen „7 * 5, 7 * 6, 7 * 7" usw. und nicht „98765 * 7", denn sonst wären Sie Adam Ries!

Obwohl wir natürlich mit der schriftlichen Multiplikation vertraut sind, ist es nicht einfach, den sich dahinter verbergenden Algorithmus ins Bewußtsein zu heben. Gerade dies müssen wir jedoch tun, wenn wir die binäre Multiplikation im allgemeinen und die 6502-Multiplikation im besonderen verstehen wollen. Hierzu ist es erforderlich, die Phasen oder Komponenten der Multiplikation isoliert zu betrachten, wobei auch Dinge beachtet werden müssen, die man bei der konventionellen Papier- und Bleistiftmethode außer acht lassen würde.

Beispiel: Wenn Sie auf einem weißen Blatt nur eine einzige schriftliche Multiplikation durchzuführen hätten, stünde Ihnen genug „freier Raum" zur Verfügung. Wenn die Aufgabe jedoch lauten würde, auf einem karierten Papier bestimmter Größe möglichst viele 7-mal-5-stellige Multiplikationen unterzubringen, müßten Sie sich schon Gedanken über die Feldlängen der Teil- und Endprodukte machen, denn Sie würden z.B. links und rechts nicht genügend Platz haben, um die Teilprodukte „beliebig" zu verschieben.

5.1.1. Zerlegungsverfahren

Die schriftliche Multiplikation setzt das (auswendig gelernte) kleine Einmaleins voraus. Die Aufgabe
98765 * 7
ließe sich nicht lösen, wenn wir nicht bereits die Produkte von 7 * 5, 7 * 6, 7 * 7 usw. wüßten. Man kann diese Multiplikation aber auch in eine reine Addition umwandeln:

```
  98765 1mal
+ 98765 2mal
+ 98765 3mal
+ 98765 4mal
+ 98765 5mal
+ 98765 6mal
+ 98765 7mal
------------
 691355 Summe = Produkt
```

Im Falle eines einstelligen Multiplikators wäre dieses Verfahren gerade noch angängig. Für die Aufgabe
77777 * 98765
würde man jedoch bereits einen Karton Papier benötigen. Also müssen wir uns bei mehrstelligen Zahlen Stelle für Stelle vorarbeiten, was als Zerlegung (in Einer, Zehner, Hunderter, Tausender usw.) bezeichnet wird. Beispiel:

```
     HZE
77 * 321
---------      MR   MD        MR    MD
    77 =        1 * 77 oder 1 *    77
   154x =      20 * 77 oder 2 *   770
  231xx =     300 * 77 oder 3 * 7700
---------
 24717
```

Wie ersichtlich, gibt es zwei Interpretationen. So kann etwa die Multiplikation des Multiplikator-Zehners 2 mit dem Multiplikanden 77 als
2(0) * 77 oder als
2 * 77(0)
aufgefaßt werden. Für die Zehnerstelle können wir aber auch (2 * 77) * 10 schreiben, d.h. allgemein formuliert:

(MR-Stellenwert * MD) * 10 ↑ MR-Stelle.

Bei der Papier-und-Bleistift-Multiplikation geht man nun noch einen Schritt weiter und zerlegt nicht nur den Multiplikator, sondern auch den Multiplikanden, denn man rechnet nicht 2 * 77, sondern
2 * 7 = 4; Übertrag 1
2 * 7 = 14; 14 + 1 = 15.

Anstelle der Zehnerstelle-Multiplikation 2(0) * 77 können wir auch die 154 als Teilprodukt von 2 * 77 nach links schieben (154x), was durch das „x" in dem obigen Beispiel angedeutet wird. Jede Verschiebung um eine Stelle nach links entspricht einer Multiplikation mit 10 (oder allgemein einer Multiplikation mit der Basis des jeweiligen Zahlensystems):

```
  7 =   7 = 7 * (10 ↑ 0)
 7x =  70 = 7 * (10 ↑ 1)
7xx = 700 = 7 * (10 ↑ 2)
```

Wir merken uns:

1. Entweder schieben wir den Multiplikanden nach links und multiplizieren *danach* mit der Multiplikatorstelle (= **Multiplikandenverschiebung**),

2. oder wir multiplizieren den Multiplikanden mit der Multiplikatorstelle und schieben *danach* das Teilprodukt nach links (= **Teilproduktverschiebung**).

5.1.2. Links- und Rechtsverfahren

Man kann beim Multiplikator mit der links stehenden, höchsten Stelle (z.B. hier mit der Tausenderstelle) beginnen. Dies nennen wir **Linksverfahren** oder genauer Von-links-nach-rechts-Verfahren. Mit jeder weiteren Multiplikatorstelle, die wir von links wegnehmen, müssen wir das entsprechende Teilprodukt um eine Stelle nach rechts einrücken. Die Teilprodukte sind damit treppenförmig von links nach rechts eingerückt.

Bei der Assemblerprogrammierung kann man das Wegnehmen einer Multiplikatorstelle durch Verschiebebefehle realisieren. Wenn man eine Linksverschiebung (ASL oder ROL) vornimmt, wird das jeweils höchste Bit, also die höchste Multiplikatorstelle „weggenommen" (= in das Carry-Flag übertragen).

Linksverfahren-Beispiel

```
MD     MR
1111 * 4321
-----------
   x4444xxx T   ZW: 04444000 T
   xx3333xx H   ZW: 04777300 T+H
   xxx2222x Z   ZW: 04799520 T+H+Z
   xxxx1111 E   ZW: 04800631 T+H+Z+E
-----------
   x4800631 P
```

Umgekehrt kann man beim Multiplikator mit der rechts stehenden, niedrigsten Stelle (regelmäßig mit der Einerstelle) beginnen. Dies nennen wir **Rechtsverfahren** oder genauer Von-rechts-nach-links-Verfahren. Mit jeder weiteren Multiplikatorstelle, die wir von rechts wegnehmen, müssen wir das entsprechende Teilprodukt um eine Stelle nach links einrücken. Die Teilprodukte sind damit treppenförmig von rechts nach links eingerückt.

Bei der Assemblerprogrammierung kann man das Wegnehmen einer Multiplikatorstelle durch Verschiebebefehle realisieren. Wenn man eine Rechtsverschiebung (LSR oder ROR) vornimmt, wird das jeweils niedrigste Bit, also die niedrigste Multiplikatorstelle „weggenommen" (= in das Carry-Flag übertragen).

Rechtsverfahren-Beispiel

```
MD     MR
1111 * 4321
-----------
   xxxx1111 E   ZW: 0000111 1 E
   xxx2222x Z   ZW: 000233 31 E+Z
   xx3333xx H   ZW: 00356 631 E+Z+H
   x4444xxx T   ZW: 0480 0631 E+Z+H+T
-----------
   x4800631 P
```

Unter **Teilprodukt** versteht man das Produkt einer Multiplikatorstelle mit dem Multiplikanden (unter Berücksichtigung der Stellenverschiebung!). Unter **Zwischensumme** versteht man die Summe der bislang errechneten Teilprodukte. Unter Endprodukt oder kurz Produkt (P) versteht man die Summe aller Teilprodukte oder die letzte Zwischensumme. Wir haben hier im Vorgriff auf den nächsten Abschnitt zusätzlich die Zwischensummen (ZW) ausgewiesen.

Bei dem Rechtsverfahren bemerken wir übrigens, daß jede weitere Zwischensumme eine weitere gültige Endziffer aufweist.

