Informatik interaktiv

Herausgegeben von
Prof. Dr. Michael Lutz und Prof. Dr. Christian Märtin
Fachhochschule Augsburg, Fachbereich Informatik

Zu dieser Buchreihe

Die Werke dieser Reihe bieten einen gezielten Einstieg in grundlegende oder besonders gefragte Themenbereiche der Informatik und benachbarter Disziplinen. Alle Autoren verfügen über langjährige Erfahrung in Lehre und Forschung zu den jeweils behandelten Themengebieten und gewährleisten Praxisnähe und Aktualität.

Die Verknüpfung zeitgemäßer und attraktiver Lehrbücher mit darauf abgestimmten multimedialen Lernsoftware-CDs berücksichtigt individuelle Lerngewohnheiten und lockert den Lernprozess auf. Eine interaktive Auseinandersetzung mit den Themen kann den Lernerfolg verstärken und das Interesse am Lehrstoff vertiefen. Besonderer Wert wurde auf reichhaltiges Übungsmaterial gelegt, das über die CDs einfach zugänglich ist und am PC rasch nachvollziehbar wird. Ergänzende Informationen befinden sich ebenfalls auf der CD oder können über Web-Links abgerufen werden. Als besonderen Bonus enthalten die CDs teils kommerzielle, teils prototypische Software-Werkzeuge, die zur weiterführenden Beschäftigung mit dem Lehrstoff geeignet sind.

Die Bände der Reihe können vorlesungsbegleitend oder zum Selbststudium eingesetzt werden. Sie lassen sich teilweise modular miteinander kombinieren. Wegen ihrer Kompaktheit sind sie gut geeignet, bestehende Lehrveranstaltungen zu ergänzen und zu aktualisieren.

Titel in dieser Reihe:

- **Bernd Breutmann, Data and Algorithms, An introductory Course**
- **Peter Forbrig, Objektorientierte Softwareentwicklung mit UML**
- **Peter Forbrig, Introduction to Programming by Abstract Data Types**
- **Fritz Jobst, Einführung in Java**
- **Wolfgang L. J. Kowarschick, Multimedia-Programmierung**
- **Henning Mittelbach, Einführung in C**
- **Henning Mittelbach, Einführung in C++**
- **Rainer Oechsle, Parallele Programmierung mit Java Threads**
- **Wolfgang Riggert, Rechnernetze, Technologien – Komponenten – Trends**

Rechnergrundlagen

Von der Binärlogik zum Schaltwerk

von Prof. Dr. Rainer Kelch

mit 95 Bildern, 54 Tabellen, 70 Beispielen, 25 Aufgaben
sowie einer CD-ROM

Fachbuchverlag Leipzig
im Carl Hanser Verlag

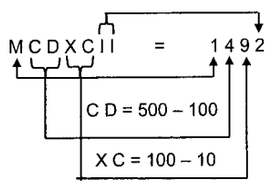
2.1 TABELLE **b-adische Stellenwertsysteme**

Basis b	Bezeichnung	Ziffernbereich: $a_i \in \{0, \dots, b-1\}$
2	Binär, Dual	$a_i \in [0, 1] = \{0, 1\}$
8	Oktal	$a_i \in [0, 7] = \{0, 1, \dots, 7\}$
10	Dezimal	$a_i \in [0, 9] = \{0, 1, \dots, 9\}$
16	Hexadezimal	$a_i \in [0, 15] = \{0, 1, \dots, 9, A, \dots, F\}$

Hinweis:
Eine Zahl muss immer interpretiert werden, denn der Wert einer Zahl (=Ziffernfolge) ist abhängig von der Basis. Ohne Angabe der Basis ist der Wert nicht eindeutig definiert.

Weniger bekannt sind *Nicht-Stellenwertsysteme*. Die römischen Zahlen sind ein Beispiel dafür und uns über die Bezeichnungen auf Ziffernblättern sowie aus der Literatur und Geschichte bekannt (siehe Bild 2.2).

2.2 BILD **Römische Zahlen**



Wir betrachten im Folgenden nur Stellenwertsysteme.

2.2 Rechnen innerhalb eines Zahlensystems

Wir betrachten die vier Grundrechenarten. Die Subtraktion kann direkt auf die Addition zurückgeführt werden, die Division auf die Multiplikation. Die Multiplikation kann zwar auch auf die Addition zurückgeführt werden, wird hier aber gesondert behandelt.

Addition und Subtraktion
Satz 2.1 zeigt die Regel, nach der zwei b-adische Zahlen addiert bzw. subtrahiert werden.

2.1 SATZ **Addition und Subtraktion von b-adischen Zahlen**

Für die beiden b-adischen Zahlen x und y gemäß

$$x = (x_N \dots x_2 x_1 x_0 , x_{-1} x_{-2} \dots x_{-M}) = \sum_{i=-M}^{N} x_i b^i$$

$$y = (y_N \dots y_2 y_1 y_0 , y_{-1} y_{-2} \dots y_{-M}) = \sum_{i=-M}^{N} y_i b^i$$

wird die Summe $x + y$ bzw. die Differenz $x - y$ nach folgender Regel gebildet:

$$x \pm y = \sum_{i=-M}^{N} x_i b^i \pm \sum_{i=-M}^{N} y_i b^i = \sum_{i=-M}^{N} (x_i \pm y_i) b^i$$

Bemerkungen:
Ohne Beeinträchtigung der Allgemeinheit (o.B.d.A.) können N, M in Satz 2.1 bei x und y identisch gewählt werden. Als Lösung für den eventuell eintretetenden Ziffern-Überlauf ($x_i + y_i > b - 1$) wird ein Übertrag eingeführt, der auf die nächst links stehende Ziffer addiert wird (siehe Beispiel 2.1). Für den Unterlauf bei der Subtraktion ($x_i - y_i < 0$) wird ähnlich verfahren.

2.1 BEISPIEL **Übertrag bei Addition**

$x_i = 6; y_i = 7;$ $x_i + y_i = 13$
$i = 5; b = 10;$ Teilsumme: $x_5 * 10^5 + y_5 * 10^5 =$
$(6 + 7) * 10^5 = (13) * 10^5 =$
$(10 + 3) * 10^5 = 1 * 10^6 + 3 * 10^5$

```
x     = 1 2 6 0 3 1 5 . 2
y     = 1 2 7 1 4 2 3 . 3
          1
x + y = 2 5 3 1 7 3 8 . 5
```

Übertrag

Hinweis:
Die exakte formale Notation für die Behandlung des Übertrages ist kompliziert, hingegen ist die passende manuelle Rechentechnik bekannt und einfach. Wir verzichten deshalb hier auf eine exakte formale Notation.

Multiplikation und Division
Die Multiplikation zweier b-adischer Zahlen x und y geschieht gemäß der Regel aus Satz 2.2.

2.2 SATZ **Multiplikation zweier b-adischer Zahlen**

Das Produkt der beiden b-adischen Zahlen x und y (definiert gemäß Satz 2.1) berechnet sich wie folgt:

$$x * y = \left(\sum_{i=-M}^{N} x_i b^i\right) * \left(\sum_{j=-M}^{N} y_j b^j\right) = \sum_{i,j=-M}^{N} x_i b^i * y_j b^j = \sum_{i,j=-M}^{N} x_i * y_j b^{i+j}$$

Es gelten die Bemerkungen unter Satz 2.1 entsprechend.

In Beispiel 2.2 wird die bekannte Rechentechnik zur Multiplikation zweier Zahlen auch mit Übertragtechnik durchgeführt, illustriert anhand der Zahlen 14.2 und 37.4.

2.2 BEISPIEL **Multiplikation**

```
  1 4 . 2 * 3 7 . 4
1 2 1 1
      4 2 6 0 0
        9 9 4 0
          5 6 8
    1 2 1
    5 3 1 . 0 8
```

Die Division kann auf die Multiplikation zurückgeführt werden. Wir verzichten an dieser Stelle auf den Formalismus und verweisen auf Beispiel 2.5 und 2.9. Dort wird die Division sowohl dezimal als auch binär durchgeführt. Es wird dabei die Multiplikation ähnlich eingesetzt wie die Addition bei der Subtraktion. Im Abschnitt 2.5.4 kommen wir wieder auf den Divisionsalgorithmus zurück.

2.3 Konvertierung

Bei der *Konvertierung* werden Zahlen von einem in ein anderes Zahlensystem umgewandelt. Die für uns relevanten Konvertierungen sind z.B.:

dezimal → binär → dezimal

Für die technische Durchführung muss noch zwischen ganzen Zahlen und gebrochenen Zahlen (rational, reell) unterschieden werden.

