Rechneraufbau und Rechnerstrukturen

Von
Walter Oberschelp,
Gottfried Vossen

10., überarbeitete und erweiterte Auflage

Oldenbourg Verlag München Wien

Prof. Dr. Gottfried Vossen lehrt seit 1993 Informatik am Institut für Wirtschaftsinformatik der Universität Münster. Er studierte, promovierte und habilitierte sich an der RWTH Aachen und war bzw. ist Gastprofessor u.a. an der University of California in San Diego, USA, an der Karlstad Universitet in Schweden, an der University of Waikato in Hamilton, Neuseeland sowie am Hasso-Plattner-Institut für Softwaresystemtechnik in Potsdam. Er ist europäischer Herausgeber der bei Elsevier erscheinenden Fachzeitschrift Information Systems.

Prof. Dr. Walter Oberschelp studierte Mathematik, Physik, Astronomie, Philosophie und Mathematische Logik. Nach seiner Habilitation in Hannover lehrte er als Visiting Associate Professor an der University of Illinois (USA). Nach seiner Rückkehr aus den USA übernahm er den Lehrstuhl für Angewandte Mathematik an der RWTH Aachen, den er bis zu seiner Emeritierung im Jahr 1998 inne hatte.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-wissenschaftsverlag.de

Lektorat: Margit Roth
Herstellung: Anna Grosser
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: Oldenbourg Druckerei Vertriebs GmbH & Co. KG
Bindung: R. Oldenbourg Graphische Betriebe Binderei GmbH

ISBN 3-486-57849-9
ISBN 978-3-486-57849-2

Kapitel 6

Darstellung von Daten im Rechner. Rechnerarithmetik

Wir wollen uns in diesem Kapitel mit verschiedenen grundlegenden Aspekten der Informationsdarstellung und -verarbeitung (insbesondere im Von-Neumann-Rechner) beschäftigen, welche bisher nur knapp oder nicht behandelt wurden. Dabei werden wir einerseits beschreiben, wie z. B. negative Zahlen oder Text im Rechner darstellbar sind, andererseits aber auch Fragen der Rechnerarithmetik behandeln. Dies wird sich insbesondere beziehen auf das Rechnen mit Fest- oder Gleitkomma-Zahlen, wobei wir speziell die Multiplikation näher untersuchen werden.

6.1 Darstellung ganzer Zahlen. Subtraktion

Die bisherigen Ausführungen haben verdeutlicht, dass die Wortlänge eines Rechners (z. B. 8, 16 oder 32 Bits) eine obere Grenze für die Größe von Zahlen darstellt, welche der Rechner verarbeiten kann. So lassen sich z. B. mit 8 Bits alle natürlichen Zahlen zwischen 0 und $2^8 - 1 = 255$ darstellen. In gewissem Umfang sind darüber hinaus durch die Verwendung einzelner Flags auch Überschreitungen dieses Bereichs, z. B. bei der Addition 156 + 184 in einem Rechner der Wortlänge 8 Bits, möglich. Die Ausführung einer Subtraktion mit negativem Resultat wirft aber bereits die Frage auf, wie am Ende einer solchen Operation der Akku-Inhalt zu interpretieren ist. Daher wollen wir zunächst die Frage der Darstellung von Zahlen mit Vorzeichen klären und hier vier verschiedene Alternativen vorstellen.

Eine erste, nahe liegende Möglichkeit hierzu ist die so genannte *Vorzeichen/Betrags-Darstellung* (engl. Sign/Magnitude). Dabei wird ein Bit eines Registers (bzw. allgemeiner einer Speicherzelle) als Vorzeichen-Bit ausgezeichnet; die restlichen Bits dienen zur Darstellung des Betrages der betreffenden Zahl im Dualsystem wie bisher. Die übliche Konvention ist dabei, den Inhalt 0 des Vorzeichen-Bits, als welches das am weitesten links stehende Bit angenommen wird, als „+", den Inhalt 1 als „−" zu interpretieren.

Beispiel 6.1 (a) Bei Verwendung der kürzest möglichen Wortlänge wird die Zahl +5 als 0101, entsprechend die Zahl −5 als 1101 dargestellt.

(b) In einem Rechner der Wortlänge 16 Bits werden +92 und −92 wie folgt dargestellt:

$$\begin{aligned} +92 &: 0000000001011100 \\ -92 &: 1000000001011100 \end{aligned}$$

□

Der z. B. durch ein 8-Bit-Register darstellbare Zahlenbereich umfasst jetzt die Zahlen von −127 bis +127 (gegenüber vorher 0 bis 255). Allgemein können bei gegebener Wortlänge n die Zahlen von $-(2^{n-1}-1)$ bis $+(2^{n-1}-1)$ dargestellt werden.

Man beachte, dass es in dieser Darstellung die beiden Nullen +0 und −0 gibt (dargestellt etwa durch 0000 bzw. 1000 bei einer Wortlänge von 4 Bits). Wenngleich beide Darstellungen intuitiv als identisch angesehen werden können, ist die Gleichheit für einen Rechner, welcher Bit-Positionen einzeln vergleicht, schwierig festzustellen. Ein weiterer Nachteil dieser nahe liegenden Darstellungsform besteht darin, dass sie ein Addier- *und* ein Subtrahierwerk erfordert, während prinzipiell nur eins dieser beiden Werke erforderlich ist (vgl. Abschnitt 5.3). Ferner ist eine Logik erforderlich, welche entscheidet, ob eine Addition oder eine Subtraktion auszuführen ist. Der Grund für diesen hohen Aufwand liegt darin, dass folgende vier Fälle unterschieden werden müssen für zwei Operanden x und y:

Fall		Operanden	auszuführende Operation	
1		$+x, +y$	$x+y$	Addition
2		$-x, -y$	$-(x+y)$	Addition
3		$+x, -y$ mit $\|x\| \geq \|y\|$	$x-y$	Subtraktion
	bzw.	$-x, +y$ mit $\|y\| \geq \|x\|$	$y-x$	Subtraktion
4		$+x, -y$ mit $\|x\| < \|y\|$	$-(y-x)$	Subtraktion
	bzw.	$-x, +y$ mit $\|y\| < \|x\|$	$-(x-y)$	Subtraktion

Man kann stattdessen mit *einem* Addierwerk auskommen, wenn man die Subtraktion auf die Addition zurückführt. Dazu kann man zwei Arten von *Komplementdarstellungen* verwenden, welche wir als nächstes beschreiben (vgl. Aufgabe 2.2).

Definition 6.1 Sei $x = (x_{n-1}, \ldots, x_0)_2 \in B^n$ eine n-stellige Dualzahl.

