Digitale Zahldarstellung und Arithmetik

Hans Jürgen Ohlbach

Keywords: Zahldarstellung: Integer, Gleitkommazahlen, Zweierkomplement, arithmetische Operationen, logische Operation, Shift Operationen, Maskierungstechnik.

Inhaltsverzeichnis

Diese Einführung in die digitale Arithmetik ist in erster Linie für Studierende, die Informatik als Nebenfach studieren.

1 Grundlegende Begriffe

Die elementarste Informationseinheit in heutigen Computern ist das *Bit*. Es steht für 0 oder 1, bzw. falsch oder wahr, oder auch für niedrige und hohe Spannung in einem elektronischen Schaltkreis.

Jeweils 8 Bit werden zu einem *Byte* zusammengefasst. In den meisten Computern ist ein Byte die kleinste Informationseinheit, die er verarbeiten kann. Der Zugriff auf einzelne Bits eines Bytes ist natürlich auch möglich, aber meist aufwendiger als die Verarbeitung eines ganzen Bytes.

Mehrere Bytes werden zu einem *Wort* zusammengefasst. Die Wortgröße hängt allerdings von der Prozessorarchitektur ab. Ältere Prozessoren verarbeiten Wörter der Größe 4 Bytes (= 32 Bits). Neuere Prozessoren verarbeiten Wörter der Größe 8 Bytes (= 64 Bits). Noch höhere Wortgrößen gibt es nur in Spezialprozessoren. Beim Download und der Installation von Programmen wird man oft danach gefragt, ob die 32-Bit Version oder die 64-Bit Version benötigt wird. Um das richtig zu entscheiden, muss man die Prozessorarchitektur seines Rechners kennen.

Ein Wort gibt die Anzahl von Bits an, die ein Prozessor mit einem Maschinenbefehl verarbeiten kann. Um die Gesamtgröße eines Speichers anzugeben, z.B. die Größe des Hauptspeichers oder die Größe eines Programmes oder von Daten, braucht man Vielfache von Bytes. Diese kann man entweder in 10er Potenzen mit *Dezimalpräfixen*, oder in 2er Potenzen mit *Binärpräfixen* angeben. Seit 1998 gilt dafür folgender Standard:

Dezimalpräfixe

Name		**Bedeutung**	
Kilobyte	(kB)	10^3 Byte =	1 000 Byte
Megabyte	(MB)	10^6 Byte =	1 000 000 Byte
Gigabyte	(GB)	10^9 Byte =	1 000 000 000 Byte
Terabyte	(TB)	10^{12} Byte =	1 000 000 000 000 Byte
Petabyte	(PB)	10^{15} Byte =	1 000 000 000 000 000 Byte
Exabyte	(EB)	10^{18} Byte =	1 000 000 000 000 000 000 Byte
Zettabyte	(ZB)	10^{21} Byte =	1 000 000 000 000 000 000 000 Byte
Yottabyte	(YB)	10^{24} Byte =	1 000 000 000 000 000 000 000 000 Byte

Binärpräfixe

IEC-Name		**Bedeutung**	
Kibibyte	(KiB)	2^{10} Byte =	1 024 Byte
Mebibyte	(MiB)	2^{20} Byte =	1 048 576 Byte
Gibibyte	(GiB)	2^{30} Byte =	1 073 741 824 Byte
Tebibyte	(TiB)	2^{40} Byte =	1 099 511 627 776 Byte
Pebibyte	(PiB)	2^{50} Byte =	1 125 899 906 842 624 Byte
Exbibyte	(EiB)	2^{60} Byte =	1 152 921 504 606 846 976 Byte
Zebibyte	(ZiB)	2^{70} Byte =	1 180 591 620 717 411 303 424 Byte
Yobibyte	(YiB)	2^{80} Byte =	1 208 925 819 614 629 174 706 176 Byte

Bei einer heutigen typischen Plattengröße von 1 Terabyte beträgt der Unterschied zwischen der Dezimal- und Binärversion ca. 10%.