Um zu einem Algorithmus zu gelangen, muß man sich Gedanken über die Stellenanzahl oder **Zahlenfeldlänge** machen. Betrachten wir hierzu die Multiplikation:

```
4321   4321   87654321 Stellen
9999 * 9999 = 99980001 Zahlen
```

Wir ersehen, daß die 4-mal-4-stellige Multiplikation höchstens zu einem 8stelligen Produkt führt. Allgemein gilt, daß bei einer x-mal-x-stelligen Multiplikation ein x-plus-x-stelliges Produktfeld immer ausreicht. Wenn wir die Teilprodukte der E-, Z-, H- und T-Multiplikationen in einem 8stelligen Feld ausrichten und dabei das Verschieben der Teilprodukte analysieren, so können wir zwei Phänomene erkennen:

1 Bei dem Rechtsverfahren wird das erste Teilprodukt (für die E-Multiplikation) zunächst *ganz rechtsbündig* ausgerichtet; danach wird mit jeder weiteren Multiplikatorstelle jedes weitere Teilprodukt um eine Stelle nach links gerückt. Demgegenüber wird bei dem Linksverfahren das erste Teilprodukt (hier für die T-Multiplikation) *nicht ganz linksbündig*, d.h. um eine Stelle von der linken Kante entfernt, ausgerichtet; danach wird mit jeder weiteren Multiplikatorstelle jedes weitere Teilprodukt um eine Stelle nach rechts gerückt.

2 Bei einem 4stelligen oder allgemein x-stelligen Multiplikator muß das Teilprodukt 3mal oder allgemein (x-1)-mal nach links oder rechts geschoben werden.

5.1.3. Normalverfahren

Bei dem obigen Beispiel (5.1.2) wurden die E-, Z-, H- und T-Teilprodukte *direkt* untereinandergeschrieben. Allgemein führt ein x-stelliger Multiplikator zu x Teilprodukten. Bei der computermäßigen Multiplikation muß man hingegen *zusätzlich* Zwischensummen bilden, weil sonst zuviel Speicherplatz verschwendet wird – eine 24-mal-24-Bit-Multiplikation benötigt 72 Bytes für die Teilprodukte –, zumal das Ablegen der Teilprodukte im Speicher (fast) solange dauert wie das Aufaddieren zur Zwischensumme.

Bei dem Rechtsverfahren würde man folgende Zwischensummen ZW bilden:

```
1111 * 4321
-----------
   00000000 0. ZW
 + 00001111 E
-----------
   00001111 1. ZW
 + 0002222x Z
-----------
   00023331 2. ZW
 + 003333xx H
-----------
   00356631 3. ZW
 + 04444xxx T
-----------
   04800631 4. ZW = P
===========
```

Für das Rechtsverfahren gilt:
0. ZW = 0
1. ZW = E-P (Einerprodukt)
2. ZW = E-P + Z-P
3. ZW = E-P + Z-P + H-P
4. ZW = E-P + Z-P + H-P + T-P
Die letzte Zwischensumme ist gleichzeitig das Gesamtprodukt der Multiplikation.

Für das Linksverfahren gilt:
0. ZW = 0
1. ZW = T-P (Tausenderprodukt)
2. ZW = T-P + H-P
3. ZW = T-P + H-P + Z-P
4. ZW = T-P + H-P + Z-P + E-P

Das soeben skizzierte Verfahren bezeichnen wir als **Normalverfahren** oder genauer als normales Zwischensummenverfahren. Das besondere Merkmal dieser Methode ist, daß die Position der Zwischensummen unverändert bleibt und nur das jeweilige Teilprodukt nach rechts (= normales Rechts[zwischensummen]verfahren) oder nach links (normales Links[zwischensummen]verfahren) verschoben wird. Die Teilproduktverschiebung kann auch als Multiplikandenverschiebung gedeutet werden (s. 5.1.1.). Wir merken uns deshalb:
Beim Normalverfahren wird das Teilprodukt oder der Multiplikand, aber niemals die Zwischensumme selbst verschoben.

5.1.4. Tausendeinsverfahren

Diesen Abschnitt sollten Sie ggf. überspringen, da jetzt eine Methode vorgestellt wird, die mit „Papier und Bleistift" nichts zu tun hat.
Wie bereits eingangs bemerkt, sind die bekannten 6502-Einführungen bezüglich der Binärmathematik völlig undidaktisch aufgebaut. Dies wird an den dort präsentierten Beispielen für die 8-mal-8-Bit-Multiplikation besonders deutlich. In dem Buch „Kehrel: Apple Assembler lernen" findet sich der weiter unten als Tausendeins-Linksverfahren bezeichnete Algorithmus, der in identischer Form in „Inman: Apple Machine Language", in „Leventhal: 6502 Assembly Language Programming" und ähnlichen Einführungen auftaucht. In all diesen Lehrbüchern wird dieser exotische Algorithmus kommentarlos in den Raum gestellt – wahrlich eine didaktische Zumutung, zumal es sich nicht einmal um den schnellsten Algorithmus handelt!
Was hat es nun mit dem Tausendeinsverfahren auf sich? Betrachten wir zu diesem Zweck den Anfang der folgenden Multiplikation nach dem „anomalen" Links[zwischensummen]verfahren (= Tausendeins-Linksverfahren):

```
0001 * 1000
-----------
   00000000 0. ZW
 + 00000001 1000 * 1!
-----------
   00000001 ???
```

Ist 1000 * 1 = 1? Dies kann doch wohl nicht stimmen! Was ist geschehen? Während wir beim normalen Links[zwischensummen]verfahren den Multiplikanden oder das Teilprodukt verschieben, wird

beim anomalen Tausendeins-Linksverfahren die Zwischensumme selbst verschoben. *Deshalb ist jede Zwischensumme bis auf die letzte falsch.* Um dies begreifbar zu machen, stellen wir Tausendeins- und Normalverfahren einander gegenüber. Zur Verdeutlichung fügen wir eine zusätzliche Verschiebungszeile ein („Shift"), die beim 1001-Verfahren die Verschiebung der Zwischensumme und beim Normalverfahren die Verschiebung des Multiplikanden anzeigt. Beim 1001-Verfahren muß man deshalb jeweils die 2. und 3. und beim Normalverfahren die 1. und 3. Zeile addieren:

Rechtsverfahren

```
anomal 1001          normal

2345 * 6789          2345 * 6789
-----------          -----------
  0000 0000            0000 0000  0. ZW
  0000 0000            xxxx 2345  kein Shift
+21105               +    2 1105  MD * 9
-----------          -----------
 21105 0000           0002 1105  1. ZW
> 2110 5000          < xxx2 345x  Shift
+18760               +   18 7600  MD * 8(0)
-----------          -----------
 20870 5000           0020 8705  2. ZW
> 2087 0500          < xx23 45xx  Shift
+16415               +  164 150   MD * 7(00)
-----------          -----------
 18502 0500           0185 0205  3. ZW
> 1850 2050          < x234 5xxx  Shift
+14070               + 1407 0000  MD * 6(000)
-----------          -----------
 15920 2050           1592 0205  4. ZW
> 1592 0205          ===========  Shift
===========
```

Zur Verdeutlichung haben wir 2 Vierergruppen gebildet, die linke und die rechte Vierergruppe.

Beim *normalen Rechtsverfahren* wird der Multiplikand zunächst in der rechten Vierergruppe eingesetzt. *Nach* jeder Addition – mit Ausnahme der letzten – wird der Multiplikand um 1 Stelle nach links verschoben [<–]. Es finden damit 4 Additionen und 3 Linksverschiebungen des Multiplikanden statt:

A-L-A-L-A-L-A

Da der Multiplikand infolge der sukzessiven Linksverschiebung die beiden Vierergruppen überlappt, muß die Addition stets beide Vierergruppen erfassen.