(i) $K_1(x) := (1 \dotplus x_{n-1}, \ldots, 1 \dotplus x_0)_2$ heißt *Einer-Komplement* (engl. *One's Complement*) von x.

(ii) $K_2(x) := (1 \dotplus x_{n-1}, \ldots, 1 \dotplus x_0)_2 + 1 = K_1(x) + 1$ (modulo 2^n) heißt *Zweier-Komplement* (engl. *Two's Complement*) von x.

Das Einer-Komplement einer Zahl x erhält man also durch stellenweises Invertieren von x, das Zweier-Komplement durch Invertieren aller Bits und anschließende Addition einer Eins (modulo 2^n). Es sei bemerkt, dass das Zweier-Komplement heute am häufigsten zur rechnerinternen Darstellung ganzer Zahlen benutzt wird.

Beispiel 6.2 Sei $x = 10110010$. Dann gilt:

$$\begin{aligned} K_1(x) &= 01001101 \\ K_2(x) &= 01001110 \end{aligned}$$

□

Es sei angemerkt, dass man allgemein in jedem b-adischen Zahlensystem das $(b-1)$- bzw. b-Komplement einer gegebenen Zahl erklären kann, wenngleich für die Informatik der Fall $b = 2$ am wichtigsten ist. Wesentlich ist, dass eine Komplement-Darstellung stets auf eine beliebige, aber fest vorgegebene Stellenzahl bezogen wird. Falls ein Rechner n Bits in einem Register oder einer Speicherzelle ablegen kann, so sind wie wir wissen — $N = 2^n$ verschiedene „Bitmuster" darstellbar. Da eine Komplement-Darstellung speziell zur Darstellung negativer Zahlen verwendet wird, kann man generell von folgender Idee ausgehen:

- Eine *positive* Zahl x wird dargestellt durch

$$+x = x$$

- Eine *negative* Zahl $-x$ wird dargestellt durch

$$-x = N - x$$

Beispiel 6.3 (a) Sei $b = 2$ und $n = 4$. Dann gilt $N = 2^4 = 16$. Im *Zweier-Komplement* stimmt die Dualdarstellung von -5 mit der von $16 - 5 = 11$ überein. Dies ist in Übereinstimmung mit Definition 5.1 (ii), denn es gilt:

$$\begin{aligned} (5)_{10} &= (0101)_2 \\ K_2(5) &= K_1(5) + 1 = (1010)_2 + 1 = (1011)_2 = (11)_{10} \end{aligned}$$

(b) Sei nun $b = 10$ und $n = 2$, d. h. wir betrachten Dezimalzahlen zwischen 00 und 99, so gilt $N = 10^2 = 100$. Im *Zehner-Komplement* wird dann -23 wie folgt dargestellt:

$$-23 \stackrel{\wedge}{=} 100 - 23 = 77$$

□

Dieses Beispiel zeigt, dass eine Komplement-Darstellung mit Mehrdeutigkeiten behaftet ist, welche zu beseitigen sind, bevor man diese Darstellungsform in einem Rechner benutzen kann. Insbesondere stellt sich in Beispiel 6.3 (a) die Frage, ob „1011" die Zahl -5 oder $+11$ darstellt; in Beispiel 6.3 (b) kann „77" sowohl -23 als auch $+77$ bedeuten. Dieses Problem wird durch eine der Vorzeichen/Betrags-Darstellung entsprechende Festlegung behoben: Eine Dualzahl, welche mit einer 0 beginnt, wird als positive Zahl aufgefasst, entsprechend eine solche, die mit 1 beginnt, als negative. (Eine entsprechende Konvention für das Zehner-Komplement lautet z. B.: Mit 0, 1, 2, 3 oder 4 beginnende Zahlen gelten als positiv, alle anderen als negativ.) Aus dieser

Festlegung folgt etwa in Beispiel 6.3 (a), dass die Zahl +11 mit 4 Bits nicht dargestellt werden kann; es sind mindestens 5 Bits erforderlich, welche dann die Darstellung 01011 erlauben.

Bei beiden Komplementdarstellungen ist also zu beachten, dass in einem Rechner stets eine bestimmte Wortlänge fest liegt, auf welche sich das Komplementieren bezieht, und dass für arithmetische Operationen lediglich negative Operanden komplementiert dargestellt werden.

Beispiel 6.4 Für $n = 16$ Bits lauten die Darstellungen von +92 und −92 im Einer- bzw. Zweier-Komplement wie folgt:

Komplement	+92	−92
Einer	dual 0000000001011100 hexadezimal 005C	dual 1111111110100011 hexadezimal FFA3
Zweier	dual 0000000001011100 hexadezimal 005C	dual 1111111110100100 hexadezimal FFA4

□

Die *Subtraktion* zweier n-stelliger Dualzahlen x und y $(x \geq y)$ lässt sich nun wie folgt bewerkstelligen:

(I) Bei Benutzung des *Einer-Komplementes* stelle man die zu subtrahierende Zahl (y) durch $K_1(y)$ dar und addiere $K_1(y)$ zu x. Tritt dabei ein Übertrag an der höchstwertigen Stelle auf, so addiere man diesen zur niedrigsten Stelle. Zur Korrektheit dieses Verfahrens vergleiche man Aufgabe 2.2.

Beispiel 6.5 Sei $x = 179$ und $y = 109$. Aus der Aufgabe

$$\begin{array}{rr} x & 10110011 \\ -y & -01101101 \end{array}$$

wird

$$\begin{array}{rr} x & 10110011 \\ +K_1(y) & +\ 10010010 \end{array}$$

mit dem Zwischenresultat

$$101000101.$$

Der Übertrag wird zur niedrigsten Stelle addiert:

$$\begin{array}{r} 01000101 \\ +\ 1 \end{array}$$

Damit entsteht das Ergebnis

$$01000110\,,$$

d. h. dezimal +70. □

Beispiel 6.6 In diesem Beispiel unterstellen wir eine rechnerinterne Wortlänge von 16 Bits:

(a) Die Subtraktion $45 - 92 = 45 + (-92) = -47$ wird wie folgt im Einer-Komplement ausgeführt:

$$\begin{array}{r} 0000000000101101 \\ +\ 1111111110100011 \\ \hline 1111111111010000 \end{array}$$

Hier tritt kein Übertrag in der höchstwertigen Stelle auf, d. h. es wird ein Übertrag von 0 zur niedrigstwertigen Stelle addiert.