2 Ganze Zahlen

In den frühen Tagen der Computerentwicklung gab es Versuche, Computer auf der Basis des vertrauten Zehnersystems zu bauen. Das hat sich aber schnell als ziemlich unpraktikabel herausgestellt. Seither wird ausschließlich das Binärsystem verwendet. In diesem System werden Zahlen als Folgen von Bits (0 oder 1) dargestellt (engl. *Integer*).

2.1 Binärdarstellung von ganzen Zahlen

Wir betrachten zunächst die Binärdarstellung von nicht-negativen ganzen Zahlen. Die Bedeutung und Verwendung von Bitfolgen für diese Zahlen ist ganz analog zum Dezimalsystem. Der Unterschied ist lediglich die Verwendung von Zweierpotenzen anstelle der Zehnerpotenzen. Das folgende Beispiel verdeutlicht die Korrespondenzen:

Zehnersystem	2	3	0	4	0	= 23040
	10^4	10^3	10^2	10^1	10^0	
Binärsystem	1	0	1	1	0	= 22
	2^4	2^3	2^2	2^1	2^0	

Wie man an dem Beispiel sieht, geschieht die Umrechnung vom Binärsystem in das Dezimalsystem, indem man die entsprechenden Zweierpotenzen, wo eine 1 vorkommt, addiert.

Umrechnung vom Dezimalsystem in das Binärsystem: Das Verfahren zur Umrechnung vom Dezimalsystem in das Binärsystem dividiert die Zahl solange durch 2 (mit Rest), bis die 0 entsteht. Die Reste inklusive der 0 als Rest schreibt man von hinten nach vorne auf. Dies ist das Ergebnis. Das folgende Beispiel verdeutlicht die Vorgehensweise: Wir wandeln die Zahl 109 in Binärdarstellung um:

	Reste
109 / 2 = 54 Rest 1	1
54 / 2 = 27 Rest 0	01
27 / 2 = 13 Rest 1	101
13 / 2 = 6 Rest 1	1101
6 / 2 = 3 Rest 0	01101
3 / 2 = 1 Rest 1	101101
1 / 2 = 0 Rest 1	1101101

Die Binärdarstellung von 109 ist also 1101101.

Wenn wir die Zahl zurückwandeln in das Dezimalsystem bekommen wir:
$(1 * 2^6) + (1 * 2^5) + (0 * 2^4) + (1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0) = 109$

2.2 Oktal- und Hexadezimaldarstellung von nicht-negativen ganzen Zahlen

Computer benutzen intern ausschließlich das Binärsystem. Allerdings sind Binärzahlen für den Menschen ziemlich schwer zu lesen und zu interpretieren. Man kann natürlich alle Binärzahlen in das Dezimalsystem umwandeln, um sie für den Menschen lesbarer zu machen. Es gibt jedoch Situationen, wo es wichtig ist, die genaue Bitfolge anzusehen, z.B. wenn Fehler passiert sind. Für diese Fälle haben sich das Oktal- und das Hexadezimalsystem als guter Kompromiss zwischen interner Zahldarstellung und Lesbarkeit für den Menschen herausgestellt.

Das Oktalsystem basiert auf der Basis 8, und das Hexadezimalsystem basiert auf der Basis 16.

Im Oktalsystem gibt es daher Ziffern 0,1,2,3,4,5,6,7. Nach der 7 kommt dann 10.

Im Hexadezimalsystem braucht man 16 Ziffern. Hierfür benutzt man folgende Konvention:

Hexadezimalziffern	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dezimaldarstellung	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Nach dem F kommt dann die 10, mit der Dezimalbedeutung $10_{16} = 16_{10}$.

(Um zu verdeutlichen, in welchem System die Zahlen zu interpretieren sind, kennzeichnet man das System durch einen Index: x_2: Binärsystem, x_8: Oktalsystem, x_{10}: Dezimalsystem, x_{16}: Hexadezimalsystem.)