2.3.1 Ganze Zahlen

Für die Umwandlung von einer Zahl zur Basis b_1 in die passende Zahl zur Basis b_2 gibt es zwei Verfahren. Wir gehen dabei o.B.d.A. davon aus, dass $b_1 < b_2$ gilt. Das *Horner-Schema* (siehe Algorithmus 2.1) nutzen wir für die Konvertierung von b_1 nach b_2 , das *Verfahren der fortgesetzten Division mit Rest* (siehe Algorithmus 2.2) für die Konvertierung von b_2 nach b_1.

2.1 ALGORITHMUS **Horner-Schema**

Eine b-adische Zahl u wird in eine b-adische Zahl v konvertiert, indem die folgende Formel angewandt wird:

$$(U)_b = \sum_{0}^{n} u_i b^i = ((\ldots (u_n b + u_{n-1}) b + \ldots) b + u_1) b + u_0 = (V)_b$$

In Beispiel 2.3 wird das Horner-Schema angewendet, um die Binärzahl $(11001)_2$ in eine Dezimalzahl zu konvertieren.

2.3 BEISPIEL **Horner-Schema zur Konvertierung von binär nach dezimal**

$$(1\,1\,0\,0\,1)_2 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 =$$

$$u_4 \quad u_3 \quad u_2 \quad u_1 \quad u_0$$

$$(((1 * 2 + 1) * 2 + 0) * 2 + 0) * 2 + 1 =$$
$$(((2 + 1) * 2 + 0) * 2 + 0) * 2 + 1 =$$
$$(3 * 2 * 2 + 0) * 2 + 1 =$$
$$12 * 2 + 1 = 25 = (25)_{10}$$

Das Verfahren der fortgesetzten Division mit Rest stellt eine Umkehrung des Horner-Schemas dar. Die allgemeine Formulierung finden Sie in Algorithmus 2.2.

2.2 ALGORITHMUS **Verfahren der fortgesetzten Division mit Rest**

Eine c-adische Zahl y soll in eine b-adische Zahl x umgeformt werden, so dass gilt:

$$(X)_b = \sum x_i b^i = (Y)_c$$

Für diese Konvertierung führen wir folgende Berechnungen durch:

$$x_0 = y \bmod b$$
$$x_1 = (y \operatorname{div} b) \bmod b$$
$$x_2 = ((y \operatorname{div} b) \operatorname{div} b) \bmod b$$

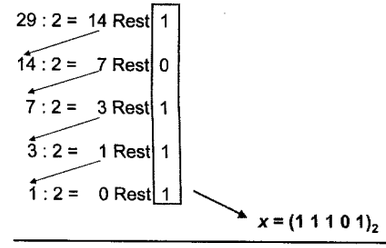
Reste

$$x_n = ((\ldots (y \operatorname{div} b) \operatorname{div} b \ldots) \operatorname{div} b) \bmod b = 0$$

Die Reste ergeben dann aneinandergereiht (erster Rest = Ziffer rechts, letzter Rest = Ziffer am weitesten links) die gesuchte Zahl x zur Basis b.

In Beispiel 2.4 wird die Dezimalzahl $(29)_{10}$ in eine Binärzahl konvertiert, indem darauf das Verfahren der fortgesetzten Division mit Rest angewendet wird.

2.4 BEISPIEL **Verfahren der fortgesetzten Division mit Rest: dezimal in binär**



2.3.2 Gebrochene Zahlen

Wir versuchen jetzt, uns an unsere frühe Schulzeit zu erinnern, als wir gelernt haben, Brüche als periodische Dezimalzahlen darzustellen.

$1 / 3 = 0.333 \ldots = 0.\overline{3}$

$1 / 9 = 0.111 \ldots = 0.\overline{1}$

$1 / 13 = ?$

Es gilt dabei, dass sich alle Brüche (bei denen der Nenner ausschließlich Primfaktoren enthält, die Teiler der Basis sind) als endliche Dezimalzahl darstellen lassen – alle anderen nur als unendliche periodische Dezimalzahlen. D.h. für Dezimalzahlen: Der Nenner

darf nur die Primfaktoren 2 und 5 enthalten, diese aber in beliebigen Potenzen. Damit ist der Bruch 1/5 als endlicher Dezimalbruch darstellbar. Wie sieht das für diesen Bruch aus, wenn wir ihn als Binärzahl darstellen wollen?

$(1/5)_{10} = (0.2)_{10}$
$(1/5)_{10} = (x)_2$ mit $x = ?$

→ Lösung:
$(1/5)_{10} = (1)_{10} / (5)_{10} = (1)_2 / (101)_2$

Diese binäre Division führen wir jetzt analog zum dezimalen Divisionsalgorithmus (siehe Beispiel 2.5) durch.

2.5 BEISPIEL **Binäre Division**

Division: 1.0000 ... : 101 = 0.0011̄0

```
 0
 ---
 10
  0
 ---
 100
  0
 ---
 1000
-101
 ---
 00110
-  101
 ---
   10
    0
```

Nebenrechnungen:

```
 1000
-  101
 ----
 0011
```

1 + 1 = (1)0 (Übertrag)

1 + 1 = 10
10 + x = 10 → x = 0

Wir testen jetzt, ob die Rechnung stimmt, indem wir die Probe machen:

Probe: $0.\overline{0011} = \frac{0011 : 11}{1111 : 11} = \frac{1}{101}$ → stimmt!

```
1111 : 11 = 101
11
--
 01
  0
 --
  11
```

Konsequenz: Endliche Dezimalbrüche, als Gleitkommazahl dargestellt, lassen sich i.Allg. im Binärsystem nicht durch eine Gleitkommazahl mit endlicher Mantisse darstellen. Dies führt unweigerlich zu Ungenauigkeiten auf dem Rechner, da dort nur mit endlich vielen Stellen gerechnet werden kann. Das Beispiel aus dem Prolog in Abschnitt 0.4 über den Absturz der Ariane V verdeutlicht diese Zusammenhänge auf fatale Weise. Vorsicht ist also geboten.

Die Konvertierung einer Binärzahl mit endlicher Mantisse in eine Dezimalzahl ist hingegen immer exakt möglich mit endlich vielen Dezimalstellen, sofern genügend Stellen zur Verfügung stehen (warum?). In Beispiel 2.6 wird die binäre rationale Zahl 0.1011 in eine dezimale rationale Zahl gewandelt. Dazu werden zuerst Zähler und Nenner in dezimale

Form gewandelt. Dann werden diese beiden Zahlen dividiert, entweder binär oder dezimal.

2.6 BEISPIEL **Konvertierung einer gebrochenen Zahl von binär nach dezimal**

$$(0.1011)_2 = \frac{(1011)_2}{(1111)_2} = \frac{(11)_{10}}{(15)_{10}} = \ldots = (0.7\overline{3})_{10}$$

Nebenrechnung:

```
11.0 : 15 = 0.73
0
---
110
105
---
 50
 45
---
 50
```

Fazit: Konvertierungsfehler (Erzeugung von unendlich-periodischen Brüchen) und Rundungsfehler (durch „Abschneiden" des unendlichen oder des endlichen Bruchs) sind typische System immanente Fehlerquellen.

Spezialfälle bei Konvertierung

Die Konvertierung zwischen b_1 und b_2, wenn $b_1 = b^n$, $b_2 = b^m$, und u.U. $n \mid m$ bzw. $m \mid n$ gilt, kann gesondert behandelt werden, da diese Konvertierungen sehr einfach durchzuführen sind. Schauen Sie sich dazu das Beispiel 2.7 an.

2.7 BEISPIEL **Konvertierung: Spezialfälle**

$b = 2$, $n = 1$, $m = 4$ → $b_1 = 2$, $b_2 = 16$ (bzw. 8), d.h.: Dualsystem bzw. Hexadezimalsystem.

$(A\ B\ C)_{16} = (10 * 16^2 + 11 * 16^1 + 12 * 16^0) =$
$= 2560 + 176 + 12 = (2748)_{10}$

Kontrolle:
$2^{10} + 2^9 + 2^7 + 2^5 + 2^4 + 2^3 + 2^2 =$
$2048 + 512 + 128 + 32 + 16 + 8 + 4 =$
2748

$(1\ 0\ 1\ 0\ \ 1\ 0\ 1\ 1\ \ 1\ 1\ 0\ 0)_2$

$(5\quad 2\quad 7\quad 4)_8$

Kontrolle:
$5 * 8^3 + 2 * 8^2 + 7 * 8^1 + 4 * 8^0 =$
$2560 + 128 + 56 + 4 = 2748$

2.4 Binäres Rechnen mit natürlichen Zahlen

Binäres Rechnen ist das Rechnen mit Binärzahlen, d.h. mit Zahlen, die im Binärsystem dargestellt werden. Dies benötigen wir für die Rechenoperationen, die auf Rechnern ausgeführt werden sollen. Das binäre Rechnen ist viel einfacher als das dezimale, nur etwas ungewohnt. Wir beginnen mit unserem einführenden Beispiel („kleines 1×1") und schreiben dieses dezimal explizit auf (siehe Tabelle 2.2).