(b) Die Aufgabe $1637 - 101 = 1637 + (-101) = 1536$ wird wie folgt gelöst:

$$\begin{array}{r} 0000011001100101 \\ +\ 1111111110011010 \\ \hline (1)0000010111111111 \\ +\ 1 \\ \hline 0000011000000000 \end{array}$$

□

Es sei bemerkt, dass auch das Einer-Komplement Probleme bei der Darstellung von 0 bereitet; wie bei der Vorzeichen/Betrags-Darstellung existieren $+0$ und -0, denn es gilt bei z. B. 4 Bits

$$(+0)_{10} = (0000)_2 \text{ und } (-0)_{10} = (1111)_2 ,$$

was intuitiv widersprüchlich ist. Wir werden weiter unten sehen, dass dies im Zweier-Komplement anders ist.

(II) Zur Ausführung einer Subtraktion der Form $x - y$ im *Zweier-Komplement* ist lediglich $K_2(y)$ zu x zu addieren; ein eventuell auftretender Übertrag wird ignoriert. Die Begründung für dieses Vorgehen erfolgt analog zu Aufgabe 2.2.

Beispiel 6.7 Sei $x = 179$ und $y = 109$, d. h. $x - y = 70$:

x	10110011
$+K_2(y)$	10010011
	(1)01000110

□

Beispiel 6.8 Unterstellen wir wieder 16 Bits zur rechnerinternen Darstellung von Zahlen, so wird die Aufgabe $92 - 45 = 92 + (-45) = 47$ wie folgt gelöst:

92	0000000001011100
$+K_2(45)$	1111111111010011
	(1)0000000000101111

□

Betrachten wir als nächstes die Darstellung von 0 im Zweier-Komplement, so ergibt sich Folgendes (für eine Wortlänge von 8 Bits):

0	00000000
$K_1(0)$	11111111
Addition von 1	+ 1
	(1)00000000

Die Vernachlässigung des Übertrags impliziert also jetzt, dass $+0 = -0$ gilt. Demnach hat die Null im Zweierkomplement nur eine Darstellung.

Die geschilderten Techniken sind auf die Lösung beliebiger Additions- bzw. Subtraktionsaufgaben unmittelbar übertragbar. Um dann z. B. die Subtraktion zweier Zahlen bei negativem Ergebnis allein durch ein Addierwerk ausführen zu können, werden negative Zahlen z. B. durch das Einerkomplement dargestellt. Zeigt dann nach Ausführung der Addition das Vorzeichenbit an, dass das Ergebnis negativ ist, so ist zur korrekten Interpretation des Ergebnisses erneut zu komplementieren.

Beispiel 6.9 Wir betrachten durch 8 Bits darstellbare Zahlen zwischen -127 und $+127$ und erläutern die Berechnung von $85 - 103$ (bei Verwendung von K_1):

$$\begin{array}{rcl} 85 & : & 01010101 \\ 103 & : & 01100111 \\ -103 & : & 10011000 \\ 85 + (-103) & : & 11101101 \end{array}$$

Die am weitesten links stehende Eins zeigt nun an, dass das Ergebnis 11101101 als negative Zahl aufzufassen ist; den Absolutbetrag erhält man also durch Komplementieren in (0)0010010 und anschließendes „Übersetzen" ins Dezimalsystem; man erhält damit die Zahl -18. □

Beispiel 6.10 Wir betrachten die Addition von -102 und -58 im Zweier-Komplement bei einer Wortlänge von 16 Bits; es gilt:

$$\begin{array}{rr} K_2(102): & 1111111110011010 \\ +K_2(58): & +\ 1111111111000110 \\ \hline & (1)1111111101100000 \end{array}$$

Das Resultat ist negativ und daher erneut zu komplementieren. Diese Rückübersetzung erfolgt jetzt in genau der gleichen Weise wie die Berechnung des Zweier-Komplements selbst: Alle Bits werden invertiert, so dass man in diesem Fall

$$0000000010011111$$

erhält. Die Addition von 1 liefert

$$0000000010100000$$

und somit dezimal -160. □

Wir wollen die bisher geschilderten Möglichkeiten zur Darstellung ganzer Zahlen zusammenfassen: Tabelle 6.1 gibt die durch Bitfolgen der Länge 4 darstellbaren Zahlen jeweils bei Verwendung von Vorzeichen/Betrags-Darstellung, Einer- bzw. Zweier-Komplement an. Diese Tabelle zeigt insbesondere, dass unter Verwendung des Zweier-Komplements bei gegebener Wortlänge von n Bits sogar alle Zahlen zwischen $-(2^{n-1})$ und $+(2^{n-1} - 1)$ darstellbar sind.

Es sei an dieser Stelle ferner auf das Problem des *Overflow* hingewiesen, welcher bei einer Addition auftreten kann: Falls bei der Addition von zwei positiven Zahlen ein (scheinbar) negatives Ergebnis entsteht bzw. bei der Addition von zwei negativen ein (scheinbar) positives, so liegt eine Bereichsüberschreitung vor:

Tabelle 6.1: Alternative Darstellungen ganzer Zahlen.

Bitfolge	Darstellung in Dezimalnotation		
	Vorz./Betrag	K_1	K_2
0000	+0	+0	0
0001	+1	+1	+1
0010	+2	+2	+2
0011	+3	+3	+3
0100	+4	+4	+4
0101	+5	+5	+5
0110	+6	+6	+6
0111	+7	+7	+7
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

Beispiel 6.11 Mit $n = 5$ Bits sind im Zweier-Komplement die Zahlen von -16 bis $+15$ darstellbar. Betrachten wir nun die Addition von 5 und 14, so erhält man folgendes Resultat:

$$\begin{array}{r} 00101 \\ +\ 01110 \\ \hline 10011 \end{array}$$

Das Ergebnis lautet also -13 und nicht $+19$. Der Grund liegt darin, dass $+19$ mit 5 Bits nicht mehr darstellbar ist. □

Hierzu sei bemerkt, dass Rechner, welche auf das Zweier-Komplement zur Ausführung arithmetischer Operationen zurückgreifen, in der Lage sind, derartige Situation zu erkennen. Falls ein Overflow auftritt, wird ein entsprechendes *Overflow-Flag* gesetzt, welches vom Programmierer abgefragt werden kann.

6.2 Darstellung von Gleitkomma-Zahlen

Bei den bisher verwendeten Zahlendarstellungen sind wir immer von ganzen Zahlen ausgegangen. Mit jedem Rechner lassen sich darüber hinaus auch nicht-ganzzahlige Dual- bzw. Dezimalbrüche verarbeiten. In ungenauer Diktion spricht man hier allgemein in der Informatik von der Verarbeitung reeller Zahlen (vom Typ `REAL`). Dies geschieht durch Verwendung spezieller Darstellungen:

Ganze Zahlen lassen sich formal als Dezimalbrüche schreiben, wenn man hinter das Komma die Ziffer 0 schreibt. Es ist z. B. $23 = 23.0$. Damit kann die bisher ausschließlich verwendete `INTEGER`-Darstellung von Zahlen als Darstellung von Zahlen

mit Komma aufgefasst werden, wenn man unterstellt, dass das Komma logisch rechts vom rechtesten Bit steht. Allgemein spricht man von einer *Festpunkt-Darstellung*, wenn eine Zahl durch eine n-stellige Dual- (bzw. Dezimal-) Zahl (eventuell komplementiert) dargestellt wird, wobei das Komma an beliebiger, aber *fester* Stelle angenommen wird.