Die Umwandlung vom Dezimalsystem in das Oktal- oder Hexadezimalsystem kann man ganz analog zur Umwandlung in das Binärsystem machen. Anstelle von der Division durch 2 dividiert man durch 8 bzw. 16, und merkt sich die Reste. Diese Richtung wird jedoch sehr selten benötigt. Viel häufiger braucht man die Umwandlungen zwischen Binärsystem und Oktal- bzw. Hexadezimalsystem. Da beide Systeme auf Zweierpotenzen basieren (Oktal: 2^3, Hexadezimal: 2^4), besteht eine unmittelbare Korrespondenz zwischen dem Binärsystem und dem Oktal- sowie Hexadezimalsystem: Eine Dreiergruppe von Bits steht für die Zahlen 0...7, und eine Vierergruppe von Bits steht für die Zahlen 0...15 (oder mit Hexadezimalziffern: 0...F).

Um eine Binärzahl in eine Oktal- oder Hexadezimal umzuwandeln, geht man daher folgendermaßen vor: Man gruppiert die Bitfolge von rechts nach links in Dreiergruppen (für die Oktaldarstellung) bzw. Vierergruppen (für die Hexadezimaldarstellung), und wandelt die Gruppen separat um. Hierbei hilft folgende Tabelle:

Oktal		Hexadezimal		Dezimal
000	0	0000	0	0
001	1	0001	1	1
010	2	0010	2	2
011	3	0011	3	3
100	4	0100	4	4
101	5	0101	5	5
110	6	0110	6	6
111	7	0111	7	7
		1000	8	8
		1001	9	9
		1010	A	10
		1011	B	11
		1100	C	12
		1101	D	13
		1110	E	14
		1111	F	15

Jetzt könne wir die Bitfolge 1101101 für die Zahl 109 in Oktal- und Hexadezimaldarstellung umwandeln:

Oktal:	1	101	101	
	1	5	5	155
Hexadezimal:		110	1101	
		6	D	6D

Wenn bei den führenden Dreier- bzw. Vierergruppen Ziffern fehlen, werden die mit 0 aufgefüllt.

Zur Kontrolle können wir die Zahlen wieder in das Dezimalsystem transformieren:

$155_8 = 1 \cdot 8^2 + 5 \cdot 8 + 5 = 109_{10}$

$6D_{16} = 6 \cdot 16 + 13 = 109_{10}$

Die Umwandlung von der Oktal- und Hexadezimaldarstellung in die Binärdarstellung funktioniert analog: Man wandelt die Ziffern separat in die entsprechenden Dreier- bzw. Vierergruppen von Bits um, und hängt die einfach aneinander.

Diese Umwandlungen sind deshalb so einfach weil beide Systeme als Basis Zweierpotenzen haben.

2.3 Negative ganze Zahlen

Die naheliegendste Möglichkeit, negative ganze Zahlen binär darzustellen, ist, für das Negationszeichen ein separates Bit zu reservieren. Ist dieses Bit = 1, dann ist die Zahl positiv, ist es 0, dann ist die Zahl negativ.

Aus technischen Gründen hat man aber eine andere Darstellung gewählt, das *Zweierkomplement*. Der Grund war, dass man mit der Zweierkomplement Darstellung Subtraktion von ganzen Zahlen *durch Addition* realisieren kann, so dass man keine Subtraktionshardware in den Prozessoren braucht.

Um zu verdeutlichen, wie das geht, nehmen wir zunächst an, die Zahlen werden nur mit 3 Bit dargestellt.

Dann gilt:

$$\begin{aligned} x - y &= x + 1000 - y - 1000 \\ &= x + (1000 - y) - 1000 \\ &= x + y' - 1000 \end{aligned}$$

Das heißt, um y von x abzuziehen müssen wir
1. $y' = 1000 - y$ berechnen (das ist das Zweierkomplement)
2. $x + y'$ addieren
3. die 1000 abziehen, indem man das linkeste Bit löscht.