2.2 TABELLE **Dezimales kleines 1×1**

*	0	1	2	...	9
0	0	0	0	...	0
1	0	1	2	...	9
2	0	2	4	...	18
:	:	:	:	:	:
9	0	9	18	...	81

Binär sieht die Tabelle für das kleine 1 × 1 und 1 + 1 deutlich einfacher aus (siehe Tabelle 2.3).

2.3 TABELLE **Binäres kleines 1×1 und 1 + 1**

*	0	1
0	0	0
1	0	1

+	0	1
0	0	1
1	1	2

← $(10)_2$ Überlauf

Hinweis:
Ein Überlauf kann bei jeder Basis auftreten (1 + 1 = 0, Übertrag 1).

Binäre Addition
Wenn wir die Regeln aus Tabelle 2.3 auf mehrziffrige Binärzahlen anwenden, erhalten wir als Übertragsrechnung („wie gewohnt vom Dezimalsystem") bei der Addition der binären Zahlen (10101) und (11111):

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

Binäre Subtraktion
Als Beispiel betrachten wir die Subtraktion der Dezimalzahlen 245 und 61.

```
 245
-  61
-----
 184
```

Teilrechnung: 4 – 6 = (14 – 6) – 10 = 8 – 10

Der Teilrechnung entnehmen wir, dass wir subtrahieren, indem wir jeweils die Ziffern mit dem Komplement der darüber liegenden Ziffer addieren bzw. das Komplement der kompletten Zahl nehmen und dann am Schluss 100 subtrahieren. In obigem Beispiel also:

Entweder $4 - 1 = 4 + 9 - 10$
und $4 - 6 = 4 + 4 - 10$

oder $244 - 61 = 244 + 39 - 100$.

Letzteres ist einfacher, vor allem binär, da nur die äußerst links stehende Ziffer um 1 reduziert werden muss. Sollte die Differenz negativ werden, dann wird die Reihenfolge umgekehrt, obiges Verfahren angewandt und am Schluss das Vorzeichen umgedreht. Binär erhalten wir also für die Teilrechnung von 12 – 7:

```
  1100
– 0111
= 0101
```

Fazit: Eine Komplementbildung kann also erforderlich sein. Dazu stellt sich die wichtige Frage, wie wir negative Zahlen darstellen, welche in Abschnitt 2.5 behandelt werden.

Binäre Multiplikation
Es wird der gleiche Algorithmus wie bei der dezimalen Multiplikation verwendet. Beispiel 2.8 zeigt die Multiplikation von 9 und 11.

2.8 BEISPIEL **Binäre Multiplikation**

Der binären Multiplikation von 9 * 11 stellen wir die geläufigere dezimale Rechnung gegenüber.

dezimal:

```
 9 * 11
---------
   90
    9
---------
   99
```

binär:

```
 1001 * 1011
-------------
  1001000
   000000
    10010
     1001
-------------
  1100011
```

kann man auch weglassen!

Wir machen die Probe mit dem Horner-Schema und erhalten die korrekte Dezimalzahl:

$(1100011)_2 = ((((1*2 + 1) * 2 + 0) * 2 + 0) * 2 + 0) * 2 + 1) * 2 + 1$
$= (6 * 2 * 2 * 2 + 1) * 2 + 1 = 99$

Binäre Division
Es wird der gleiche Algorithmus wie bei der dezimalen Division verwendet, nur geht es viel einfacher. Dazu als Beispiel die Division von 299 durch 13 (siehe Beispiel 2.9). Hierbei muss bei jedem Teilschritt geraten werden, welche Zahl mit der zu bestimmenden Ziffer multipliziert, gerade noch unter einem bestimmten Wert liegt. Die Anzahl der Wahlmöglichkeiten beträgt damit 10 (= Anzahl der möglichen Ziffern). Zusätzlich muss dann noch die Multiplikation wirklich durchgeführt und verglichen werden.
Im Binär-Fall ist dies einfacher: Es gibt nur zwei Möglichkeiten für die neue zu bestimmende Ziffer: 0 oder 1. In beiden Fällen muss NICHTS gerechnet werden, denn die Multiplikation mit 0 bzw. 1 kann direkt ohne Überlegen aufgeschrieben werden. Die Entscheidung (Auswahl), welches Ergebnis genommen wird, kann auch sehr schnell durch „Draufsicht" gefällt werden. Obiges dezimales Beispiel finden Sie jetzt als binäre Rechnung, d.h. Sie dividieren jetzt 100101011 durch 1101:

2.9 BEISPIEL **Dezimale und binäre Division**

Der binären Division von 299/13 stellen wir die geläufigere dezimale Rechnung gegenüber.

```
299 : 13 = 23          100101011 : 1101 = 10111
 26                    -1101
 --                    -----
 39                     1011
  3                     - 0
 --                     ----
  0                     10110
                        -1101
                        -----
                        10011
                        - 1101
                        ------
                         1101
                        - 1101
                        ------
                            0
```

Dezimal ↑

Binär →

Probe: Rechnen wir $(10111)_2$ um, erhalten wir $(23)_{10}$.

Spezialfälle

Als Spezialfälle, bei denen sich die Berechnung deutlich vereinfacht, betrachten wir die Multiplikation bzw. Division mit einer Potenz der verwendeten Basis. Hier erhalten wir das Ergebnis durch einen passenden Rechts- bzw. Links-Shift (siehe Beispiel 2.10).

2.10 BEISPIEL **Multiplikation und Division mit Shift**

Multiplikation mit einer Potenz der Basis

Bei dezimaler Darstellung multiplizieren wir 27 mit 100 (= 10^2). Dazu müssen wir nur die Ziffernfolge um 2 Stellen nach links schieben und rechts mit 0 auffüllen:

$27 * 100 = 2700$

D.h.: Es werden einfach „2 Nullen angehängt". Anders formuliert: Wir verschieben das Komma um 2 Stellen nach rechts, d.h. die Zahl nach links (*left shift*).

Jetzt nehmen wir die Zahl $(1011)_2$ und multiplizieren sie mit $2^3 = (1000)_2$. Wir erhalten durch einen left shift um 3 Binärstellen:

$(1011)_2 * (1000)_2 = (1011000)_2$.

Division durch eine Potenz der Basis

Hier geht es analog, d.h. der left shift wird durch einen right shift ersetzt. Als Beispiel dividieren wir dezimal 11 durch 8 binär und erhalten das Ergebnis durch Verschieben der Ziffernfolge um 3 Stellen nach rechts:

$(1011)_2/(1000)_2 = (1.011)_2$

2.5 Darstellung negativer ganzer Zahlen

Bisher haben wir die Betrachtung unabhängig von der Speicherung der Zahlen im Computer gewählt. Jetzt sind die Zahlen nur mit fester Länge darstellbar. Binäre Ziffern entsprechen 1 Bit (binary digit, binary unit). Wir machen jetzt folgende Annahme: Es stehen für die Speicherung von Zahlen die Bits 0,1, ... , m zur Verfügung (z.B.: 32-Bit-Wort: 32 binäre Zeichen, die für eine Zahl verfügbar sind). Dabei sind verschiedene Darstellungsarten denkbar und auch realisiert. Wir erläutern vier verschiedene Arten.

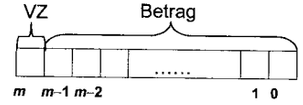
2.5.1 Vorzeichen und Betrag

Dies ist die zuerst nahe liegendste Darstellungsweise. Sie ist angelehnt an unser natürliches Verständnis von einer ganzen Zahl: Sie hat ein Vorzeichen (+ oder –) und einen Wert (Betrag). Das Vorzeichen wird durch das führende Bit dargestellt (0 für + und 1 für –). Damit erhalten wir für die Darstellung der Zahl z die Formel

$$z = \text{sign}(z) * |z|,\ z = (-1)^n * \sum_{i=0}^{N} z_i * 2^i$$

Bild 2.3 zeigt Aufbau und Bedeutung dieser Darstellungsart, Tabelle 2.4 stellt den Zahlwerten z die Bitfolge w gegenüber und Beispiel 2.11 erläutert die Idee mit konkreten Zahlen.

2.3 BILD **Ganze Zahlen: Darstellung mit Vorzeichen und Betrag**



(m + 1) Bits
VZ: 0, bei „+"
1, bei „–"

Hinweis:
Da wir die Binärstellen von 0 bis m nummerieren, ist dies bei der Angabe der insgesamt verwendeten Bits zu berücksichtigen.