Beispiel 6.12 (a) Das Komma wird rechts von der Stelle mit dem niedrigsten Wert angenommen. Ein n-Bit Wort $(x_{n-1}, \ldots, x_0)_2$ stellt dann die Zahl

$$z = \sum_{i=0}^{n-1} x_i \cdot 2^i \quad \text{dar.}$$

(b) Das Komma wird links von der Stelle mit dem höchsten Wert angenommen. Ein n-Bit Wort $(x_1, \ldots, x_n)_2$ stellt dann die Zahl

$$z = \sum_{i=1}^{n} x_i \cdot 2^{-i} \quad \text{dar.}$$

Ist z. B. $n = 8$, so ist 10110010 die Darstellung von

$$1 \cdot 2^{-1} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-7}$$

$$= \frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{128} = \frac{89}{128} = 0,6953125.$$

□

Allgemein stellt eine Bitfolge $(x_{n-1}, \ldots, x_1, x_0, x_{-1}, \ldots, x_{-m+1}, x_{-m})_2$, falls der („Dual"-) Punkt rechts von der Stelle x_0 angenommen wird, die Zahl

$$x = \sum_{i=-m}^{n-1} x_i 2^i$$

dar. Sollen auch negative Zahlen dargestellt werden können, so ist wieder ein Bit für das Vorzeichen zu reservieren oder eine der im letzten Abschnitt behandelten Komplement-Darstellungen zu verwenden.

Das letzte Beispiel zeigt insbesondere, dass ein Dual-Bruch auf einfache Weise in einen Dezimal-Bruch umgerechnet werden kann.

Das Gleiche gilt in umgekehrter Richtung: Die entsprechende Transformation verläuft „komplementär" zu der in Kapitel 1 beschriebenen Transformation natürlicher Zahlen ins Dualsystem (vgl. Beispiel 1.3). Anstatt durch die Basis zu *dividieren* und die entstehenden Reste in umgekehrter Reihenfolge zu lesen, wird jetzt mit der Basis *multipliziert* und die vor dem Komma entstehenden Ergebnisse werden in der Reihenfolge des Entstehens gelesen:

Beispiel 6.13 Zur Darstellung von 0,375 als Dualbruch gehen wir wie folgt vor:

$$
\begin{array}{r}
0{,}375 \\
\times\ 2 \\
\hline
0{,}750 \\
\times\ 2 \\
\hline
1{,}500 \\
\times\ 2 \\
\hline
1{,}000
\end{array}
$$

In jedem Schritt wird der *links* vom Komma entstehende Anteil des Ergebnisses der letzten Multiplikation ignoriert; die Berechnung endet, falls *rechts* vom Komma ausschließlich Nullen auftreten. Das Ergebnis lautet also

$$(0,375)_{10} = (0,011)_2 \ .$$

□

In genau der gleichen Weise kann mit jeder anderen Basis (8, 16 usw.) verfahren werden, falls die Basis lediglich 2 oder 5 als Faktoren enthält. Es sei nicht verschwiegen, dass bei Rechnungen mit höchster Genauigkeit (Dezimalstellen im Millionenbereich) die Konvertierung einen beträchtlichen Rechenaufwand erfordert.

Um Operationen mit Festkomma-Zahlen durchführen zu können, ist natürlich wesentlich, dass das Komma bei allen Operanden an der gleichen Stelle angenommen wird. Das bedeutet, dass Operanden gegebenenfalls zu transformieren sind. Werden z. B. bei einer Wortlänge von 16 Bits 12 Stellen vor und 4 hinter dem Komma angenommen, so muss z. B. die Zahl 0,00011101 durch

$$0000\ 0000\ 0000\ [,]\ 0001$$

abgerundet dargestellt werden, wobei 4 signifikante Stellen verloren gehen. Diesen Nachteil vermeidet die Gleitkomma-Darstellung.

Bei der *Gleitkomma-Darstellung*, welche auch halblogarithmische Darstellung genannt wird, wird jede Zahl z in der Form

$$z = \pm m \times b^{\pm d}$$

dargestellt. Dabei heißt m *Mantisse* und d *Exponent*; b ist die Basis für den Exponenten. Es sei bemerkt, dass b nicht notwendig mit der Basis des zugrunde liegenden Zahlensystems übereinstimmt, welche in einem Rechner 2 ist.

Beispiel 6.14 Die dezimale Zahl 1228.8 ist wie folgt darstellbar:

$$1228.8 = 2.4 \times 8^3.$$

In diesem Fall gilt $b = 8$, während 10 die Basis des verwendeten Zahlensystems ist. □

Wir werden weiter unten sehen, aus welchen Gründen es sinnvoll sein kann, als Wert für b eine Zahl $\neq 2$ und insbesondere eine *Potenz* von 2 zu wählen. Für den

Moment wollen wir $b = 2$ annehmen. Da die Basis für alle auftretenden Exponenten von Gleitkomma-Zahlen die gleiche ist, braucht sie insbesondere nicht gespeichert zu werden; die rechnerinterne Darstellung einer Gleitkomma-Zahl kann daher als ein Paar

$$(\pm m\ ,\ \pm d)$$

angesehen werden.

Wesentlich für das Rechnen mit Gleitkomma-Zahlen ist die Beobachtung, dass die Gleitkomma-Darstellung einer gegebenen Zahl nicht eindeutig ist; eine *Gleitkomma-Operation* erfordert daher unter Umständen gewisse Vorbereitungen.

Beispiel 6.15 (a) Es gilt z. B.

$$\begin{aligned} 1228,8 &= 12,288 \times 10^2 \\ &= 0,12288 \times 10^4 \\ &= 122880 \times 10^{-2} \end{aligned}$$

(b) Für eine Addition von $1,2288 \times 10^3$ und $0,000375 \times 10^7$ wird man zunächst den zweiten Operanden durch $3,75 \times 10^3$ darstellen, um sodann

$$(1,2288 + 3,75) \times 10^3$$

rechnen zu können. □

Zur Vermeidung von Problemen im Zusammenhang mit der Nicht-Eindeutigkeit einer Gleitkomma-Darstellung wird in realen Rechnern eine *normalisierte* Darstellung verwendet:

Definition 6.2 Eine Gleitkomma-Zahl der Form $\pm m \cdot b^{\pm d}$ heißt *normalisiert*, falls gilt:

$$\frac{1}{b} \leq |m| < 1$$

Im Fall $b = 2$ (als Basis für Exponent *und* Mantisse) folgt hieraus unmittelbar, dass für die Mantisse einer normalisierten Gleitkomma-Zahl gilt:

$$\frac{1}{2} \leq |m| < 1$$

Mit anderen Worten wird das Komma links von der linkesten Stelle der Mantisse angenommen, und die höchstwertige (Binär-) Stelle der Mantisse (d. h. das am weitesten links stehende Bit) ist $\neq 0$.