Berechnung des Zweierkomplements:
Für Zahlen y in 3-Bit Darstellung ist das Zweierkomplement $1000 - y$. Für Zahlen in 32-Bit Darstellung ist das Zweierkomplement $1\underbrace{00000000000000000000000000000000}_{32Bit} - y$ Bleiben wir aber bei

der 3-Bit Darstellung.

Es ist $1000 - y = 111 - y + 1$. Wie man sich leicht vergewissert, dreht $111 - y$ einfach die Bits in y um. Beispiel: $y = 101$

$$\begin{array}{r} 111 \\ -101 \\ \hline 010 \end{array}$$

Dies gilt für jedes y egal in welcher n-Bit Darstellung. Um das Zweierkomplement zu berechnen, muss man also nur alle Bits umdrehen (das nennt man dann das *Einerkomplement*), und dann noch 1 dazuzählen.

Dafür gibt es sogar noch eine Abkürzung:

Betrachten wir dazu eine Zahl in 8-Bit Darstellung: 0110 1100
Das Einerkomplement davon ist: 10010011 Die Addition von 1 macht aus den letzten 1en eine 0, und addiert zur rechtesten 0 den Übertrag 1. Daraus wird dann 10010100. Vergleichen wir das mit der Ausgangsdarstellung, dann ergibt sich, dass von links alle Bits umgedreht werden, bis auf die rechteste 1. Diese und die nachfolgenden 0en werden nicht umgedreht.

Also zusammengefasst:
Um das **Zweierkomplement** einer Zahl in n-stelliger Binärdarstellung zu berechnen, drehe von links nach rechts alle Bits um, bis auf die rechteste 1. Diese und alle nachfolgenden 0en werden nicht umgedreht.

Aus dem Zweierkomplement einer Zahl kann man mit genau der gleichen Methode wieder die Originalzahl rekonstruieren.

32-Bit und 64-Bit Darstellung von ganzen Zahlen: Positive Zahlen werden in der normalen Binärdarstellung gespeichert, und zwar nur mit 31 bzw. 63 Bits. Das linkeste Bit ist immer 0. Negative Zahlen werden als Zweierkomplement dargestellt. Das Zweierkomplement von 32-Bit Zahlen bzw. 64-Bit Zahlen mit führender 0 hat dann eine führende 1. Daher kann man positive von negativen Zahlen unterscheiden, indem man das führende Bit testet: ist es 0, dann ist die Zahl positiv, ist es 1 dann ist die Zahl negativ.

Die darstellbaren Zahlenbereiche für ganze Zahlen ergeben sich nun zu:

32-Bit Darstellung:	-2.147.483.648	-	2.147.483.647
64-Bit Darstellung:	-2.223.372.036.854.775.808	-	9.223.372.036.854.775.807

In manchen Programmiersprachen, z.B. in C und C++ unterscheidet man jedoch zwischen *signed Integer* und *unsigned Integer*. Unsigned Integer nutzen die vollen 32 bzw. 64 Bit aus, können dann aber keine negativen Zahlen repräsentieren. Bei den positiven Zahlen verdoppelt sich aber der darstellbaren Bereich:

32-Bit Darstellung:	0	-	4.294.967.295
64-Bit Darstellung:	0	-	18.446.744.073.709.551.615

Programme, die nur mit positiven Zahlen arbeiten können dann diesen erweiterten Zahlenbereich nutzen.

Überläufe: Was passiert nun, wenn man zur größten darstellbaren Zahl noch 1 dazu zählt, oder von der kleinsten noch 1 abzieht? Hier gibt es im Prinzip zwei Strategien:

1. Das System generiert eine Fehlermeldung (arithmetischer Überlauf).
2. Es wird ohne Warnung addiert bzw. subtrahiert. Was dann passiert können wir z.B. an einer 4-Bit Darstellung illustrieren:

   Die größte darstellbare positive Zahl in 4-Bit Darstellung ist 0111 ($= 7_{10}$) . Addiert man dazu die 1, ergibt sich 1111. Diese Zahl wird als negative Zahl im Zweierkomplement interpretiert. Das Zweierkomplement davon wiederum ist 0001. Also entspricht 1111 der Zahl -1. Es wurde also $7 + 1 = -1$ gerechnet.