Beim *anomalen Rechtsverfahren* (1001-Methode) wird der unverschobene Multiplikand mit der jeweiligen Multiplikatorstelle malgenommen und das sich ergebende Teilprodukt stets in der linken Vierergruppe eingetragen. Da die 4 Stellen der linken Vierergruppe meist nicht ausreichen, wird eine zusätzliche Übertrag-Stelle eingerichtet. Die linke Vierergruppe ist mithin eigentlich eine „Fünfergruppe". *Nach* jeder Addition – einschließlich der letzten – wird die sich ergebende Zwischensumme um 1 Stelle nach rechts verschoben [–>], womit der Übertrag wieder in die linke Vierergruppe hineingeschoben wird. Es finden damit 4 Additionen und 4 Rechtsverschiebungen statt:

A-R-A-R-A-R-A-R

Da der Multiplikand nicht verschoben wird, muß die Addition nur die beiden linken Vierergruppen erfassen, wobei jedoch meist ein Übertrag zu berücksichtigen ist.

Linksverfahren

```
anomal 1001          normal

2345 * 6789          2345 * 6789
-----------          -----------
  0000 0000            0000 0000  0. ZW
  0000 0000          > x234 5xxx  Shift
+    1 4070          + 1407 0000  MD * 6(000)
-----------          -----------
  0001 4070            1407 0000  1. ZW
< 0014 0700          > xx23 45xx  Shift
+    1 6415          +  164 150   MD * 7(00)
-----------          -----------
  0015 7115            1571 1500  2. ZW
< 0157 1150          > xxx2 345x  Shift
+    1 8760          +   18 7600  MD * 8(0)
-----------          -----------
  0158 9910            1589 9100  3. ZW
< 1589 9100          > xxxx 2345  Shift
+    2 1105          +    2 1105  MD * 9(0)
-----------          -----------
  1592 0205            1592 0205  4. ZW
===========          ===========
```

Beim *normalen Linksverfahren* wird der Multiplikand zunächst in der linken Vierergruppe eingetragen und bereits *vor* der ersten Addition um 1 Stelle nach rechts verschoben [–>]. Es finden damit 4 Rechtsverschiebungen und 4 Additionen statt:

R-A-R-A-R-A-R-A

Da der Multiplikand infolge der sukzessiven Rechtsverschiebung die beiden Vie-

rergruppen überlappt, muß die Addition stets beide Vierergruppen erfassen.
Beim *anomalen Linksverfahren* wird der unverschobene Multiplikand mit der jeweiligen Multiplikatorstelle malgenommen und stets in der rechten Vierergruppe eingetragen. Da die 4 Stellen der rechten Vierergruppe meist nicht ausreichen, erfolgt ein Übertrag in die erste Stelle der linken Vierergruppe. Dies ist normal, denn anders als beim anomalen Rechtsverfahren braucht keine zusätzliche Übertrag-Stelle eingerichtet zu werden. Nach jeder Addition – mit Ausnahme der letzten – erfolgt eine Linksverschiebung [<–]. Es finden damit 4 Additionen und 3 Linksverschiebungen statt:
A-L-A-L-A-L-A
Da der Multiplikand nicht verschoben wird, muß die Addition nur die beiden rechten Vierergruppen erfassen, wobei jedoch meist ein Übertrag zu berücksichtigen ist.

5.2. Binäre Multiplikation

Für die binäre Multiplikation gelten folgende Grundregeln:

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

5.2.1. Zerlegungsverfahren

Während bei der dezimalen Multiplikation das kleine Einmaleins beherrscht werden muß, reduziert sich das binäre Zerlegungsverfahren auf die Frage, ob der Multiplikand mit 1 oder mit 0 multipliziert werden soll, oder anders formuliert, ob der *unveränderte*, wenngleich stellenverschobene Multiplikand oder Null als Teilprodukt notiert werden soll. Fazit: Die Multiplikation des Multiplikanden mit einem Einserbit des Multiplikators läuft auf das stellenverschobene *Abschreiben* des Multiplikanden hinaus. Beispiele:

```
1111 * 11
---------
    1111
   1111

1111 * 00
---------
    0000
   0000
```

5.2.2. Links- und Rechtsverfahren

Die Links- und Rechtsverfahren sind analog anwendbar. Bei den nachfolgenden Beispielen sind wieder die Überträge als Additionshilfen vermerkt:

Linksverfahren-Beispiel

```
MD      MR
1011 * 1110   (11 * 14)
-----------
   x1011000   1 * 1011000
   xx101100   1 *  101100
   xxx10110   1 *   10110
   xxxx0000   0 *    1011
-----------
   11111      Übertrag
-----------
   10011010   (154)
===========
```

Rechtsverfahren-Beispiel

```
MD      MR
1111 * 1011   (15 * 11)
-----------
   xxxx1111   1 *    1111
   xxx11110   1 *   11110
   xx000000   0 *  111100
   x1111000   1 * 1111000
-----------
     1111     Übertrag
     1111     Übertrag
-----------
   10100101   (165)
===========
```

Wie ersichtlich, führt eine 4-mal-4-Bit-Multiplikation höchstens zu einem 8stelligen Produkt. Allgemein gilt, daß eine x-plus-x-stellige Zahlenfeldlänge für ein x-mal-x-Bit-Produkt stets ausreichend ist.

5.2.3. Normalverfahren

Auch das normale Zwischensummenverfahren ist analog anwendbar. Bei dem nachfolgenden Beispiel für eine 8-mal-8-Bit-Multiplikation im Rechtsverfahren stehen jeweils Zwischensumme (ZW) und Teilprodukt als 16-Bit-Zahlen untereinander, wobei neben dem Teilprodukt der 8-Bit-Multiplikator (MR) mit der rechts herausgeschobenen Multiplikatorstelle notiert wird.

```
MD        * MR
00001111 * 10101010
15        * 170

 00000000 00000000   0. ZW
+00000000 00000000   x1010101-0 MR
-------------------
 00000000 00000000   1. ZW
+00000000 00011110   xx101010-1 MR
-------------------
 00000000 00011110   2. ZW.
+00000000 00000000   xxx10101-0 MR
-------------------
 00000000 00011110   3. ZW
+00000000 01111000   xxxx1010-1 MR
-------------------
 00000000 10010110   4. ZW
+00000000 00000000   xxxxx101-0 MR
-------------------
 00000000 10010110   5. ZW
+00000001 11100000   xxxxxx10-1 MR
-------------------
 00000010 01110110   6. ZW
+00000000 00000000   xxxxxxx1-0 MR
-------------------
 00000010 01110110   7. ZW
+00000111 10000000   xxxxxxxx-1 MR
-------------------
 00001001 11110110   8. ZW
===================
```

5.2.4. Tausendeinsverfahren

Das Tausendeinsverfahren ist aus programmiertechnischer Sicht – nicht aus der Papier-und-Bleistift-Sicht! – effizienter als das Normalverfahren. Der Abschnitt über das dezimale Tausendeinsverfahren (5.1.4) wird hier vorausgesetzt. Zunächst ein Beispiel:

```
Rechtsverf.          Linksverf.

1111 * 1011          1111 * 1011
-----------          -----------
  0000 0000            0000 0000
  0000 0000          < 0000 0000
+ 1111               +      1111   MD
  0                          0     Übertrag
-----------          -----------
  01111 0000           0000 1111
> 0111 1000         < 0001 1110
+ 1111               +      0000   MD
  1                          0     Übertrag
-----------          -----------
  10110 1000           0001 1110
> 1011 0100         < 0011 1100
+ 0000               +      1111   MD
  0                          1     Übertrag
-----------          -----------
  01011 0100           0100 1011
> 0101 1010         < 1001 0110
+ 1111               +      1111   MD * 0
  1                          1     Übertrag
-----------          -----------
  10100 1010           1010 0101
> 1010 0101
==========
```

In Abweichung zum dezimalen Tausendeinsverfahren können wir folgendes konstatieren:
1. Beim binären Rechts/Linksverfahren paßt das Teilprodukt exakt in die linke/rechte Vierergruppe, während beim dezimalen Rechts/Linksverfahren eine „Fünfergruppe", d.h. ein 5stelliges Feld vorgesehen werden muß.
2. Erst wenn die Zwischensumme gebildet wird, ist beim binären Tausendeinsverfahren ein Übertrag zu erwarten. Beim Rechtsverfahren geht der Übertrag in die 9. Stelle (externer Übertrag) und beim Linksverfahren in die 5. Stelle (interner Übertrag).
3. Beim Rechtsverfahren kann sich die Addition auf die linke Vierergruppe beschränken. Ein ggf. entstehender externer Übertrag wird durch die anschließende Rechtsverschiebung wieder in die linke Vierergruppe hineingeschoben.
4. Beim Linksverfahren kann sich die Addition auf die rechte Vierergruppe beschränken. Ein ggf. entstehender interner Übertrag führt bei der linken Vierergruppe höchstens zu einer Addition von 1 (wozu programmtechnisch ein Inkrement-Befehl ausreicht).