2.4 TABELLE **Ganze Zahlen: Darstellung mit Vorzeichen und Betrag**

Zahl z	Bitfolge w
+ 0	0 000 ... 00
+ 1	0 000 ... 01
+ 2	0 000 ... 10
:	:
$+ 2^m - 1$	0 111 ... 11
– 0	1 000 ... 00
– 1	1 000 ... 01
:	:
$- 2^m + 1$	1 111 ... 11

Bemerkung: Die „Null" taucht zweimal auf!

2.11 BEISPIEL **Zahl mit Vorzeichen und Betragsdarstellung**

Die Zahl $z = -13$ soll mit 5 Bit dargestellt werden. Der Zahlenbereich geht von –15 bis +15. z wird dargestellt als 11101.

2.5.2 Exzess

Die Idee ist, intern nur mit nicht negativen Zahlen rechnen zu müssen. Dazu ist die Addition einer Zahl (Exzess q genannt) auf alle Zahlen notwendig, die groß genug ist, damit auch die kleinste negative Zahl nicht negativ wird. Wir könnten jetzt den Exzess wie folgt berechnen:

$$q = |\text{ kleinste negative Zahl }| = |-2^m + 1| = 2^m - 1$$

Damit hätten wir aber wieder zwei Darstellungen für die 0. Dies kann man vermeiden, wenn man für q den Wert 2^m nimmt. Jetzt ist die kleinste Zahl 1. Die 0 nehmen wir jetzt zusätzlich dazu und erhalten die in Tabelle 2.5 abgebildeten Bitfolgen w für die Zahlen z. In Beispiel 2.12 wird eine konkrete Zahl mit Exzess dargestellt.

2.5 TABELLE **Ganze Zahlen: Darstellung mit Exzess**

z	interner Wert	Bitfolge w
-2^m	0	0 0 ... 00
$-2^m + 1$	1	0 0 ... 01
:	:	:
-1	$2^m - 1$	0 1 ... 11
0	2^m	1 0 ... 00
1	$2^m + 1$	1 0 ... 01
:	:	:
$2^m - 1$	$2^{m+1} - 1$	1 1 ... 11

Bemerkung: Die Null taucht jetzt nur einmal auf. Dafür kommt die Zahl -2^m dazu.

2.12 BEISPIEL **Zahl mit Exzess-Darstellung**

Mit $m = 4$ (d.h 4 + 1 Bit stehen zur Verfügung) ergibt sich als Menge der darstellbaren ganzen Zahlen S:

$S = \{-15, -14, \ldots, -1, 0, 1, \ldots, 14, 15\}$

mit $q = 2^4 = 16$: $x \rightarrow x + q$ für $x \in S$

$\rightarrow S^* = \{ \mathbf{0}, 1, \ldots, 14, 15, 16, 17, \ldots, 31\}$

kommt dazu

Wählen wir die Zahl $z = 13$: In 5-Bit-Exzess-Darstellung wird sie mit der Bitfolge 10011 dargestellt.

Hinweis:
Bei arithmetischen Operationen muss der Exzess berücksichtigt werden, da er bei der Konvertierung und Rückkonvertierung je 1-mal berücksichtigt wird. Wenn wir addieren, addieren wir insgesamt 2-mal den Exzess, subtrahieren ihn aber vom Ergebnis nur 1-mal.

In Beispiel 2.13 führen wir eine Addition mit Exzessdarstellung durch.

2.13 BEISPIEL **Addition bei Exzess-Darstellung**

5 + 6 = 11 in Exzess-Darstellung:
5 = 10101
6 = 10110
→ Summe:
101011 = 32+11 = 43
– 16 ./. Exzess von 16 (normale Rückkonvertierung)
– 16 ./. gesonderte Subtraktion des Exzess
= 11

Falls n Zahlen addiert werden, muss der Exzess insgesamt n-mal subtrahiert werden: 1-mal allein durch Rückkonvertierung und $(n-1)$-mal wegen des hier beschriebenen Vorganges.

2.5.3 Komplement-Darstellungen von negativen ganzen Zahlen

Die grundlegende Motivation für eine Komplementdarstellung (egal, welche der beiden Möglichkeiten) ist folgende: Man will bei arithmetischen Operationen mit *einem* Addierwerk alleine auskommen und die übrigen Operationen auf die Addition zurückführen können (siehe Subtraktion in Beispiel 2.14). Der Begriff *Komplement* bezieht sich auf die *logische* Komplementbildung (1er-Komplement) bzw. auf die *arithmetische* Komplementbildung (2er-Komplement). Die Subtraktion kann nämlich i.Allg. durch die Addition des Komplements ersetzt werden. Logisches Komplement ist bitweise zu verstehen, d.h. das Komplement von x = 11001 ist 00110. Addiert man diese beiden Zahlen, so erhält man eine Folge von Einsen: 11111. Arithmetisches Komplement ist wertweise zu verstehen, d.h. Zahl und Komplement ergeben summiert die 2er-Potenz, die mit dieser Anzahl von Bits gerade nicht mehr darstellbar ist. Bei 5 Bit ist das 32. Damit ist das 2er-Komplement von x = 11001 die Zahl 00111, d.h. im Wert genau um 1 höher als das 1er-Komplement von derselben Zahl x. Addiert ergibt sich 100000 als eine Folge von Nullen, denen eine 1 vorangestellt ist. Diese 1 fällt bei der Modulo-Rechnung unter den Tisch. Die rechnerische Durchführung ist bei den beiden möglichen Varianten der Komplementbildung (1er- bzw. 2er-Komplement) ähnlich und wird jetzt genauer formuliert.

Variante 1: 1er-Komplement
Zur Berechnung werden alle Bits von x „gekippt". Als Beispiel nehmen wir $n = 4$, $x = 5 = (0101)_2$ und erhalten damit für das 1er-Komplement von x:

$$\bar{x} = (1010)_2 = (10)_{10}$$

Satz 2.3 zeigt die Formulierung der 1er-Komplementbildung.

Variante 2: 2er-Komplement
Zur Berechnung werden alle Bits von x „gekippt" und 1 hinzuaddiert. Als Beispiel nehmen wir $n = 4$, $x = 5 = (0101)_2$ und erhalten damit für das 2er-Komplement:

$$\begin{array}{rl} \bar{x} = & 0101 \\ & 1010 \quad \text{„Bits kippen"} \\ + & 1 \quad \text{„1 addieren"} \\ \hline & (1011)_2 = (11)_{10} \end{array}$$

Satz 2.4 zeigt die Formulierung der 2er-Komplementbildung.

2.14 BEISPIEL **Komplementbildung bei der Subtraktion**

$x - y = x + \bar{y} - k$ mit $\bar{y} = k - y$,
k = Komplement (d.h. 1er- oder 2er-Komplement), also $k = 2^n - 1$ oder $k = 2^n$.
Wegen der Rechnung „modulo k" gilt: $x - y = x + \bar{y}$

2.5.4 1er-Komplement

Die Negation findet als Ergänzung auf $2^{m+1} - 1$ statt, d.h. es liegt eine komplette Negation (komplette Komplementbildung) vor. Beispiel 2.15 zeigt Additionen im 2er-Komplement.

2.3 SATZ **1er-Komplement**

Falls eine Darstellung mit m +1 Bit gegeben ist (d.h. Stellen von 0 bis m), wird das 1er-Komplement $\bar{z}$ einer ganzen Zahl z wie folgt berechnet:

$$\bar{z} = \begin{cases} |z| \,, & z > 0 \\ 2^{m+1} - 1 - |z| \,, & z < 0 \end{cases}$$

Idee: $z + \bar{z} = 0$ bzw. $(111...11)_2 \Rightarrow |z| + 2^{m+1} - 1 - |z| = 2^{m+1} - 1$

Wir erhalten die in Tabelle 2.6 abgebildeten Bitfolgen w für die Zahlen z.

Oder komplement bei ungerader Zahl.

2.6 TABELLE **1er-Komplementdarstellung**

z	Interner Wert	Bitfolge w
0	0	0 0 ... 00
1	1	0 0 ... 01
:	:	:
$2^m - 1$	$2^m - 1$	0 1 ... 01
$-2^m + 1$	2^m	1 0 ... 00
$-2^m + 2$	$2^m + 1$	1 0 ... 01
:	:	:
–1	$2^{m+1} - 2$	1 1 ... 10
–0	$2^{m+1} - 1$	1 1 ... 11

Bemerkungen:
(1) Das Vorzeichen ist am führenden Bit erkennbar.
(2) Addition und Subtraktion können wie bekannt durchgeführt werden, d.h. ohne irgendwelche zusätzlichen Operationen wie beim Exzess (entsprechend Rechnung modulo $2^{m+1} - 1$).
(3) Bei der Überlauf-Behandlung muss eine „1" dazuaddiert werden, da eine 0 zuviel auftaucht (siehe Bild 2.4, Kreisanordnung).