   Die kleinste negative Zahl ist 1000 ($= -8_{10}$). Subtrahiert man davon -1, indem man das Zweierkomplement 1111 addiert, dann ergibt sich theoretisch 10111. Von diesen 5 Bits wird in 4-Bit Darstellung das linkeste Bit ignoriert. Daher ist das tatsächliche Ergebnis 0111 ($= 7_{10}$). Es wurde also $-8 - 1 = 7$ gerechnet.

Solche Überläufe ohne Fehlermeldung sind i.A. ungewollt. Es gibt nur wenige Anwendungen, wo man das tolerieren kann.

Subtraktion mit Zweierkomplement: Die Subtraktion von Binärzahlen in n-Bitdarstellung reduziert sich also auf die Addition mit dem Zweierkomplement des Subtrahenden. Allerdings muss am Schluss dann noch die Zahl $1\underbrace{0\ldots0}_{n-Bits}$ abgezogen werden. Bei einer 32- oder 64-Bit Darstellung hat die Zahl, die abgezogen wird, 33 bzw. 65 Bits. wobei das linkeste Bit 1 ist. Solche zu langen Zahlen werden dann einfach links abgeschnitten. Dies ist äquivalent zur Subtraktion von $1\ \underbrace{0\ldots0}_{32/64\ Bits}$.

Wir illustrieren das der Einfachheit halber an 4-Bit Zahlen und berechnen 5-3.

	0101	0101	
	- 0011	+1101	Zweierkomplement
		10010	
Abgeschnitten:		0010	= 2

Also ist $5 - 3 = 2$, wie erwartet.

Wie steht es jetzt mit $3 - 5$?

0011	0011	
- 0101	+1011	Zweierkomplement
	1101	= -2

Da das Resultat eine negative Zahl ist (erkennbar an der linkesten 1), repräsentiert es das Zweierkomplement direkt, und es muss nichts abgeschnitten werden. Macht man diese Zahl wieder positiv, ergibt sich $0010 = 2_{10}$. Also ist jetzt $3 - 5 = -2$, ebenfalls wie erwartet.

3 Gleitkommazahlen

Für viele Anwendungen, insbesondere im naturwissenschaftlichen Bereich, reichen die 32/64 Bit Darstellungen von ganzen Zahlen nicht aus. Aus diesem Grund hat man eine weitere Zahldarstellung geschaffen, die *Gleitkommazahlen* (engl. *Floating Point Numbers*). Sie entspricht der bekannten wissenschaftlichen Darstellung mit Mantisse und Exponenten. Ein Beispiel ist

$$\underbrace{3.5}_{Mantisse} \cdot 10 \overbrace{^{-15}}^{Exponent}$$

Überträgt man das auf Binärzahlen, dann bekommt man eine Darstellung, die z.B. folgendermaßen aussehen könnte:

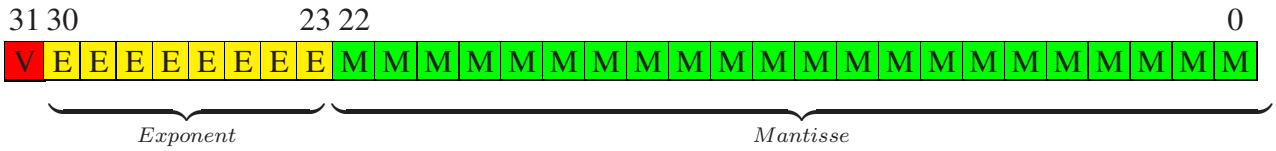
$$1,1101 \cdot 2^{01101}$$

Mantisse und Exponent sind natürlich Zahlen in Binärdarstellung.