Beispiel 6.16 (a) Die normalisierte Darstellung von (dual)

$$0,000011101$$

lautet

$$0,11101 \times 2^{-4}.$$

(b) Die normalisierte Darstellung von (dual)

$$10011,101 \times 2^{10}$$

lautet

$$0,10011101 \times 2^{15}.$$

□

Im Fall $b \neq 2$ gilt analog, dass das Komma links von der Mantisse angenommen wird, und dass die erste Ziffer der Mantisse *zur Basis b* ungleich 0 ist:

Beispiel 6.17 (a) Es sei $b = 8$, und gesucht sei die normalisierte Darstellung von

$$(0,000011)_2 \times 8^2.$$

Die Mantisse m dieser Darstellung muss die Ungleichung

$$\frac{1}{8} \leq |m| < 1$$

erfüllen. Die binäre Mantisse 0,000011 kann oktal als 0,03 geschrieben werden, d. h. als normalisierte Darstellung ergibt sich

$$\begin{aligned}(0,000011)_2 \times 8^2 &= (0,03)_8 \times 8^2 \\ &= (0,3)_8 \times 8^1 \\ &= (0,011)_2 \times 8^1\end{aligned}$$

Die erste, dem Komma folgende *Oktal*-Ziffer ist also $\neq 0$. Anders ausgedrückt entspricht jetzt eine Veränderung der Exponenten um 1 einer Multiplikation mit oder Division durch 8 ($= 2^3$), so dass das Komma nicht um einzelne Stellen, sondern nur um drei Stellen gleichzeitig verschoben werden kann.

(b) Es sei $b = 16$. Eine entsprechende Argumentation wie unter (a) ergibt, dass die Zahl

$$(0,000000110101)_2 \times 16^4$$

die normalisierte Darstellung

$$(0,00110101)_2 \times 16^3$$

besitzt. □

Wir wenden uns als nächstes der rechnerinternen Darstellung von Gleitkomma-Zahlen zu. Offensichtlich ist dazu zunächst festzulegen, wie viele Bits für eine Mantisse und wie viele für einen Exponenten reserviert werden sollen. Als Beispiel betrachten wir einen Rechner der Wortlänge 32 Bits und unterstellen folgende Aufteilung: 1 Bit werde für das Vorzeichen der Mantisse verwendet, 23 Bits für die Mantisse (d. h. für die Mantisse insgesamt wird die Vorzeichen/Betrags-Darstellung verwendet) und 8 Bits für den Exponenten. Weiter werde eine Mantisse normalisiert gespeichert, die Basis des Exponenten sei 2, und der Exponent werde im Zweier-Komplement dargestellt. Dann ist z. B.

$$\underbrace{0}_{\text{VZ}}\ \underbrace{10011101001110011000000}_{\text{Mantisse}}\ \underbrace{00001101}_{\text{Exponent}}$$

die Darstellung der Zahl

$$+(0,10011101001110011)_2 \times 2^{13} = (1001110100111,0011)_2 = (5031,1875)_{10} .$$

Man überlegt sich leicht, dass mit der gerade genannten Aufteilung *positive* Zahlen z im Bereich

$$0,5 \times 2^{-128} \leq z \leq (1 - 2^{-23}) \times 2^{127}$$

und *negative* Zahlen z im Bereich

$$-(1 - 2^{-23}) \times 2^{127} \leq z \leq -0,5 \times 2^{-128}$$

darstellbar sind. Es folgt, dass um den Nullpunkt herum ein kleines „Loch" auf der Zahlenachse nicht erfasst ist, welches insbesondere die Null selbst enthält. Zur Darstellung der Null wird daher im Allgemeinen von der üblichen Konvention zur Darstellung von Gleitkomma-Zahlen abgewichen; so kann man 0,0 darstellen als positive Zahl (d. h. Vorzeichen-Bit = 0) mit dem Exponenten 0, der Wert der Mantisse wird dabei „ignoriert".

Hierdurch wird gleichzeitig das folgende Problem gelöst: Falls $b = 2$ gilt für die Basis b des Exponenten, so ist bei der Mantisse *jeder* normalisierten Gleitkomma-Zahl das am weitesten links stehende Bit = 1. Daher braucht dieses Bit nicht gespeichert zu werden (man spricht von einem „hidden bit"), so dass ein weiteres Bit für die Mantisse zur Verfügung steht. In vielen Rechnern wird dieser Trick zur Erhöhung der Genauigkeit für die Mantisse angewendet; allerdings bedeutet dann die als $0 \ldots 0$ gespeicherte Mantisse *nicht* die Zahl „0,0", sondern $\frac{1}{2}$. In jedem Fall ist für einen Rechner sicherzustellen, dass Verwechselungen mit 0,0 ausgeschlossen sind.

Die größte darstellbare Gleitkomma-Zahl ist also (bei der oben genannten Aufteilung von Bits auf Mantisse und Exponent) $\approx 2^{127}$, während bei Verwendung der Festkomma-Dualdarstellung mit 32 Bits maximal die Zahl $2^{32} - 1$ dargestellt werden kann. Es folgt, dass unter Verwendung von Gleitkomma-Zahlen ein erheblich größerer Zahlenbereich darstellbar wird; allerdings ist dies mit Einbußen hinsichtlich der Genauigkeit verbunden: Während vorher 32 Bits für die Mantisse zur Verfügung standen und damit etwa 10 signifikante Dezimalstellen darstellbar sind, sind mit 23 Bits nur noch etwa 7 Dezimalstellen erfassbar. Diese „Diskrepanz" zwischen Genauigkeit und darstellbarem Zahlenbereich wird noch vergrößert, falls eine andere Basis als 2 für den Exponenten verwendet wird:

Werden Exponenten etwa zur Basis 16 angenommen (anstatt 2), so werden mit im Zweier-Komplement repräsentierten 8-Bit-Exponenten Zahlen zwischen 16^{-128} und 16^{127} darstellbar. Offensichtlich ist auf diese Weise der darstellbare Zahlenbereich erheblich vergrößert, denn es gilt $2^{128} \approx 10^{38}$, aber $16^{128} \approx 10^{154}$; diese Erweiterung ist jedoch wieder mit einer verringerten Genauigkeit verbunden. Dieser „Trade-off" zwischen darstellbarem Zahlenbereich und erzielbarer Genauigkeit wird in realen Rechnern im Allgemeinen dadurch wenigstens teilweise aufgefangen, dass verschiedene Formate zur Darstellung von Gleitkomma-Zahlen zur Verfügung stehen.