Wenn wir das Beispiel generalisieren und anstelle der Vierergruppen von „Achtergruppen" (= Bytes) sprechen, haben wir den Schlüssel zur binären Multiplikation nach dem Tausendeinsverfahren gefunden.

5.3. 6502-Multiplikation

Während die meisten 16-Bit-Prozessoren über mehr oder weniger komfortable Multiplikationsbefehle verfügen, muß bei 8-Bit-Prozessoren die Multiplikation auf Additions- und Schiebebefehle reduziert werden. Dies kostet natürlich Prozessortakte. Beispielsweise benötigt beim 68000 eine 16-mal-16-Bit-Multiplikation unter Verwendung des MULU-Befehls ca. 100 Takte, aber ohne Verwendung des MULU-Befehls, d.h. konventionell mit ADD-Befehlen usw., ca. 550 Takte. Ebenfalls ca. 550 Takte werden bei der 6502-16-mal-16-Bit-Multiplikation im optimierten Tausendeinsverfahren benötigt. Eine präzise Bestimmung der Takte ist indes nicht möglich, weil die Multiplikationsdauer u.a. von der Anzahl der Einserbits des Multiplikators abhängt.

Es gibt prinzipiell vier Multiplikationsalgorithmen:
1. MRLN = Von-rechts-nach-links-Normalverfahren.
2. MLRN = Von-links-nach-rechts-Normalverfahren
3. MRLT = Von-rechts-nach-links-Tausendeinsverfahren
4. MLRT = Von-links-nach-rechts-Tausendeinsverfahren

Nachfolgend stellen wir alle vier Varianten für die 8-mal-8-Bit-Multiplikation vor. Die Programme setzen die Beherrschung der 6502-Befehle ASL, LSR, ROL und ROR voraus, die aus Platzgründen an dieser Stelle nicht erläutert werden können. Es lassen sich folgende Schritte unterscheiden:

Schritt 1: Zunächst muß das spätere Produkt auf 0 gesetzt werden. Falls Multiplikator und Multiplikand erhalten bleiben sollen, müssen Shift-Multiplikator (und beim Normalverfahren Shift-Multiplikand) initialisiert werden. Man beachte, daß beim normalen Linksverfahren MDL in SMDH eingetragen wird (s. Listing).
Schritt 2: Als nächstes muß der Bit-Zähler für die (hier 8) Multiplikator-Bits gesetzt werden.
Schritt 3: Bei den Rechtsverfahren wird jetzt der Multiplikator rechtsverschoben. Bei den Linksverfahren wird jetzt entweder der Multiplikand rechtsverschoben (Normalverfahren) oder das Produkt (= momentane Zwischensumme und späteres Endprodukt) linksverschoben (Tausendeinsverfahren).
Schritt 4: Bei den Rechtsverfahren wird jetzt im Falle eines gesetzten Carry-Flags die Addition durchgeführt; man beachte die unterschiedlichen Additionen! Bei den Linksverfahren wird jetzt der Multiplikator linksverschoben.
Schritt 5: Bei den Rechtsverfahren wird jetzt der Multiplikand linksverschoben (Normalverfahren) oder das Produkt rechtsverschoben (Tausendeinsverfahren). Bei den Linksverfahren wird jetzt im Falle eines gesetzten Carry-Flags die Addition durchgeführt; man beachte die unterschiedlichen Additionen!
Schritt 6: Jetzt wird geprüft, ob alle (hier alle 8) Bits des Multiplikators erledigt sind. Wenn nein, dann zurück zu Schritt 3. Wenn ja, dann Ende.

Bemerkungen zu M8RLN/M8LRN

Wenn Sie M8RLN genau untersuchen, werden Sie feststellen, daß die *letzte* (hier 8.) Linksverschiebung des Shift-Multiplikanden (SMD) in Zeile 44 unnötig ist. Derartige „Leerverschiebungen" am Anfang oder Ende der Schleife nimmt man wegen der Kürze des Programms oft in Kauf.
Wie ersichtlich, muß hier bei M8RLN und M8LRN die Addition Low- und High-Bytes gleichermaßen einbeziehen. Deshalb dauert das Normalverfahren länger als das Tausendeinsverfahren.

Bemerkungen zu M8RLT/M8LRT

Wenn Sie M8LRT genau untersuchen, werden Sie feststellen, daß die erste Linksverschiebung des Produkts in Zeile 29 unnötig ist („Leerverschiebung").
Bei M8LRT tritt in Zeile 37 der BCC-INC-Befehl auf, der bei M8RLT fehlt. Deshalb ist M8RLT nicht nur der kürzeste, sondern auch der schnellste Algorithmus.

Erkennungsmerkmale

Rechtsverfahren erkennen Sie daran, daß der Multiplikator mit LSR oder ROR nach *rechts* verschoben wird. Sinngemäß erkennen Sie Linksverfahren daran, daß der Multiplikator mit ASL oder ROL nach *links* verschoben wird.
Normalverfahren erkennen Sie daran, daß die *Rechts*verschiebung des Multiplikators mit der *Links*verschiebung des Multiplikanden gekoppelt ist und umgekehrt. Sinngemäß erkennen Sie Tausendeinsverfahren daran, daß die *Rechts*verschiebung des Multiplikators mit der *Rechts*verschiebung des Produkts gekoppelt ist und umgekehrt.

M8RLN
Normales Rechtsverfahren (8x8)

```
1           ORG  $0300
2   * M8RLN
3   * ====
4           JMP  S1
5   * Multiplikand Low
6   MDL     HEX  00
7   * Multiplikator Low
8   MRL     HEX  00
9   * Produkt Low-High
10  PL      HEX  00
11  PH      HEX  00
12  * Shift-MD Low-High
13  SMDL    HEX  00
14  SMDH    HEX  00
15  * Shift-MR Low
16  SMRL    HEX  00
17  * Bit-Zähler
18  BITS    HEX  00
19  * 1. Werte initialisieren
20  S1      LDA  #0
21          STA  PL
22          STA  PH
23          LDA  MDL
24          STA  SMDL
25          LDA  #0
26          STA  SMDH
27          LDA  MRL
28          STA  SMRL
29  * 2. Bit-Zähler setzen
30  S2      LDA  #8
31          STA  BITS
32  * 3. SMR rechtsverschieben
33  S3      LSR  SMRL
34          BCC  S5
35  * 4. Wenn C=1, P = P + SMD
36  S4      CLC
37          LDA  PL
38          ADC  SMDL
39          STA  PL
40          LDA  PH
41          ADC  SMDH
42          STA  PH
43  * 5. SMD linksverschieben
44  S5      ASL  SMDL
45          ROL  SMDH
46  * 6. Alle Bits erledigt?
47  S6      DEC  BITS
48          BNE  S3
49          RTS
```