Da die Computer mit einer Wortgröße von 32 oder 64 Bit arbeiten, muss man entscheiden, wie man Mantisse und Exponent auf diese Bits verteilt, und wie man negative Mantisse und Exponenten darstellt. Dies wurde durch den internationalen Standard IEEE-754 normiert.

Bevor wir auf die Details eingehen, noch eine Vorbemerkung: In der Exponentialdarstellung kann man die Zahlen immer so normieren, dass in der Mantisse vor dem Komma nur eine Ziffer ist. Z.B. ist $35 = 3,5 \cdot 10^1$ oder $0,035 = 3,5 \cdot 10^{-2}$. In der Dezimaldarstellung steht dann vor dem Komma eine Zahl zwischen 1 und 9. In der Binärdarstellung steht dann immer nur die 1 (es sei denn die ganze Zahl ist lediglich die 0). Wenn da immer nur die 1 steht, ist sie redundant und kann weggelassen werden. Das macht sich die IEEE-Norm zu Nutze. Sie sieht folgendermaßen aus:

32 Bit Floating Point Darstellung (single precision)



64 Bit Floating Point Darstellung (double precision)

V ist das Vorzeichen der Mantisse. Die Mantisse wird in normalisierter Form gespeichert: 1,..., wobei allerdings die 1 vor dem Komma weggelassen wird. Die tatsächliche 0 wird erkannt, wenn alle 32/64 Bits 0 sind. Negative Zahlen werden also nicht in Zweierkomplement dargestellt, sondern das Vorzeichen wird explizit gespeichert.

Der Exponent wird auch nicht in Zweierkomplement Darstellung gespeichert, sondern in sog. Biased Darstellung, wobei für die 32 Bit Version der $Bias = 127$ und bei der 64 Bit Version der $Bias = 1023$ ist.

Der echte Exponent ergibt sich dabei folgendermaßen:
echter Exponent = gespeicherter Exponent - Bias.

Beispiel:
32 Bit: gespeicherter Exponent: 128 $\rightarrow$ echter Exponent = 128-127 = 1
64 Bit: gespeicherter Exponent: 1000 $\rightarrow$ echter Exponent = 1000-1023 = -23

Für manche wissenschaftlichen Anwendungen reicht die Genauigkeit der Floating Point Darstellungen nicht aus. Daher erlaubt der IEEE-754 Standard auch *extended Formate* mit mehr Bits. Diese sind jedoch nicht genau festgelegt, sondern es werden nur Minimalgrößen gefordert:

single extended: ≥ 43 Bit mit ≥ 31 Bit für die Mantisse und ≥ 11 Bit für den Exponenten.

double extended: ≥ 79 Bit mit ≥ 63 Bit für die Mantisse und ≥ 15 Bit für den Exponenten.

Dafür ist jedoch Spezialhardware nötig, die man in den normalen Rechnern üblicherweise nicht hat.

Die Genauigkeit der Zahldarstellung sowie der Umfang der darstellbaren Zahlen ist:

Typ	Dezimalstellen	kleinste Zahl	größte Zahl
single	7...8	$2^{-126} \sim 1,175 \cdot 10^{-38}$	$(1-2^{-24}) \cdot 2^{128} \sim 3,403 \cdot 10^{38}$
double	15...16	$2^{-52} \cdot 2^{-1022} \sim 5 \cdot 10^{-324}$	$(1-2^{-53}) \cdot 2^{1024} \sim 1,798 \cdot 10^{308}$

Einige Bitkombinationen werden jedoch gesondert behandelt:

Sind alle Bits im Exponenten 0, dann bedeutet die Zahl entweder 0, falls die Mantisse auch 0 ist, ansonsten wird die Mantisse für sog. denormalisierte Zahlen verwendet, mit denen man noch kleinere Zahlen darstellen kann.