In Beispiel 2.15 sehen Sie Additionen zweier Zahlen im 1er-Komplement, einmal mit und einmal ohne auftretenden Überlauf.

2.15 BEISPIEL **1er-Komplement: Addition**

Ohne Überlauf:
6 + 6 = 12:

$$\begin{array}{r} 0110 \\ + \; 0110 \\ \hline 1100 \end{array} \rightarrow -3$$

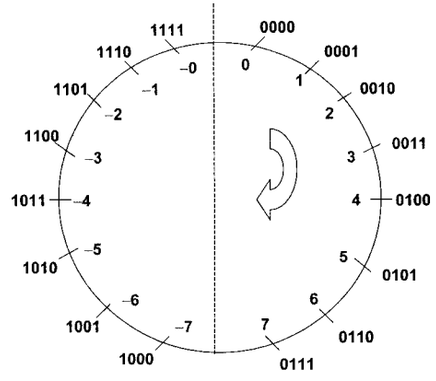
→ –3 = Komplement(15) → 12

Mit Überlauf:
(–4) + (–3) = (–7):

$$\begin{array}{r} 1011 \\ + \; 1100 \\ \hline 1\,0111 \\ + \quad 1 \\ \hline 1\,1000 \end{array} \rightarrow -7$$

Diese Rechnung kann auch mit der Kreisanordnung in Bild 2.4 gerechnet werden.

2.4 BILD **Kreisanordnung für 1er-Komplement für $m = 3$**



2.5.5 2er-Komplement

Die Negation findet als Ergänzung auf 2^{m+1} statt, d.h. es liegt eine komplette Negation (komplette Komplementbildung) vor, siehe auch im Abschnitt 2.5.3 die einführenden Sätze. Beispiel 2.16 zeigt die Darstellung von 13 im 2er-Komplement.

2.16 BEISPIEL **2er-Komplementdarstellung**

Mit $z + \bar{z} = 2^{m+1}$
erhalten wir für $m = 3$ und $z = 13$ (binär: (1101)):

$\bar{z} = 3$ (binär: (0011))

2.4 SATZ **2er-Komplement**

Falls eine Darstellung mit m+1 Bit gegeben ist (d.h. Stellen von 0 bis m), wird das 2er-Komplement $\bar{z}$ einer ganzen Zahl z wie folgt berechnet:

$$\bar{z} = \begin{cases} |z| \,, & z > 0 \\ 2^{m+1} - |z| \,, & z < 0 \end{cases}$$

Idee: $z + \bar{z} = 0 = 2^{m+1}$ (bei modulo-Rechnung) $\Rightarrow$ $|z| + 2^{m+1} - |z| = 2^{m+1}$

Wir erhalten die in Tabelle 2.7 abgebildeten Bitfolgen w für die Zahlen z.

2.7 TABELLE

z	Interner Wert	Bitfolge w
0	0	0 0 ... 00
1	1	0 0 ... 01
:	:	:
$2^m - 1$	$2^m - 1$	0 1 ... 01
-2^m	2^m	1 0 ... 00
$-2^m + 1$	$2^m + 1$	1 0 ... 01
:	:	:
-1	$2^{m+1} - 1$	1 1 ... 11

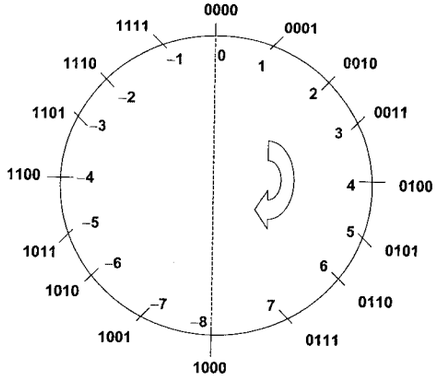
Bemerkungen:
(1) Die Null ist eindeutig dargestellt.
(2) Das Vorzeichen ist am führenden Bit erkennbar.
(3) Addition und Subtraktion können durchgeführt werden wie bekannt, d.h. ohne irgendwelche zusätzlichen Operationen wie beim Exzess (entsprechend Rechnung modulo $2^{m+1} - 1$). Überläufe können ignoriert werden, denn die Null ist eindeutig (tatsächlicher Übertrag: durch Vorzeichenüberprüfung!).

2.17 BEISPIEL **2er-Komplement: Addition**

$$\begin{array}{rr} & 1111 \\ + & 1110 \\ \hline = & 1\,1101 \end{array} \rightarrow \text{korrekt, aber Überlauf, also Ergebnis} = 1101$$

In Beispiel 2.17 sehen Sie die Addition der Zahlen –1 und –2 im 2er-Komplement. Dieses Beispiel kann auch mit der Kreisanordnung in Bild 2.5 gerechnet werden.

2.5 BILD



Ausblick auf die Multiplikation
Bei der Multiplikation von zwei m-stelligen Zahlen ist das Ergebnis eine $2m$-stellige Zahl. Das hat Konsequenzen für die Details im Algorithmus, je nach verwendeter Zahlendarstellung. Zumindest müssen die Faktoren auf $2m$ Stellen erweitert werden (siehe Beispiel 2.18). Positive (negative) Zahlen müssen mit Nullen (bzw. Einsen, je nach Darstellungsart) ergänzt werden.

2.18 BEISPIEL **Multiplikation bei Komplementdarstellung**

Für das Beispiel $6 * (-3)$ benötigen wir zur Zahldarstellung der Eingangsgrößen 3 bzw. 4 Bit, für das Ergebnis aber bereits 6 Bit. Das Ergebnis einer Multiplikation benötigt i. Allg. doppelt so viele Stellen wie die Eingangsgrößen. Dies ist bei der Komplementbildung zu beachten, da jetzt die führenden Nullen eine Rolle spielen.

$6 * (-3) = -18$

→	$6 = (000110)_2$	$18 = (010010)_2$
	$3 = (000011)_2$	$-18 = (101101) + 1$
	$-3 = (111101)_2$	$= (101110)_2$

Bei einer 12-Bit-Darstellung erhalten wir für –3 mit der 2er-Komplement-Darstellung:
$-3 = 2^6 - 3 = 64 - 3 = 61 = (111\ 111\ 111\ 101)_2$
Die Zahl 6 wird „normal" dargestellt:
$6 = (000\ 000\ 000\ 110)_2$
Für das Ergebnis erhalten wir:
$-18 = (111\ 111\ 101\ 110)_2$

2.6 Darstellung reeller und rationaler Zahlen

In der Mathematik werden Zahlen bestimmten Zahlenmengen mit gewissen Eigenschaften zugeordnet. Man beginnt in der Regel mit den natürlichen Zahlen, gefolgt von den ganzen, rationalen und reellen Zahlen. Wenn wir versuchen, diese Zahlen auf dem Rechner darzustellen, bekommen wir genau genommen schon bei den natürlichen Zahlen Probleme: Es gibt unendlich viele natürliche Zahlen, der Rechner verkraftet aber nur endlich viele! Wir müssen also den darstellbaren Bereich einschränken. Aber zumindest können wir die Zahlen, die wir zulassen, exakt darstellen. Dies gilt auch für die ganzen Zahlen.

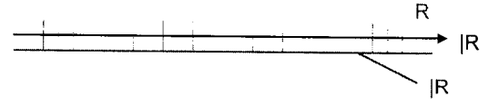
2.6.1 Genauigkeitsbetrachtungen

Anders wird es bei den rationalen und erst recht bei den reellen Zahlen: Im Abschnitt 2.3 (Konvertierung) haben wir bereits feststellen müssen, dass die dezimale Zahl 0,2 binär nicht exakt mit endlich vielen Nachkommastellen darstellbar ist. D.h. wir haben nicht nur eine Einschränkung in der Größenordnung der darstellbaren Zahlen, sondern auch in der Genauigkeit. Nehmen Sie also gleich Abschied von der Vorstellung, dass der Rechner genauer rechnet als Sie! (Ok, bei den meisten Rechenoperationen produzieren die heutigen Rechner brauchbare Ergebnisse, aber vergessen Sie nicht den Raketenabsturz aus dem Prolog!). Die unterschiedliche Darstellung von Zahlen in der Mathematik und auf einem Rechner wird durch eine unterschiedliche Schreibweise der jeweiligen Zahlenmengen gewürdigt:

Mathematik:	$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$
Rechnerdarstellung:	$N \subset Z \subset Q \subset R$
eindeutige Abbildung:	unendliche Zahlenmenge → endliche Zahlenmenge
	Kontinuum → Gleitkommaraster

Dabei steht z.B. R für die Menge der „Real"-Zahlen, die eine echte und endliche Teilmenge der reellen Zahlen ist (siehe auch Bild 2.6 und [Kul1, Kul2, Stet]).