M8LRN
Normales Linksverfahren (8x8)

```
1           ORG  $0300
2   * M8LRN
3   * ====
4           JMP  S1
5   * Multiplikand Low
6   MDL     HEX  00
7   * Multiplikator Low
8   MRL     HEX  00
9   * Produkt Low-High
10  PL      HEX  00
11  PH      HEX  00
12  * Shift-MD Low-High
13  SMDL    HEX  00
14  SMDH    HEX  00
15  * Shift-MR Low
16  SMRL    HEX  00
17  * Bit-Zähler
18  BITS    HEX  00
19  * 1. Werte initialisieren
20  S1      LDA  #0
21          STA  PL
22          STA  PH
23          LDA  MDL
24          STA  SMDH
25          LDA  #0
26          STA  SMDL
27          LDA  MRL
28          STA  SMRL
29  * 2. Bit-Zähler setzen
30  S2      LDA  #8
31          STA  BITS
32  * 3. SMD rechtsverschieben
33  S3      LSR  SMDH
34          ROR  SMDL
35  * 4. SMR linksverschieben
36  S4      ASL  SMRL
37          BCC  S6
38  * 5. Wenn C=1, P = P + SMD
39  S5      CLC
40          LDA  PL
41          ADC  SMDL
42          STA  PL
43          LDA  PH
44          ADC  SMDH
45          STA  PH
46  * 6. Alle Bits erledigt?
47  S6      DEC  BITS
48          BNE  S3
49          RTS
```

M8RLT
Anomales Rechtsverfahren (8x8)

```
1           ORG  $0300
2    * M8RLT
3    * ======
4           JMP  S1
5    * Multiplikand Low
6    MDL    HEX  00
7    * Multiplikator Low
8    MRL    HEX  00
9    * Produkt Low-High
10   PL     HEX  00
11   PH     HEX  00
12   * Shift-MR Low
13   SMRL   HEX  00
14   * Bit-Zähler
15   BITS   HEX  00
16   * 1. Werte initialisieren
17   S1     LDA  #0
18          STA  PL
19          STA  PH
20          LDA  MRL
21          STA  SMRL
22   * 2. Bit-Zähler setzen
23   S2     LDA  #8
24          STA  BITS
25   * 3. SMR rechtsverschieben
26   S3     LSR  SMRL
27          BCC  S5
28   * 4. Wenn C=1, PH = PH + MDL
29   S4     CLC
30          LDA  PH
31          ADC  MDL
32          STA  PH
33   * 5. P rechtsverschieben
34   S5     ROR  PH
35          ROR  PL
36   * 6. Alle Bits erledigt?
37   S6     DEC  BITS
38          BNE  S3
39          RTS
```

M8LRT
Anomales Linksverfahren (8x8)

```
1           ORG  $0300
2    * M8LRT
3    * ======
4           JMP  S1
5    * Multiplikand Low
6    MDL    HEX  00
7    * Multiplikator Low
8    MRL    HEX  00
9    * Produkt Low-High
10   PL     HEX  00
11   PH     HEX  00
12   * Shift-MR Low
13   SMRL   HEX  00
14   * Bit-Zähler
15   BITS   HEX  00
16   * 1. Werte initialisieren
17   S1     LDA  #0
18          STA  PL
19          STA  PH
20          LDA  MRL
21          STA  SMRL
22   * 2. Bit-Zähler setzen
23   S2     LDA  #8
24          STA  BITS
25   * 3. P linksverschieben
26   S3     ASL  PL
27          ROL  PH
28   * 4. SMR linksverschieben
29   S4     ASL  SMRL
30          BCC  S6
31   * 5. Wenn C=1, P = P + MDL
32   S5     CLC
33          LDA  PL
34          ADC  MDL
35          STA  PL
36          BCC  S6
37          INC  PH
38   * 6. Alle Bits erledigt?
39   S6     DEC  BITS
40          BNE  S3
41          RTS
```

Nachfolgend bringen wir noch für die 16-mal-16-Bit-Multiplikation Algorithmen für das normale und das anomale Rechtsverfahren. Die entsprechenden Linksverfahren befinden sich auf der Peeker-Sammeldisk.

M16RLN
Normales Rechtsverfahren (16x16)

```
1           ORG  $0300
2    * M16RLN
3    * ======
4           JMP  S1
5    * Multiplikand High/Low
6    MDH    HEX  00
7    MDL    HEX  00
8    * Multiplikator High/Low
9    MRH    HEX  00
10   MRL    HEX  00
11   * Produkt High bis Low
12   PH     HEX  00
13   PN     HEX  00
14   PM     HEX  00
15   PL     HEX  00
16   * Shift-Multiplikand
17   SMDH   EQU  $00F9
18   SMDN   EQU  $00FA
19   SMDM   EQU  $00FB
20   SMDL   EQU  $00FC
21   * Shift-Multiplikator
22   SMRH   EQU  $00FD
23   SMRL   EQU  $00FE
24   * Bit-Zähler
25   BITS   EQU  $00FF
26   * 1. Werte initialisieren
27   S1     LDY  #3
28          LDA  #0
29   LOESCHE STA  PH,Y
30          STA  SMDH,Y
31          DEY
32          BPL  LOESCHE
33          LDY  #1
34   KOPIERE LDA  MRH,Y
35          STA  SMRH,Y
36   * MD.HL nach SMD.ML
37          LDA  MDH,Y
38          STA  SMDM,Y
39          DEY
40          BPL  KOPIERE
41   * 2. Bit-Zähler initialisieren
42   S2     LDA  #16
43          STA  BITS
44   * 3. SMR rechtsverschieben
45   S3     LSR  SMRH
46          ROR  SMRL
47          BCC  S5
48   * 4. Wenn C=1, P = P + SMD
49   S4     CLC
50          LDY  #3
51   ADDIERE LDA  PH,Y
52          ADC  SMDH,Y
53          STA  PH,Y
54          DEY
55          BPL  ADDIERE
56   * 5. SMD linksverschieben
57   S5     ASL  SMDL
58          ROL  SMDM
59          ROL  SMDN
60          ROL  SMDH
61   * 6. Alle Bits erledigt?
62   S6     DEC  BITS
63          BNE  S3
64          RTS
```

M16RLT
Anomales Rechtsverfahren (16x16)

```
1           ORG  $0300
2    * M16RLT
3    * ======
4           JMP  S1
5    * Multiplikand High/Low
6    MDH    HEX  00
7    MDL    HEX  00
8    * Multiplikator High/Low
9    MRH    HEX  00
10   MRL    HEX  00
11   * Produkt High bis Low
12   PH     HEX  00
13   PN     HEX  00
14   PM     HEX  00
15   PL     HEX  00
16   * Shift-Multiplikator
17   SMRH   EQU  $00FD
18   SMRL   EQU  $00FE
19   * Bit-Zähler
20   BITS   EQU  $00FF
21   * 1. Werte initialisieren
22   S1     LDY  #3
23          LDA  #0
24   LOESCHE STA  PH,Y
25          DEY
26          BPL  LOESCHE
27          LDY  #1
28   KOPIERE LDA  MRH,Y
29          STA  SMRH,Y
30          DEY
31          BPL  KOPIERE
32   * 2. Bit-Zähler initialisieren
33   S2     LDA  #16
34          STA  BITS
35   * 3. SMR rechtsverschieben
36   S3     LSR  SMRH
37          ROR  SMRL
38          BCC  S5
39   * 4. Wenn C=1, P.HN = P.HN + MD.HL
40   S4     CLC
41          LDY  #1
42   ADDIERE LDA  PH,Y
43          ADC  MDH,Y
44          STA  PH,Y
45          DEY
46          BPL  ADDIERE
47   * 5. P rechtsverschieben
48   S5     ROR  PH
49          ROR  PN
50          ROR  PM
51          ROR  PL
52   * 6. Alle Bits erledigt?
53   S6     DEC  BITS
54          BNE  S3
55          RTS
```

Geschwindigkeitsoptimierte 8-mal-8-, 16-mal-8- und 16-mal-16-Bit-Multiplikationsroutinen – allesamt Tausendeins-Rechtsverfahren – finden Sie in dem Aufsatz „Fast 6502 Multiplication" in „Call A.P.P.L.E.", Heft 6/1983. Eine Variante der schnellen 16-mal-16-Bit-Multiplikation ist außerdem in meinem „Apple Assembler", S. 65, abgedruckt.
Die Peeker-Sammeldisk enthält zudem die Programme MULT8RLN.DEMO und MULT16RLN.DEMO, die den Multiplikationsvorgang Schritt für Schritt in Form von Binärzahlen anzeigen und sich deshalb besonders zum Üben eignen.