Sind alle Bits im Exponenten 1, dann bedeutet die Zahl, abhängig von den Mantissenwerten und dem Vorzeichen entweder $+\infty$ oder $-\infty$, oder NaN (Not a Number). Damit kann man auch einigermaßen sinnvoll mit Division durch 0 rechnen.

Die genaue Bitstruktur für die verschiedenen Zahldarstellungen kann man sich mit den folgenden Java Methoden ansehen.

```
public static String toBits(int n) {
    String s = "";
    for(int i = 0; i < 32; ++i) {
        if(i > 0 && i % 8 == 0) {s = "|"+s;}
        if((n&1) == 1) {s = "1"+s;}
        else           {s = "0"+s;}
        n >>= 1;}
    return s;}

public static String toBits(long n) {
```

```
  String s = "";
  for(int i = 0; i < 64; ++i) {
     if(i > 0 && i % 8 == 0) {s = "|"+s;}
     if((n&1) == 1) {s = "1"+s;}
     else           {s = "0"+s;}
     n >>= 1;}
  return s;}

public static String toBits(float n) {
   return toBits(Float.floatToRawIntBits(n));}

public static String toBits(double n) {
   return toBits(Double.doubleToRawLongBits(n)); }
```

Die Programmierung mit Gleitkommazahlen ist nicht ganz unproblematisch. Eigentlich sollten die Gleitkommazahlen ja reelle Zahlen darstellen. Da die Anzahl Bits aber begrenzt ist, muss auf- oder abgerundet werden. Zum Beispiel wird die Zahl 2/3 als float Zahl aufgerundet auf 0.6666667. Bei längeren Ketten von Berechnungen können sich daher Rundungsfehler soweit verstärken, dass das Ergebnis unbrauchbar wird.

Ganz problematisch ist die Subtraktion von nahezu gleich großen Zahlen. Wenn man in Java folgende Berechnung macht:

```
float  a = 2f/3f;
double b = 2d/3d;
double c = a-b;
```

erhält man c = 1.9868214962137642E-8. Das scheint ein sehr kleiner Fehler zu sein. Da der korrekte Wert aber 0 ist, ist der relative Fehler unendlich groß.

Ein wichtiger Bereich der Numerik als Teilgebiet der Mathematik ist es, Algorithmen so zu gestalten, dass solche Rundungsfehler minimiert werden.

Die Arithmetik von Gleitkommazahlen ist kompliziert, und wird auch meist nicht durch spezielle Coprozessoren gemacht. Daher wird in diesem Text nicht weiter darauf eingegangen.

4 Integer Operationen

In diesem Abschnitt betrachten wir wieder ausschließlich die 32/64 Bit Darstellung der ganzen Zahlen.

Addition Die Addition von Binärzahlen kann man auf dem Papier ganz analog zur Addition im Dezimalsystem machen. Man summiert Bit für Bit von rechts nach links und berücksichtigt einen eventuellen Übertrag. Ein Beispiel (mit 16 Bits) ist:

	0011101010101101	
+	0010110110011010	
	0111111101110000	Übertrag
	0110100001000111	

In diesem Beispiel ist das linkeste Bit immer noch = 0. Wenn die Summanden aber noch größer werden, dann kann das linkeste Bit 1 werden. Eine 1 als linkestes Bit bedeutet in der Zweierkomplementdarstellung aber eine negative Zahl. Das Ergebnis wäre also fehlerhaft. Auch hiermit kann man auf zwei Weisen umgehen, entweder man ignoriert es, oder des System meldet einen Fehler (Integer Overflow). Als Programmierer sollte man sich informieren, wie das Programmiersystem, mit dem man arbeitet mit diesen Fehlern umgeht.

Die Additionsalgorithmus, von rechts nach links die Bits Schritt für Schritt mit Übertrag addieren ist allerdings nicht besonders effizient. Da Addition aber eine der wichtigsten Grundoperationen ist, haben moderne Prozessoren eine spezielle Hardware, die diese Additionen parallelisiert und extrem schnell macht.