2.6 BILD **Zahlendarstellung auf dem Rechner**



Neben der Einschränkung in der Größenordnung haben wir also eine Einschränkung in der Anzahl der darstellbaren Ziffern einer Zahl. Wir können immer nur eine endliche Teilmenge des reellen Kontinuums in einem Rechner abbilden. Diese Teilmenge nennen wir *Gleitkommaraster*. Bild 2.6 soll dies andeuten. Bei der Computerarithmetik ist also von anderen Zahlenmengen und damit auch von anderen Gesetzmäßigkeiten auszugehen. Z.B. gilt das Assoziativgesetz i.Allg. nicht mehr (siehe Beispiel 2.19):

$(a + b) + c$	=	$a + (b + c)$	„+": reelle Addition
$(a \oplus b) \oplus c$	≠	$a \oplus (b \oplus c)$	„⊕": Addition im Gleitkommaraster (d.h.: „auf Rechner")
	im Allgemeinen		

2.19 BEISPIEL **Assoziativgesetz auf dem Rechner**

Dezimales Beispiel mit 8 verfügbaren Stellen:

(11111113 ⊕ –11111111) ⊕ 7.5111111 =
2.0000000 ⊕ 7.5111111 =
9.5111111

11111113 ⊕ (–11111111 ⊕ 7.5111111) =
11111113 ⊕ –11111103 =
10.000000

Ursache: Auslöschungseffekte

Wir werden auf diese Themen in diesem Buch nicht näher eingehen. Mehr dazu erfahren Sie in [Kul1, Kul2, Stet].

2.6.2 Formate für gebrochene Zahlen

Es gibt zwei Darstellungsformen für gebrochene Zahlen (sowohl in der Mathematik als auch im Rechner), die Festkomma- und die Gleitkommadarstellung (siehe Tabelle 2.8).

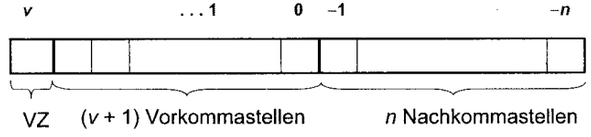
2.8 TABELLE **Fest- und Gleitkommaformate**

	Festkomma	Gleitkomma
Typischer Einsatzbereich	kommerzielle Anwendungen	wissenschaftliche/ technische Anwendungen
Stärke	+, – → exakt	*, /
Schwäche	*, / → unterschiedliche Größenordnungen	+, – → Auslöschungs-effekte

Die *Festkommadarstellung* (fixed point representation) einer ganzen Zahl hat das in Bild 2.7 zu sehende Format. Die Einteilung der zur Verfügung stehenden Stellen zwischen Vor- und Nachkommstellen ist fest vorgegeben. Wir erhalten gemäß Bild 2.7 die Formel

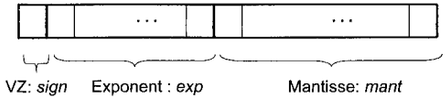
$$z = sign * \left(\sum_{i=-n}^{v} z_i \, b^i \right)$$

2.7 BILD **Festkommaformat**



Die *Gleitkommadarstellung* (floating point representation) einer gebrochenen Zahl trennt die Größenordnung (Exponent exp) und die eigentliche Ziffernfolge (Mantisse mant). Damit erhält man eine von der absoluten Größe unabhängige Darstellungsweise (siehe Bild 2.8). Die Reihenfolge der Komponenten VZ, exp und mant von links nach rechts entspricht der abnehmender Signifikanz dieser Bestandteile für den Zahlenwert. Wir betrachten im Folgenden nur die Gleitkommaformate von gebrochenen Zahlen für die Darstellung im Rechner.

2.8 BILD **Gleitkommaformat**



Damit lässt sich die Gleitkommazahl z wie folgt darstellen:

$$z = sign * \left(\sum_{i=-N}^{M} m_i \, b^i \right) * b^{exp} = sign * mant * b^{exp}$$

Beispiel 2.20 zeigt verschiedene Varianten, ein- und dieselbe Zahl als Gleitkommazahl darzustellen.

Hinweise:
(1) Für die Darstellung von gebrochenen Zahlen wird oft neben der in Deutschland üblichen „Komma"-Schreibweise die „Punkt"-Schreibweise verwendet, wobei dann eine Aufbereitung von großen Zahlen mit einem Punkt nach drei Vorkommastellen durch ein Komma ersetzt wird: Aus 2.500.000,50 € („zwei Millionen fünfhunderttausend Euro und 50 Cent) wird dann 2,500,000.50 €. Welche Schreibweise gerade verwendet wird, geht i.Allg. aus dem Zusammenhang hervor.
(2) Angelehnt an die englischen Begriffe (z.B. floating-point für Gleitkomma) spricht man im Deutschen auch oft vom Gleitpunkt- bzw. Festpunktformat.
(3) Wir wählen in diesem Buch meistens die Darstellung mit Dezimalpunkt.

2.20 BEISPIEL **Gleitkommadarstellungen einer gebrochenen Zahl**

$z = 0.00123 = 1.23 * 10^{-3}$
$= 0.123 * 10^{-2}$
$= 123 * 10^{-5}$
$= 12.3 * 10^{-4}$
usw … → keine eindeutige Darstellung!!!

Sie sehen, dass die Darstellung nicht eindeutig ist. Wir suchen jetzt eine normalisierte eindeutige Darstellung. Dazu gibt es zwei Ansätze:
(a) Immer eine 0 vor dem Komma, erste Nachkommastelle ≠ 0.
→ $z = 0.123 * 10^{-2}$
(b) Eine geltende Ziffer vor dem Komma ≠ 0.
→ $z = 1.23 * 10^{-3}$

Bei Variante (a) gilt für eine eindeutige normalisierte Darstellung:
Basis $b = 10$: $|m| \in [0.1 , 0.999...9] = [0.1 , 1)$
Basis $b = 2$: $|m| \in [0.1 , 1)$ (binär zu lesen!)
d.h.: $|m| = 0.1\underbrace{m_{-2}m_{-3}m_{-4}...m_{-N}}_{\text{Mantisse}}$ mit $m_i \in \{0, 1\}$

Zur Darstellung der Variante (a) auf dem Rechner gilt: Sowohl für den Exponenten als auch für die Mantisse kann man eine der vier möglichen Darstellungsarten für ganze Zahlen (siehe Abschnitt 2.6.1) wählen. Meistens ist dabei folgende Realisierung gegeben:
- der Exponent in Exzess-Darstellung
- die Mantisse mit Vorzeichen und Betrag.

Zur Variante (b) werden wir weiter unten in diesem Abschnitt etwas sagen (IEEE-Standard).

Das Beispiel 2.21 zeigt Ihnen anhand einer konkreten Zahl, wie die Darstellung bei Variante (a) im Rechnerformat aussehen wird.

2.21 BEISPIEL **Gleitkommazahl in interner normalisierter Darstellung**

Es soll die Dezimalzahl $(5.75)_{10}$ als Binärzahl $(z)_2$ dargestellt werden. Gegeben ist dafür das folgende Format: Länge des Exponenten = 5 Bit, Länge der Mantisse = 6 Bit, 1 Bit für das Vorzeichen (= 0 bei +, 1 bei –). D.h.: insgesamt (5 + 6 + 1) Bit = 12 Bit für die Darstellung einer Zahl. In rein mathematischer Notation erhalten wir für z:

$z = (5.75)_{10} = (101.11)_2$

Jetzt müssen Exzess und Mantisse bestimmt werden. Wir erhalten für den Exzess q:

$q = 2^m = 2^{5-1} = 16$

Für den Exponenten *exp* berechnen wir damit:

exp = q + „Anzahl Vorkommastellen (hier: von 5 = $(101)_2$)"
= 16 + 3 = 19 = $(10011)_2$

Die Mantisse *mant* wird aus der normalisierten Zahl z ($z = 0.10111 * 2^3$) gewonnen und an die Mantissenlänge (hier: 6 Bit) angeglichen:

mant = 0.10111 0 ← Ergänzung auf 6 Nachkommastellen

Damit kommen wir zu folgender rechnerinternen Darstellung der Zahl z:

	sign	*exp*					*mant*				
→ z =	0	1	0	0	1	1	1	0	1	1	1	0

Zahlenbereich: Größenordnung und Genauigkeit

Für die größte positive Zahl (*VZ* = 0) suchen wir den größten Exponenten und die größte Mantisse. Das ist jeweils eine Folge von Einsen. Der größte Exponent ist damit bei m Bit für den Exponent: $2^{(m+1)} - 1$, also bei unserem Beispiel mit m = 5 gleich 63. Davon muss noch der Exzess subtrahiert werden. Für die betragsmäßig größte Mantisse gilt bei m Bit in der Mantisse: $1 - 2^{(-m)}$, also hier 1 – 1/64.