6. Division

Die Division ist leichter verständlich als die Multiplikation, da exotische Algorithmen in der Art des Tausendeinsverfahrens nicht möglich sind.

6.1. Dezimale Division

Dividieren heißt teilen. Bei dem Ausdruck
22 : 7 = 3 Rest 1
ist die 21 der Dividend DD (= die zu teilende Zahl), die 7 der Divisor DR (= Teiler), die 3 der Quotient Q (= das ganz-

zahlige Ergebnis) und die 1 der Rest R, der bei natürlichen Zahlen dann entsteht, wenn die Division nicht „aufgeht".
Da die Division als Umkehroperation zur Multiplikation verstanden wird, kann man sich von der Richtigkeit einer Division überzeugen, indem man den errechneten Quotienten mit dem Divisor multipliziert und den eventuellen Rest hinzuzählt, also $(7 * 3) + 1 = 22$.
Wie bei der Subtraktion können auch bei der Division die Operanden nicht vertauscht werden, denn beispielsweise führt
7 : 2
zu einem anderen Ergebnis als
2 : 7.
Ferner wird in der traditionellen Mathematik die Division durch 0 ausgeschlossen. Es gelten dann – im Hinblick auf die binäre Division – folgende Divisionsregeln:

```
DD : DR = Q Rest R
1  : 1  = 1 Rest 0  (Q = DD)
0  : 1  = 0 Rest 0  (R = DD)
1  : 0  = verboten
0  : 0  = verboten
```

Die Division kann man in eine reine Subtraktion überführen, z.B.
369 : 123 = 3 Rest 0:

```
 369
-123 1mal
-123 2mal
-123 3mal
 ---
 000
```

Wenn Dividend und Divisor die gleiche Anzahl der Stellen haben, ist dieses Verfahren empfehlenswert. Für die Aufgabe
98765432 : 2
würde man jedoch bereits einen Karton Papier benötigen.

Intuitives „Herunterholen"

Bei der schriftlichen Division wendet man üblicherweise das „Herunterhol"-Verfahren an:

```
 3150 : 25 = 126
-25''
 --
  65' (5 herunter)
 -50
  --
  150' (0 herunter)
 -150
  ---
    0
```

Typisch für dieses Verfahren ist,
a) daß der Divisor zunächst linksbündig unter den Dividenden gesetzt wird und
b) daß im Falle einer nicht erfolgreichen Subtraktion die nächste Stelle *oder auch mehrere gleichzeitig* „heruntergeholt" werden. Dies setzt jedoch voraus, daß man entweder ein guter Kopfrechner ist oder gut „raten" kann.
Wenn man als schlechter Kopfrechner die 24-durch-8-stellige Division
997869868785696868697845 : 12367578
nach diesem intuitiven Verfahren zu lösen versucht, begreift man die Unzulänglichkeit des Algorithmus.

Partialdivisionsverfahren

Das „Herunterholen" läßt sich durch die Stellen- oder Partialdivision beweisen. Das Verfahren ist jedoch derart umständlich, daß es als maschinensprachlicher Algorithmus ausscheidet:

```
  (3000 + 0100 + 0050 + 0000)     (3150)
: (0020 + 0005)                 : (  25)
= (0100 + 0020 + 0006)          = ( 126)
=========================================
  (3000 + 0100)  [0100 heruntergeholt]
- (0400 + 0100)  = (0020 + 0005) * 0020
= (2600 + 0000)
-----------------------------------------
  (2600 + 0050)  [0050 heruntergeholt]
- (0200 + 0050)  = (0020 + 0005) * 0010
= (2400 + 0000)
-----------------------------------------
  (2000 + 0400)  [2400 neu zerlegt]
- (1600 + 0400)  = (0020 + 0005) * 0080
= (0400 + 0000)
-----------------------------------------
  (0300 + 0100)  [0400 neu zerlegt]
- (0300 + 0075)  = (0020 + 0005) * 0015
= (0000 + 0025)
-----------------------------------------
  (0020 + 0005)  [0025 neu zerlegt]
- (0020 + 0005)  = (0020 + 0005) * 0001
-----------------------------------------
                                = 0126
=========================================
```

Genormtes „Herunterholen"

Wenn Dividend und Divisor beide x-stellig sind, genügen jeweils x-stellige Zahlenfelder für den Quotienten und den Rest. Beispiele:
999 : 001 ergibt Q = 999, R = 000.
998 : 999 ergibt Q = 000, R = 998.
Wenn der Dividend (2 * x)-stellig und der Divisor x-stellig sind, genügen ein (2 * x)-stelliges Feld für den Quotienten und ein x-stelliges Feld für den Rest. Beispiele:
9999 : 01 ergibt Q = 9999, R = 00.
0098 : 99 ergibt Q = 0000, R = 98.
Im folgenden beschränken wir uns auf Rechenbeispiele mit 4stelligem Dividenden und Divisor und somit 4stelligem Quotienten und Rest. Bei unserem Einführungsbeispiel müssen wir mithin mit Nullen auffüllen:
3150 : 0025.

Wenn wir vom Kopfrechnen überhaupt keine Ahnung hätten, würden wir den „Herunterhol"-Algorithmus unter Berücksichtigung genormter Zahlenfelder etwa so implementieren:

```
   DD     DR     Q
  3150 : 0025 = xxxx
-------------------
0000''''      = xxxx
 0003' (3)
-0025
 ----
 0003         = 0xxx
  0031' (1)
 -0025
 ----
 0006         = 01xx
  0065' (5)
 -0025
 -0025
 ----
 0015         = 012x
  0150' (0)
 -0025
 -0025
 -0025
 -0025
 -0025
 -0025
 ----
 0000         = 0126
```

Bei diesem Algorithmus gibt es zwei Besonderheiten:
a) Der Dividend hat eine feste Position, und der Divisor wurde von links nach rechts *unter* dem Dividenden verschoben. Alternative: Der Divisor erhält eine feste Position, und der Dividend wird von rechts nach links *über* dem Divisor verschoben, was auf dasselbe hinausläuft. Der Dividend fungiert dann programmtechnisch als „Shift-Dividend" (Linksverschiebung).
b) Das Einfügen des jeweils errechneten Stellenergebnisses im Quotientenfeld ist algorithmisch* ungeschickt. Alternative: Die Stellenergebnisse werden von rechts nach links in das Quotientenfeld hineingeschoben, was auf dasselbe hinausläuft. Der Quotient fungiert dann programmtechnisch als „Shift-Quotient" (Linksverschiebung).