Schließlich sei erwähnt, dass Exponenten häufig nicht im Zweier-Komplement repräsentiert werden, sondern es wird die so genannte *Biased*-Notation, auch *Excess*-Darstellung genannt, verwendet. Als Beispiel betrachten wir durch 8 Bits dargestellte Exponenten d, für welche bei Verwendung des Zweier-Komplements $-128 \leq d \leq 127$ gilt. Durch Addition von $128 = 2^{8-1}$ zu jedem solchen d erhält man dann Exponenten

d' im Bereich $0 \leq d' \leq 255$. Diese „Verschiebung" der Darstellung hat den Vorteil, dass der Vergleich von Exponenten erleichtert wird: Falls $d_1 \leq d_2$ gilt, so gilt das Gleiche für die (gewöhnlichen) Dualdarstellungen von d_1 und d_2. Dies ist von Bedeutung für die Ausführung von Operationen auf Gleitkomma-Zahlen wie etwa einer Addition, welche – wie weiter unten erwähnt – gleiche Exponenten voraussetzt.

Allgemein erhält man die Excess-Darstellung d' eines Exponenten d, falls g Bits für diesen zur Verfügung stehen, wie folgt:

$$d' := d + 2^{g-1}$$

Für $g = 8$ spricht man z. B. von der *Excess-128*-Darstellung.

6.3 Rechnerarithmetik, insbesondere Multiplikation

In diesem Abschnitt knüpfen wir an die Ausführungen von Kapitel 1 über die Ausführung arithmetischer Operationen an. Addier-Netze bzw. -Werke haben wir in Kapitel 2 bzw. 5 vorgestellt, und diese können allgemeiner auch zur Multiplikation bzw. Division benutzt werden. Wir wollen die Durchführung von Multiplikationen jetzt vor allem für Festkomma-Zahlen behandeln (und auf die Behandlung der Division verzichten).

Grundsätzlich werden Multiplikation und Division im Dualsystem in der gleichen Weise wie im Dezimalsystem durchgeführt. Bei der Multiplikation wird der Multiplikand nacheinander mit jedem einzelnen Bit des Multiplikators multipliziert, wobei jeweils ein Teilprodukt entsteht. Beginnt man mit dem am weitesten rechts stehenden Bit, so wird ab der zweiten Stelle das Teilprodukt jeweils um eine Stelle nach links geschoben. Das Ergebnis erhält man schließlich durch Summation aller Teilprodukte.

Beispiel 6.18 Wir betrachten die Aufgabe 13×9:

1101	Multiplikand
1001	Multiplikator
1101	
0000	Teilprodukte
0000	
1101	
1110101	Ergebnis

□

Bei der Division einer Dualzahl durch eine andere entstehen neue Probleme, die hier nicht behandelt werden sollen.

Dieses Beispiel zeigt bereits, dass die Multiplikation höheren physikalischen Aufwand erfordert als die Addition; neben einem doppelt langen Ergebnis-Register muss die Hardware in der Lage sein, Shift-Operationen durchzuführen. Außerdem zeigt das Beispiel, dass die bekannte Schulmethode nicht sehr effizient ist: Nullen, welche im Multiplikator auftreten, erfordern den gleichen Rechenaufwand wie Einsen, tragen

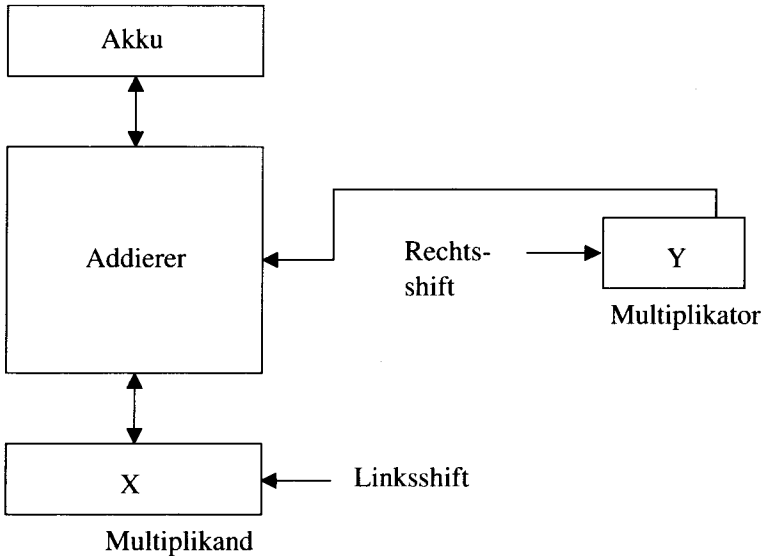

Abbildung 6.1: Schaltung zur Multiplikation.

aber nicht zum Ergebnis bei; Teilprodukte sind (scheinbar) zwischenzuspeichern und erst im letzten Schritt zu addieren. Formal ist dieses Verfahren wie folgt beschreibbar:

Sei x der Multiplikand, $y = (y_{n-1}, \ldots, y_0)$ der Multiplikator, dann ist

$$\begin{aligned} x \cdot y &= x \cdot y_0 + x \cdot y_1 \cdot 2 + x \cdot y_2 \cdot 2^2 + \ldots + x \cdot y_{n-1} \cdot 2^{n-1} \\ &= \sum_{i=0}^{n-1} x \cdot y_i \cdot 2^i \end{aligned}$$

In der Praxis ist es sinnvoll, jeden Term der Form $x \cdot y_i \cdot 2^i$ zu addieren, sobald er generiert wurde, so dass man obige Multiplikation wie folgt ausführen kann (es handelt sich um einen mit dem so genannten *Horner-Schema* verwandten Rechentrick):

$$x \cdot y = (\ldots((x \cdot y_0 + x \cdot y_1 \cdot 2) + x \cdot y_2 \cdot 2^2) \ldots + x \cdot y_{n-1} \cdot 2^{n-1})$$

Ohne die Verwendung eines doppelt langen Registers bzw. eines Registerpaares zur Aufnahme des Ergebnisses kommt man aus, wenn beide Operanden auf halbe Wortlänge beschränkt werden. Damit ist z. B. die in Abbildung 6.1 gezeigte Schaltung in der Lage, die in Beispiel 6.20 angegebene Multiplikation auszuführen.