Für die kleinste positive Zahl (die also gerade noch größer als null ist) wählen wir den kleinsten Exponenten (=0), subtrahieren den Exzess und wählen die kleinste Mantisse (nur eine 1 an der ersten Stelle, der Rest 0 → muss so sein wegen der Normalisierung, also 1/2). Das ergibt für unser Beispiel: $z = 2^{-q} * 2^{-1} = 2^{-q-1} = 2^{-16}$.

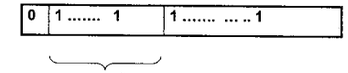
Für den Darstellungsbereich von z gilt:

$\Rightarrow |z| \in [2^{-16}, 2^{15}) \approx [10^{-5}, 10^{4.5})$

Diese Rechnungen sehen Sie in Beispiel 2.22 ausgeführt. Die relative Genauigkeit hängt allein von der Länge der Mantisse ab und beträgt hier bei m = 6: 2^{-6}, d.h. 1/64.

2.22 BEISPIEL **Berechnung des darstellbaren Zahlenbereichs**

Wir gehen aus von einer 12-Bit-Schablone, bei der 1 Bit für das Vorzeichen, 5 Bit für den Exponenten in Exzessdarstellung und 6 Bit für die Mantisse zur Verfügung stehen. Damit erhalten wir als größte darstellbare Zahl:



q –1
= $2^4 - 1 = 15$
→ $b^{15} = 2^{15}$

***mant* (m): *mant* < 1 = 1 – 2^{-m}**

⇒ max = $(1 - 2^{-m}) * 2^{q-1} = (1 - 2^{-6}) * 2^{15}$

Der Darstellungsbereich von gebrochenen Zahlen auf dem Rechner ist also folgendermaßen eingeschränkt:

- Größenordnung: bestimmt durch *Exponent*-Länge
- Genauigkeit: bestimmt durch *Mantissen*-Länge.

Das bedeutet: falls die Gesamtzahl der Bits für eine Gleitkommazahl fest angegeben ist, richtet sich die Aufteilung der Bits für *exp* und *mant* nach der Anforderung (Ist es wichtiger, einen großen Wertebereich zu haben – oder eine größere Genauigkeit?).

IEEE-Formate für die Gleitkommadarstellung

Das *Institute of Electronical and Electric Engineers* (*IEEE*) stellt in vielen technischen Bereichen eine Art Normbehörde dar, an die sich viele Hardware-Anbieter halten. Der IEEE-Standard ist auf vielen Chips, Prozessoren, Co-Prozessoren realisiert, der anders als in unserer Darstellung normalisiert: Die Mantisse wird um eine Stelle verschoben, d.h. das erste signifikante Bit steht *vor* dem Komma. Die Darstellung *VZ – exp – mant* ist identisch. Weiterhin werden Normaufteilungen der Bits auf *VZ*, *exp* und *mant* sowie feste Wortlängen vorgegeben (siehe Tabelle 2.9), was ja auch für den konkreten Entwurf von Prozessoren absolut sinnvoll ist. Aus methodischen Gründen werden wir uns hier aber nicht an den IEEE-Standard halten. Beispiel 2.23 zeigt die IEEE-Normalisierung für die Zahl 5.75.

2.23 BEISPIEL **IEEE-Normalisierung bei Gleitkommazahlen**

$(5.75)_{10} = (101.11)_2 = 1.011 * 2^2$

← Immer eine 1 vor dem Komma

Tabelle 2.9 zeigt übliche Aufteilungen für IEEE-Gleitkommaformate.

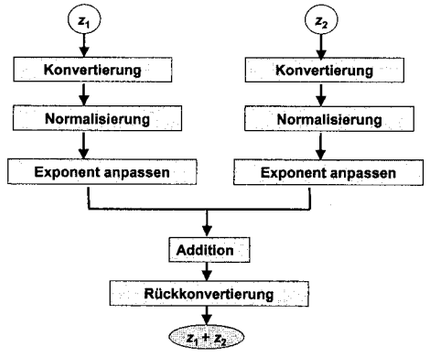
2.9 TABELLE

Wortlänge	VZ	exp	mant	Wertebereich	Genauigkeit
32	1	8	23	$\|z\| \in [2^{-129}, 2^{127}) \approx (10^{-39}, 10^{39})$	$2^{-23} \approx 10^{-7}$
64	1	11	52	$\|z\| \in [2^{-1025}, 2^{1023}) \approx (10^{-307}, 10^{307})$	$2^{-52} \approx 10^{-15}$

2.7 Addition und Subtraktion von Gleitkommazahlen

In diesem Abschnitt wird der Algorithmus für die Addition von Gleitkommazahlen explizit anhand von Beispielen behandelt. Er ist in Bild 2.9 dargestellt.

2.9 BILD **Addition und Subtraktion von Gleitkommazahlen**



Was ist wann zu tun? Bevor es losgeht, wird geklärt, ob gleiche Vorzeichen vorliegen und wenn nicht, welcher Summand betragsmäßig größer ist. Gegebenfalls wird die Reihenfolge der Summanden vertauscht. D.h., bei unterschiedlichen Exponenten wird der kleinere Exponent um die Differenz *d* zum größeren erhöht, die Mantisse wird um *d* Stellen nach rechts geschoben. Gegebenenfalls rutschen dabei Einsen aus der Schablone für die Mantisse raus, d.h. dieser Teil der Mantisse wird abgeschnitten: Ein Rundungsfehler entsteht. Nach der Addition wird gegebenenfalls die Zahl normalisiert und dann rückkonvertiert.

Wir verweisen hier auf umfangreiche Übungen mit dem CBT-Lernprogramm, welches Sie unter *\CBT\Gleitkommarechnung* aufrufen können. In Beispiel 2.24 können Sie die Addition von zwei Gleitkommazahlen mit den rechnerinternen Formaten und Regeln nachvollziehen.

2.24 BEISPIEL **Addition zweier Gleitkommazahlen im Rechnerformat**

Wir wollen die Summe $erg = z_1 + z_2$ berechnen, wobei die Summanden wie folgt gegeben sind:
$z_1 = (-15.2)_{10}$, $z_2 = (-5.75)_{10}$

Als Format haben wir zur Verfügung: 8 Bit für *mant*, 6 Bit für *exp*, 1 Bit für *VZ*.

1. Schritt: Konvertieren
Die beiden Eingangsgrößen werden in das Binärformat gewandelt.

→ $z_1 = (-1111.\overline{0011})_2$
→ $z_2 = (-101.11)_2$

2. Schritt: Normalisieren
Die Binärzahlen werden in das rechnerinterne Gleitkommaformat gewandelt und dazu vorher normalisiert. Für den Exponenten ist der Exzess $q = 2^{6-1} = 2^5 = 32$ zu berücksichtigen.

→ $z_1 = (-0.1111\overline{0011})_2 * 2^4$ → $exp = 32 + 4 = 36$
→ $z_2 = (-0.10111)_2 * 2^3$ → $exp = 32 + 3 = 35$

Anzahl Stellen von z

→ z_1 = | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
VZ exp mant

→ z_2 = | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
VZ exp mant

3. Schritt: Exponenten anpassen
Die Exponenten für die Addition werden angepasst, sofern erforderlich. Hier ist der Exponent von z_2 kleiner als der von z_1. Also muss der von z_2 angepasst werden.

$z_2 = (-0.10111)_2 * 2^3 = (-0.010111)_2 * 2^4$

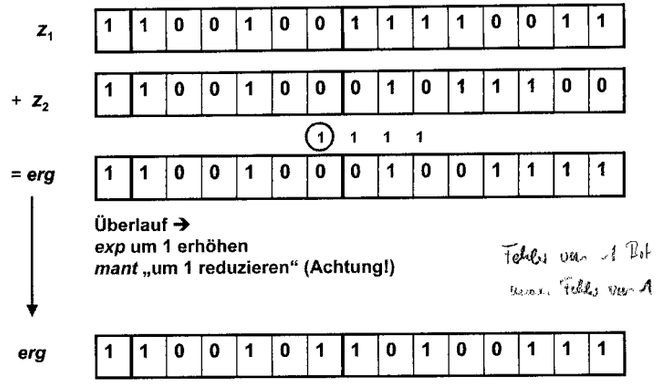
→ z_2 = | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
VZ exp mant
normalisiert
angepasst

| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
VZ exp mant

exp: 1 addieren → shift um 1 Bit

$$(z_1 + z_2 = VZ_1 * mant_1 * b^{exp1} + VZ_2 * mant_2 * b^{exp2} =$$
$$= VZ_1 * mant_1 * b^{exp1} + VZ_2 * mant^*_2 * b^{exp1} =$$
$$= (VZ_1 * mant_1 + VZ_2 * mant^*_2) * b^{exp2})$$

4. Schritt: Addition ausführen
Die Addition erfolgt bei gleichem Vorzeichen so, dass der Exponent übernommen wird (nach der vorherigen Anpassung im Schritt 3) und die Mantissen addiert werden. Falls es dabei einen Übertrag gibt, ist dies mit einer erneuten Normalisierung im nächsten Schritt und passender Exponentenanpassung zu beantworten.