Genormtes Schiebeverfahren

Unter Berücksichtigung der obigen Modifikationen gelangen wir nunmehr zu einem genormten Schiebeverfahren:

```
  1.Sp  2.Sp  3.Sp
-------------------------
  R-DR  DD    Q
  0000  3150  xxxx   0.
-------------------------
< 0003  150x
- 0025        xxx0   1.
= 0003
-------------------------
< 0031  50xx         2.
- 0025        xx01
= 0006
-------------------------
< 0065  5xxx         3.
- 0025        x011
- 0025        x012
= 0015
-------------------------
< 0150  xxxx         4.
- 0025        0121
- 0025        0122
- 0025        0123
- 0025        0124
- 0025        0125
- 0025        0126
= 0000  Rest
```

Spalte 1 enthält den jeweiligen Rest (R) und den Divisor (DR), Spalte 2. den Shift-

* Algorithmisch ungeschickt in bezug auf die 6502-Schiebebefehle, die keine Bit-Einfügung zulassen. Wenn wir mit ROL das Carry-Flag in den Quotienten hineinschieben, führen wir quasi durch die Hintertür das 1001-Verfahren bei der Division ein.

Dividenden (DD) und Spalte 3. den Shift-Quotienten (Q).

0. Block: Wir initialisieren R auf 0000, tragen DD ein und löschen Q. Damit die Schiebetechnik besser auffällt, schreiben wir x statt 0.

1. Block: Wir schieben DD um eine Stelle nach links in R hinein [<–]. Damit kommt die 3 als höchste Stelle von 3150 in die niedrigste Stelle von R. Nunmehr subtrahieren wir
0003 - 0025 = 0 R 0003,
schieben das *Stellen*ergebnis 0 (daher nur 0 und nicht 0000) von rechts in Q hinein und übernehmen 0003 als R in die nächste Zeile.

Block 2: Wir schieben wieder DD um eine Stelle nach links in R hinein. Nunmehr subtrahieren wir
0031 - 0025 = 1 R 0006,
schieben das Stellenergebnis 1 von rechts in Q hinein und übernehmen 0006 als R in die nächste Zeile.

Block 3: Wir schieben wieder DD um eine Stelle nach links in R hinein. Nunmehr subtrahieren wir
0065 - 0025 - 0025 = 2 R 0015.
Wie ersichtlich, muß die 0025 hier zweimal subtrahiert werden. Bei Dezimalzahlen muß 0-9mal, bei Binärzahlen nur 0-1mal subtrahiert werden. Auch wenn wir mehrfach subtrahieren müssen, ist das Ergebnis trotzdem stets einstellig. Deshalb schieben wir auch hier das Stellenergebnis 2 von rechts in Q hinein und übernehmen 0015 als R in die nächste Zeile.

Block 4: Auch hier ist die Prozedur wieder dieselbe, wobei allerdings nunmehr das Stellenergebnis 6 beträgt. R ist nachher 0000, und damit geht die Division insgesamt ohne Rest auf.
Selbst wenn wir, z.B. bei der Aufgabe
3151 : 0025 = 0126 R 0001,
einen Rest vorgefunden hätten, wäre die Division jetzt beendet, weil der Dividend bereits 4mal nach links geschoben worden ist.

Algorithmus in Kurzform
Schritt 1: Rest und Quotienten auf 0 setzen.
Schritt 2: Dividenden von rechts nach links um eine Stelle in den Rest hineinschieben.
Schritt 3: Subtraktion von Rest minus Divisor durchführen. Stellenergebnis der Subtraktion von rechts nach links in den Quotienten hineinschieben sowie neuen Rest eintragen.
Schritt 4: Prüfen, ob bereits sooft linksverschoben worden ist, wie der Dividend Stellen hat. Wenn nein, dann zurück zu Schritt 2. Wenn ja, dann Ende.

6.2. Binäre Division

Die binäre Division läßt sich haargenau wie die genormte dezimale Division durchführen. Es gibt nur eine kleine und zugleich erfreuliche Besonderheit: Das Stellenergebnis ist höchstens 1, und deshalb muß der Divisor vom jeweiligen Rest auch höchstens ein einziges Mal subtrahiert werden. Eine x-durch-x-Bit-Division besteht damit aus exakt

x Linksverschiebungen von DD und R
x Subtraktionen von R und DR
x Linksverschiebungen von Q,
wobei gleichzeitig entweder 0 oder 1 als Stellenergebnis von rechts in Q hineingeschoben wird.

Könnte das Stellenergebnis nicht doch einmal größer als 1 sein? Betrachten wir hierzu die nachstehende Division:

```
  11 :    7 =    1 R    4
1011 : 0111 = 0001 R 0100
-------------------------
  R-DR  DD    Q
  0000  1011  xxxx   0.
-------------------------
< 0001  011x        1.
- 0111       xxx0
= 0001
-------------------------
< 0010  11xx        2.
- 0111       xx00
= 0010
-------------------------
< 0101  1xxx        3.
- 0111
= 0101       x000
-------------------------
< 1011  xxxx        4.
- 0111       0001
= 0100  Rest
```

Wir haben also 4mal schieben müssen, bevor eine Subtraktion, nämlich die letzte, „aufging". Wenn Sie ähnliche Beispiele ausprobieren, werden Sie feststellen:
Die höchste Einserbit-Stelle des momentanen Rests kann die höchste Einserbit-Stelle des Divisors höchstens um eine einzige Stelle überragen, weil die Division durch 0 verboten ist und somit der Divisor stets größer als 1 ist. Wenn dann der Divisor vom momentanen Rest abgezogen werden kann, wird die besagte Einserbit-Stelle beim neuen Rest eliminiert. Wenn umgekehrt der Divisor nicht vom momentanen Rest abgezogen werden kann, dann muß es sich um den Rest nach der letzten Linksverschiebung handeln. Damit ist jedoch gleichzeitig die Division beendet.

Nachfolgend bringen wir ein komplizierteres Beispiel
dezimal 65535 : 17 = 3855 Rest 0
für eine Binärdivision, wobei wir die Zahlenkolonnen nach der 6502-Division in Abschnitt 6.3 anordnen. Es bedeuten
R = Rest
S = Shift-Dividend
D = Divisor
Q = Quotient
↑ = Es wurde 1mal subtrahiert.
Wenn der Rest kleiner als der Divisor ist, wird die (versuchte) Subtraktion nicht angezeigt und der Rest sofort weiterverschoben.

```
R 00000000 00000000 S 11111111 11111111
---------------------------------------
R 00000000 00000001 S 11111111 11111110
D 00000000 00010001 Q xxxxxxxx xxxxxxx0
R 00000000 00000011 S 11111111 11111100
D 00000000 00010001 Q xxxxxxxx xxxxxx00
R 00000000 00000111 S 11111111 11111000
D 00000000 00010001 Q xxxxxxxx xxxxx000
R 00000000 00001111 S 11111111 11110000
D 00000000 00010001 Q xxxxxxxx xxxx0000
R 00000000 00011111 S 11111111 11100000
D 00000000 00010001 Q xxxxxxxx xxx00001
= 00000000 00001110                   ↑
R 00000000 00011101 S 11111111 11000000
D 00000000 00010001 Q xxxxxxxx xx000011
= 00000000 00001100                   ↑
R 00000000 00011001 S 11111111 10000000
D 00000000 00010001 Q xxxxxxxx x0000111
= 00000000 00001000                   ↑
R 00000000 00010001 S 11111111 00000000
D 00000000 00010001 Q xxxxxxxx 00001111
= 00000000 00000000                   ↑
R 00000000 00000001 S 11111110 00000000
D 00000000 00010001 Q xxxxxxx0 00011110
R 00000000 00000011 S 11111100 00000000
D 00000000 00010001 Q xxxxxx00 00111100
R 00000000 00000111 S 11111000 00000000
D 00000000 00010001 Q xxxxx000 01111000
R 00000000 00001111 S 11110000 00000000
D 00000000 00010001 Q xxxx0000 11110000
R 00000000 00011111 S 11100000 00000000
D 00000000 00010001 Q xxx00001 11100001
= 00000000 00001110                   ↑
R 00000000 00011101 S 11000000 00000000
D 00000000 00010001 Q xx000011 11000011
= 00000000 00001100                   ↑
R 00000000 00011001 S 10000000 00000000
D 00000000 00010001 Q x0000111 10000111
= 00000000 00001000                   ↑
R 00000000 00010001 S 00000000 00000000
D 00000000 00010001 Q 00001111 00001111
= 00000000 00000000                   ↑
```