Diese Multiplikation läuft wie folgt ab: Der Akku-Inhalt wird gelöscht, die rechten 4 Bits des X-Registers nehmen den Multiplikanden auf, Y den Multiplikator. Eine zusätzliche Logik, welche in Abbildung 6.1 nicht gezeigt ist, testet das 0-te Bit von Y. Da im letzten Beispiel $y_0 = 1$ gilt, wird der Inhalt von X zum Akku-Inhalt addiert, so dass dieser nun das (erste) Teilergebnis 0000 1101 enthält. Sodann wird der Inhalt von Y (durch ein entsprechendes Steuersignal) um ein Bit nach rechts, der Inhalt von X um ein Bit nach links geschoben. Da $y_1 = 0$ ist, wird im zweiten Schritt keine

Addition, sondern lediglich ein erneuter Shift von X und Y um jeweils eine Stelle ausgeführt. Der dritte Schritt verläuft analog, so dass zu Beginn des vierten Schrittes X den Wert 0110 1000 enthält. Wegen $y_3 = 1$ wird dann der Inhalt von X zum Akku-Inhalt addiert, so dass dieser das Ergebnis 0111 0101 erhält.

Die in Abbildung 6.1 angegebene Schaltung läßt sich leicht auf den Fall erweitern, dass Zahlen *mit* Vorzeichen zu verarbeiten sind. Eine zusätzliche Logik bestimmt dann das Vorzeichen des Ergebnisses, welches „+" ist, falls Multiplikand und Multiplikator das gleiche Vorzeichen haben, und „−" sonst.

Die Beschränkung auf Operanden halber Wortlänge kann entfallen, wenn bei Multiplikand und Multiplikator das Komma ganz links angenommen wird, wie es etwa bei den normierten Mantissen in Gleitkomma-Technik der Fall ist. In diesem Fall ist der Betrag beider kleiner als 1, so dass auch das Produkt dem Betrag nach kleiner als 1 ist.

Es sei darauf hingewiesen, dass sich die oben vorgeführte Multiplikation nach der Schulmethode (wie auch die Division) hardwaremäßig *beschleunigen* lässt. Möglichkeiten hierzu bieten z. B. die Verwendung eines Addiernetzes mit schneller Carry-Berechnung, eine Zwischenspeicherung des bei der Addition auftretenden Übertrags und Verarbeitung desselben erst in späteren Schritten oder ein Malnehmen des Multiplikanden in jedem Schritt nicht nur mit einem Bit des Multiplikators, sondern mit $k > 1$ benachbarten Bits. Grundlage schneller Multiplizierer sind im Allgemeinen schnelle Addierer, was wir am Beispiel des in Abschnitt 2.5 beschriebenen Carry-Save-Addiernetzes demonstrieren wollen (man bezeichnet das Ergebnis als *Carry-Save-Multiplikation*). Wir betrachten dazu noch einmal die im Kontext von Beispiel 6.20 gezeigte Berechnung, welche sich in tabellarischer Form für $n = 4$ wie folgt darstellen lässt:

				x_3	x_2	x_1	x_0	x
				$\times\ y_3$	y_2	y_1	y_0	y
0	0	0	0	x_3y_0	x_2y_0	x_1y_0	x_0y_0	M_1
0	0	0	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0	M_2
0	0	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0	M_3
0	x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0	M_4

Wie bereits beschrieben, lässt sich eine Multiplikation von x und y durch Addition der in dieser Tabelle gezeigten Zeilen implementieren. Da es sich im konkreten Fall um vier Zeilen handelt, reicht ein zweistufiges CSA-Netz aus, durch welches die Anzahl der Summanden auf zwei reduziert wird; diese werden sodann durch irgendein Addiernetz summiert. Dieses Prinzip ist in Abbildung 6.2 veranschaulicht. In dieser Abbildung wird einerseits unterstellt, dass es sich bei den wie in obiger Tabelle mit M_i, $1 \leq i \leq 4$, bezeichneten Summanden jeweils um achtstelligen Dualzahlen handelt. Ferner wird angenommen, dass die M_i durch weitere Hardware aus den Operanden x und y bereits erzeugt sind (jeweils durch ein Schaltnetz mit vier Und-Gattern).

Bei der Multiplikation von zwei Zahlen mit größerer Stellenzahl n wird man zur schnellen Reduktion der Anzahl n der Summanden den in Kapitel 2 vorgestellten Wallace-Tree verwenden, so dass die Tiefe der Schaltung (vor der abschließenden Addition) logarithmisch in der Stellenzahl n der Faktoren wird.

Es sei der Vollständigkeit halber erwähnt, dass die Grundlage aller hier vorgestellten Multiplikationsverfahren die bekannte Methode der schriftlichen Multiplikation

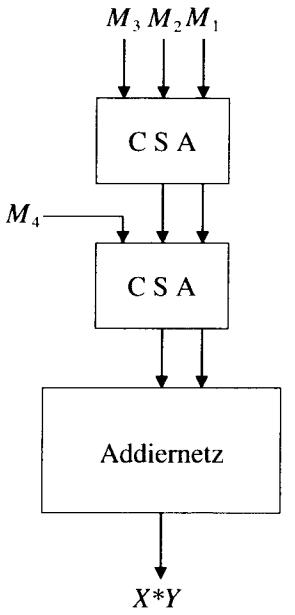

Abbildung 6.2: Carry-Save-Multiplikation.

ist, die im Prinzip auf die arabische Mathematik des 9. Jahrhunderts zurück geht (Al Chwarizmi) und die in Deutschland insbesondere durch die Bücher von Adam Riese im 15. Jahrhundert populär geworden ist. Diese Schulmethode führt — wie jeder weiß, der die „Päckchenmethode" kennt — bei dem Produkt zweier n-stelliger Zahlen zu einer Schrittzahl der Größenordnung n^2. Inzwischen kennt man Algorithmen (Karatsuba 1962, Schönhage und Strassen 1971), welche für sehr große n mit einer deutlich geringeren Schrittzahl auskommen: Die Größenordnungs-Potenz von n kann beliebig nahe an 1, aber oberhalb von 1, gewählt werden. Die Software-Implementierungen dieser (rekursiven) Algorithmen basieren aber darauf, dass für klein-dimensionierte Produkte die von uns beschriebenen Hardware-Techniken verwendet werden.