5. Schritt: Normalisieren
Ein Normalisieren ist hier notwendig, da die Ergebnis-Mantisse zuerst einen Übertrag beinhaltet. Dadurch erhöht sich der Exponent um 1. Die Mantisse wird um eine Stelle nach rechts geschoben, wodurch rechts 1 Bit rausfällt. Dies schlägt sich später in der Genauigkeit des Ergebnisses nieder.

mant: 1.01001111 → (/ 2) → 0.10100111 1 **neue Mantisse**

fällt weg → Rundungsfehler

6. Schritt: Rückkonvertierung
Das rechnerinterne Format wird in eine normale mathematische Binärdarstellung zurückgewandelt. Diese wiederum wird dann in eine Dezimalzahl konvertiert. Aus dem Vorzeichen-Bit erhalten wir als Vorzeichen Minus. Der Exponent ergibt sich nach Subtraktion des Exzess zu:
$e - q = e - 2^5 = 37 - 32 = 5$

Die Mantisse *mant* hat den Wert *mant*: $(0.10100111)_2$. Damit erhalten wir als Ergebnis die Zahl
$erg = (-0.10100111)_2 * 2^5 = (-10100.111)_2$
Den Vorkommaanteil, d.h. den ganzzahligen Anteil, können wir direkt umrechnen und erhalten –20. Den Nachkommanteil berechnen wir wie folgt als Dezimalzahl:
$(0.111)_2 = 1/2 + 1/4 + 1/8 = (0.875)_{10}$

Damit ist das Ergebnis dezimal:
$erg = (-20.875)_{10} = -0.20875 * 10^2$

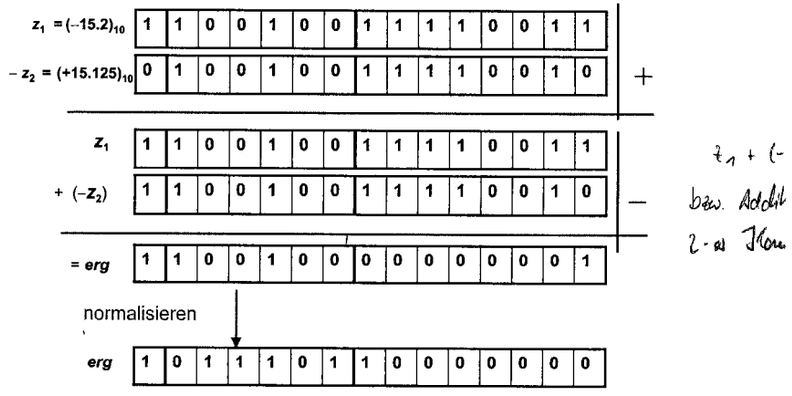
Vergleich mit exaktem Ergebnis
Das exakte Ergebnis ist:
$erg = z_1 + z_2 = -15.2 + (-5.75) = -20.95$
Das berechnete Ergebnis ist:
$erg^* = -20.875$

Damit liegt ein Fehler vor. Die Differenz zwischen „*erg* = exaktes Ergebnis" und „*erg** = maschinell berechnetes Ergebnis" ergibt den absoluten Fehler:
$|erg - erg^*| = \varepsilon = 20.95 - 20.875 = 0.075$
Setzen wir diesen Fehler in Relation zum absoluten Wert, erhalten wir den relativen Fehler:
$\rho = \varepsilon / |erg| = 0.075 / 20.95 \approx 0.0037 \approx 0{,}37\ \%$

Die Subtraktion wird auf die Addition mit Komplementbildung zurückgeführt. Beispiel 2.25 zeigt Ihnen die konkrete Vorgehensweise.

2.25 BEISPIEL **Subtraktion im rechnerinternen Gleitkommaformat**

Hier kürzen wir die einzelnen Schritte etwas ab, da die meisten Schritte analog vorgenommen werden wie im vorigen Additionsbeispiel. Die Subtraktion bedeutet hier Addition zweier Zahlen mit unterschiedlichem Vorzeichen:
$z_1 = (-15.2)_{10}$, $z_2 = (-15.125)_{10}$



Um die Mantisse zu normalisieren, muss ein left shift um 7 Stellen durchgeführt werden. Dazu muss vom Exponent 7 subtrahiert werden, also:
$36 - 7 = 29 = (011101)_2$; *exp*: $e - q = 29 - 32 = -3$

Wir erhalten für das Ergebnis:
$erg = -2^{-3} * (0.1)_2 = -2^{-3} * 2^{-1} = -2^{-4} = (-0.0625)_{10}$

Das exakte Ergebnis ist:
$erg_{exakt} = (-15.2) + 15.125 = (-0.075)_{10}$

Für den absoluten und relativen Fehler ergibt sich:

$\Rightarrow \varepsilon = |0.0625 - (-0.075)| = 0.014$

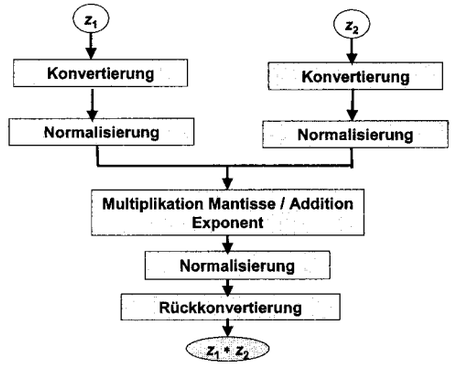
$\rho = \varepsilon / |erg| = 0.015 / 0.075 = 1/5 = 20\ \%$

Der Grund für den Fehler sind Auslöschungseffekte, d.h. die Mantissen sind nahezu identisch. Dadurch erhalten wir viele führende Nullen in der Ergebnis-Mantisse, was zu den Ungenauigkeiten führt.

2.8 Multiplikation und Division von Gleitkommazahlen

Die Multiplikation bzw. Division von Gleitkommazahlen ist einfacher als die Addition bzw. Subtraktion. Folgende Schritte sind notwendig (siehe Bild 2.10 mit Algorithmus).

2.10 BILD **Multiplikation zweier Gleitkommazahlen**



Auch hier gibt es ein konkretes Beispiel 2.26, in welchem wir die wesentlichen Schritte des Algorithmus explizit anhand der Zahlen –15.2 und –15.75 ausführen.

2.26 BEISPIEL **Multiplikationsbeispiel**

Berechnung des Exponenten durch Addition der Einzelexponenten unter Berücksichtigung des Exzesses:

$(-15.2)_{10} * (-5.75)_{10} = ?$

***exp*:** **–15.2 → (Exzess)** **→ 100100**
–5.75 → (Exzess) **→ 100011** +

1 000111
– 100000 Exzess

100111 **$exp = e - q = (111)_2 = 7$**

Berechnung der Mantisse durch Multiplikation der Einzelmantissen:

***mant*:** **–15.2 → m_1 = 0.11110011**
–5.75 → m_2 = 0.10111000

$m_1 * m_2$: **0.11110011 * 0.10111**

1 1 1 1 0 0 1 1
1 1 1 1 1 0 0 1 1
1 1 1 1 1 0 0 1 1
1 1 1 1 1 0 0 1 1
1 1 0 0 0 0 0 0 1 1 1 1 1 5 um 1 nach rechts

0.10101110101 01

mant **abschneiden → Rundungsfehler**

Damit haben wir schon das Ergebnis *erg*:

$erg = +2^7 * (0.10101110)_2 = (1010111)_2 = (87)_{10}$

Ein Vergleich mit dem exakten Ergebnis liefert folgenden Ausgangsfehler:

$erg_{exakt} = 87.405 \rightarrow \varepsilon = 0.405 \rightarrow \rho = 0.405 / 87.405 \approx 0.5\ \%$
(„Ausgangs"fehler)

Da wir aber bereits bei der Konvertierung der Eingangsgrößen einen Fehler machen mussten, lag folgender Eingangsfehler vor:

$z_1 \approx 2^4 * 0.11110011 = 1111.0011 = 15 + 2^{-3} + 2^{-4} = 15\ 3/16 = 15.1875$

„Eingangs"fehler (durch Konvertierung von $z_1 = 15.2$):

$\rightarrow \varepsilon = 15.2 - 15.1875 = 0.0125 \rightarrow \rho = 0.0125 / 15.2 \approx 0.1\ \%$

Der Eingangsfehler hat sich also durch die intern notwendigen Umrechnungen verstärkt.