6.3. 6502-Division

Bei der nachfolgenden Version 1 einer 16-durch-16-Bit-Division (D16V1) lassen sich folgende Schritte hervorheben:

Schritt 1: Quotient und Rest werden auf 0 gesetzt und der Dividend wird in einen Shift-Dividenden dupliziert, damit der ursprünglich Dividend erhalten bleibt.
Schritt 2: Vorab wird geprüft, ob der Divisor 0 ist. Wenn ja, wird die Routine sofort verlassen.
Schritt 3: Da der Dividend 16 Bits umfaßt, wird der Bit-Zähler auf 16 gesetzt. Der Bit-Zähler richtet sich nach dem Dividenden und wäre deshalb bei einer 16-durch-8-Bit-Division ebenfalls 16.
Schritt 4: Der Shift-Dividend wird von rechts nach links in den Rest hineingeschoben.
Schritt 5: Dann wird geprüft, ob die Subtraktion von Rest minus Divisor ohne Überlauf möglich ist. Im Falle eines Überlaufs ist C = 0, sonst C = 1.
Schritt 6: Wenn ja, wird die Differenz von Rest und Divisor als neuer Rest gespeichert.

Schritt 7: Das Ergebnis der erfolgreichen (C = 1) oder erfolglosen (C = 0) Subtraktion wird von rechts in den Quotienten hineingeschoben.
Schritt 8: Der Bit-Zähler wird dekrementiert und auf 0 überprüft. Wenn noch nicht alle 16 Bits abgearbeitet worden sind, kehrt die Programmschleife zu Schritt 4 zurück.

D16 V1

```
1             ORG  $0300
2   *
3   * D16 Version 1
4   * ===
5             JMP  S1
6   * Dividend
7   DDL       HEX  FF
8   DDH       HEX  FF
9   * Divisor
10  DRL       HEX  0F
11  DRH       HEX  00
12  * Quotient
13  QL        HEX  00
14  QH        HEX  00
15  * Rest
16  RL        HEX  00
17  RH        HEX  00
18  * Shift-Dividend
19  SDDL      HEX  00
20  SDDH      HEX  00
21  * Bit-Zähler
22  BITS      HEX  00
23  * 1. Q = 0, R = 0, SDD = DD
24  S1        LDX  #1
25  S1A       LDA  #0
26            STA  QL,X
27            STA  RL,X
28            LDA  DDL,X
29            STA  SDDL,X
30            DEX
31            BPL  S1A
32  * 2. DR = 0?
33  S2        LDA  DRL
34            ORA  DRH
35            BEQ  S9
36  * 3. Bit-Zähler auf 16
37  S3        LDA  #16
38            STA  BITS
39  * 4.  <-- SDDH-SDDL-0 links
40  S4        ASL  SDDL
41            ROL  SDDH
42  * 4A. <-- RH-RL-C links
43  S4A       ROL  RL
44            ROL  RH
45  * 5. R - DR >= 0?
46  S5        SEC
47            LDA  RL
48            SBC  DRL
49            TAX
50            LDA  RH
51            SBC  DRH
52            BCC  S7          ;C=0!
53  * 6. Wenn ja, R = R - DR
54  S6        STX  RL          ;C=1!
55            STA  RH
56  * 7. und <-- QH-QL-C links
57  S7        ROL  QL
58            ROL  QH
59  * 8. Bit-Zähler = 0?
60  S8        DEC  BITS
61            BNE  S4
62  * 9. Ende (auch bei DR=0)
63  S9        RTS
```

Die analoge Version 2 (D16V2) überträgt den Dividenden nicht in einen gesonderten Shift-Dividenden, sondern in den Quotienten. Dies ist möglich, weil mit jeder Linksverschiebung des „Quotienten" = Shift-Dividenden das rechte, nullte Bit frei wird und somit das Carry-Flag der erfolgreichen/erfolglosen Subtraktion aufnehmen kann. Wegen des INC-Befehls in Zeile 53 ist Version 2 geringfügig langsamer. Die Peeker-Sammeldisk enthält außerdem die „getunten" Versionen 3 (D16V3) und 4 (D16V4), von denen Version 4 ca. 25% schneller ist. Das zusätzliche Programm D24V2 ist eine 24-durch-24-Bit-Division nach dem Algorithmus von Version 2. Ferner sind auf die Sammeldisk die Demos DIV16.DEMO und DIV24.DEMO aufgenommen worden, die den internen Rechenprozeß Schritt für Schritt in Form von Binärzahlen anzeigen und sich deshalb besonders für die Schule eignen.

D16 V2

```
1             ORG  $0300
2   *
3   * D16 Version 2
4   * ===
5             JMP  S1
6   * Dividend
7   DDL       HEX  FF
8   DDH       HEX  FF
9   * Divisor
10  DRL       HEX  0F
11  DRH       HEX  00
12  * Quotient
13  QL        HEX  00
14  QH        HEX  00
15  * Rest
16  RL        HEX  00
17  RH        HEX  00
18  * Bit-Zähler
19  BITS      HEX  00
20  * 1. R = 0, Q = DD
21  S1        LDX  #1
22  S1A       LDA  #0
23            STA  RL,X
24            LDA  DDL,X
25            STA  QL,X
26            DEX
27            BPL  S1A
28  * 2. DR = 0?
29  S2        LDA  DRL
30            ORA  DRH
31            BEQ  S9
32  * 3. Bit-Zähler auf 16
33  S3        LDA  #16
34            STA  BITS
35  * 4.  <-- QH-QL-0 links
36  S4        ASL  QL
37            ROL  QH
38  * 4A. <-- RH-RL-C links
39  S4A       ROL  RL
40            ROL  RH
41  * 5. R - DR >= 0?
42  S5        SEC
43            LDA  RL
44            SBC  DRL
45            TAX
46            LDA  RH
47            SBC  DRH
48            BCC  S8
49  * 6. Wenn ja, R = R - DR
50  S6        STX  RL
51            STA  RH
52  * 7. und QL = QL + 1
53  S7        INC  QL
54  * 8. Bit-Zähler = 0?
55  S8        DEC  BITS
56            BNE  S4
57  * 9. Ende (auch bei DR=0)
58  S9        RTS
```

Peeker-Sammeldisk

Auf der Sammeldisk #14 befinden sich diverse Assemblerprogramme (mit Big-Mac-Quelltexten) und Demos zu diesem Beitrag, wobei die mit RUN gekennzeichneten Applesoft-Demos die jeweils nachfolgenden Maschinenprogramme starten:

Multiplikationsprogramme
M8TEST (RUN)
T.M8RLN und M8RLN
T.M8LRN und M8LRN
T.M8RLT und M8RLT
T.M8LRT und M8LRT
M16TEST (RUN)
T.M16RLN und M16RLN
T.M16LRN und M16LRN
T.M16RLT und M16RLT
T.M16LRT und M16LRT
MULT8RLN.DEMO (RUN)
T.MULT8RLN und MULT8RLN
MULT16RLN.DEMO (RUN)
T.MULT16RLN und MULT16RLN
Divisionsprogramme
D16TEST (RUN)
T.D16V1 und D16V1
T.D16V2 und D16V2
T.D16V3 und D16V3
T.D16V4 und D16V4
D24TEST (RUN)
T.D24V2 und D24V2
DIV16.DEMO (RUN)
T.DIV16 und DIV16
DIV24.DEMO (RUN)
T.DIV24 und DIV24