Die bisher beschriebenen Schaltungen bzw. Verfahren lassen sich auch zur Ausführung von Operationen an Gleitkomma-Zahlen benutzen, indem man die Mantissen und die Exponenten nacheinander in der Hardware verarbeitet. Zur Multiplikation (Division) sind dann die Mantissen zu multiplizieren (dividieren) und die Exponenten zu addieren (subtrahieren). Bei Addition bzw. Subtraktion von Gleitkomma-Zahlen muss man darauf achten, dass die Operanden gleiche Exponenten haben, was im Allgemeinen nur dadurch zu erreichen ist, dass der Operand mit dem kleineren Exponenten „denormalisiert" wird. Durch die *feste* Wortlänge und insbesondere durch die fest gewählte Anzahl von Bits zur Darstellung von Mantisse bzw. Exponent muss man dabei beachten, dass für Gleitkomma-Zahlen nicht alle der üblichen Rechengesetze gelten, da diese keinen Körper im Sinne der Algebra bilden. Wir erläutern dies

am Beispiel des Assoziativgesetzes: Sei

$$\begin{aligned} x &= +0,1235 \cdot 10^3 \\ y &= +0,5512 \cdot 10^5 \\ z &= -0,5511 \cdot 10^5 \end{aligned}$$

so gilt:

$$\begin{aligned} x+y &= +0,1235 \cdot 10^3 + 0,5512 \cdot 10^5 \\ &= +0,0012 \cdot 10^5 + 0,5512 \cdot 10^5 \\ &= +0,5524 \cdot 10^5 \end{aligned}$$

$$\begin{aligned} (x+y)+z &= +0,5524 \cdot 10^5 - 0,5511 \cdot 10^5 \\ &= +0,0013 \cdot 10^5 \\ &= +0,1300 \cdot 10^3 \end{aligned}$$

Andererseits gilt:

$$\begin{aligned} y+z &= +0,5512 \cdot 10^5 - 0,5511 \cdot 10^5 \\ &= +0,1000 \cdot 10^2 \end{aligned}$$

$$\begin{aligned} x+(y+z) &= +0,1235 \cdot 10^3 + 0,1000 \cdot 10^2 \\ &= +0,1235 \cdot 10^3 + 0,0100 \cdot 10^3 \\ &= +0,1335 \cdot 10^3 \end{aligned}$$

Das Assoziativgesetz ist verletzt: $(x+y)+z$ und $x+(y+z)$ haben verschiedene Werte.

Zusammenfassend gelten für das Rechnen mit Gleitkomma-Zahlen folgende Regeln: Seien $x = m_x \cdot 2^{d_x}$, $y = m_y \cdot 2^{d_y}$:

$$\begin{aligned} x+y &= (m_x \cdot 2^{d_x - d_y} + m_y) \cdot 2^{d_y} \text{ falls } d_x \leq d_y \\ x-y &= (m_x \cdot 2^{d_x - d_y} - m_y) \cdot 2^{d_y} \text{ falls } d_x \leq d_y \\ x \cdot y &= (m_x \cdot m_y) \cdot 2^{d_x + d_y} \\ x : y &= (m_x : m_y) \cdot 2^{d_x - d_y} \end{aligned}$$

Wenn diese Operationen mit den bereits bekannten Addierwerken ausgeführt werden sollen, muss die Behandlung von Mantissen und Exponenten im Allgemeinen softwaremäßig erfolgen. Für eine Addition bedeutet dies z. B. genauer:

1. Vergleich der Exponenten.
2. Shift der Mantisse der Zahl mit dem kleineren Exponenten.
3. Ausführung der eigentlichen Addition.
4. gegebenenfalls Normalisierung des Ergebnisses (siehe unten).

Um dies hardwaremäßig zu bewerkstelligen und damit die Ausführungszeit von Gleitkomma-Operationen in ähnliche Größenordnungen wie die von Festkomma-Operationen zu bringen, sind viele Rechenanlagen mit separaten Gleitpunkt-Rechenwerken (*Floating-Point-Prozessoren*) ausgestattet. Diese verfügen dann über hinreichend lange Register zur Aufnahme von Operanden bzw. Ergebnissen, über eine geeignete Verknüpfungslogik und über ein eigenes Steuerwerk.

Schließlich sei erwähnt, dass bei Gleitkomma-Operationen sowohl *Overflows* als auch *Underflows* auftreten können. So führt z. B. die Addition

$$\begin{array}{r} 0,537 \times 10^2 \\ +0,520 \times 10^2 \\ \hline 1,057 \times 10^2 \end{array}$$

auf einen Overflow, da die Mantisse des Ergebnisses eine signifikante Ziffer *links* vom Komma besitzt. Entsprechend führt z. B. die Subtraktion

$$\begin{array}{r} 0,5678 \times 10^5 \\ -0,5643 \times 10^5 \\ \hline 0,0035 \times 10^5 \end{array}$$

zu einem Underflow, da jetzt eine 0 unmittelbar *rechts* vom Komma auftritt. Beides ist offensichtlich durch eine Normalisierung zu beheben.

Außerdem sei darauf hingewiesen, dass speziell eine Gleitkomma-Arithmetik in Rechnern stets mit *Rundungsfehlern* behaftet ist, welche zum Teil daher rühren, dass der Zwang zur Normalisierung ein „Abschneiden" signifikanter Ziffern erfordert.

6.4 Darstellung alphanumerischer Daten

Zum Abschluss dieser Ausführungen über Informationsdarstellung wollen wir uns noch mit der Darstellung alphanumerischer Daten, also insbesondere von Text, beschäftigen. Bei unseren Ausführungen haben wir uns zwar bisher auf arithmetische Operationen auf Zahlen beschränkt, jedoch müssen reale Rechner in der Praxis auch eine Vielzahl anderer Aufgaben erledigen. Hierzu gehören z. B. die Verarbeitung, insbesondere Übersetzung von Befehlen einer (höheren) Programmiersprache (in ausführbare Befehle der Maschinensprache) oder — wenn man an das Arbeiten mit Texten, Dateien oder Datenbanken denkt — das Klassifizieren und Sortieren von Text. Derartige Texte bestehen im Allgemeinen aus einer Vielzahl einzelner Zeichen wie Buchstaben, Ziffern, Punkt, Komma und anderer Sonderzeichen. Zur Verarbeitung von Informationen, welche allgemein aus so genannten *Characters* bestehen, ist zunächst die Wahl einer geeigneten rechnerinternen Darstellung durch Bits wesentlich. Wie bei Zahlendarstellungen finden auch dabei wieder spezielle *Codes* (vgl. Abschnitt 5.4.2) Verwendung.

6.4.1 Der ASCII-Code

Beim Entwurf bzw. bei der Auswahl eines solchen Codes ist zuerst die Frage zu klären, welchen Umfang der darstellbare Zeichensatz haben soll. Stehen z. B. 64 Code-Worte als Bit-Folgen der Länge 6 zur Verfügung, so sind damit 26 Buchstaben, 10 Ziffern und