Subtraktion Durch die Zweierkomplementdarstellung wird die Subtraktion auf die Addition zurückgeführt. Daher benötigen Prozessoren keine spezielle Subtraktionshardware. Allerdings kann es bei der Subtraktion zu einem analogen Phänomen kommen wie bei der Addition. Wenn die Differenz zu klein wird, kann als linkestes Bit eine 0 erscheinen, was als positive Zahl interpretiert wird. Ein Beispiel mit 4 Bits zeigt das Phänomen. Die kleinste negative Zahl mit 4 Bits ist 1000, das ist die Zahl -8. Wenn man davon 1 subtrahiert, d.h im Zweierkomplement 1111 addiert, ergibt sich folgendes:

	1000
+	1111
	10111

Das linkeste Bit würde abgeschnitten werden, so dass das Ergebnis 0111 ist. Die Rechnung ergab damit $-8 - 1 = 7$.

In Java kann man das ausprobieren:

```
System.out.println(Integer.MIN_VALUE-1);
```

druckt die *positive* Zahl 2147483647 aus, ohne irgendeine Fehlermeldung!

Multiplikation: Die Schulmethode zur Multiplikation von Dezimalzahlen lässt sich auch unmittelbar auf die Multiplikation von Binärzahlen anwenden. Dort ist es sogar noch einfacher, da nur Multiplikationen mit 0 oder 1 vorkommt. Die 0 kann man überspringen, und für die 1 muss man die zu multiplizierende Zahl entsprechend verschieben, bevor man sie zum Zwischenergebnis addiert. In der Hardware wird daher die Multiplikation als Folge von Additionen und Verschiebeoperationen realisiert.

Wenn die Zahlen lange gleichwertige Bitketten enthalten, d.h. lange Sequenzen von Nullen oder Einsen, dann hat sich der *Booth-Algorithmus* als überlegen erwiesen, da bei diesem Algorithmus diese Ketten übersprungen werden. Die Details hierzu findet man im Internet.

Die Multiplikation zweier n-Bit Zahlen benötigt i.A. 2n Bits. Da die Prozessoren aber mit einer festen Bitzahl arbeiten, kommt es auch hier zu Überläufen.

Als Beispiel, die größte mit 4 Bits darstellbare positive Zahl ist 0111 ($= 7_{10}$). Die Multiplikation mit 2 ergibt die Zahl 1110 ($= -2_{10}$). In Java kann man das ausprobieren:

```
System.out.println(Integer.MAX_VALUE*2);
```

druckt -2, ohne Fehlermeldung!

Für sehr große Zahlen (in Java gibt es dazu die Datenstruktur BigInteger) verwendet man weiter optimierte Multiplikationsalgorithmen, z.B. den Karazuba-Algorithmus, oder den Schönhage-Strassen-Algorithmus. Dieser wird in Java für Zahlen mit mehr als 74000 Dezimalziffern verwendet.

Division: Die Schulmethode zur Integerdivision lässt sich ebenfalls direkt auf Binärzahlen übertragen. Da hier ebenfalls nur Nullen und Einsen auftreten, läuft das Verfahren auf eine Folge von Subtraktionen und Verschiebeoperationen hinaus. Für die Realisierung in Hardware wurden aber eine Reihe von Optimierungen entwickelt, die die Division stark beschleunigen.

Die Division von zwei Integerzahlen muss keine Integerzahl mehr ergeben. In diesem Fall wird der Bruchanteil einfach abgeschnitten. Z.B. ergibt $5/2 = 2$. Das muss man bei der Programmierung berücksichtigen.

Verschiebeoperationen (engl. shift-Operationen): Einfacher als arithmetische Operationen, aber trotzdem oft sehr nützlich, sind Verschiebeopertionen, bei denen die Bitfolgen nach links oder nach rechts verschoben werden. Auch hier gibt es eine Analogie zum Dezimalsystem. Verschiebt man eine Zahl, z.B. 123 nach links, und füllt rechts mit 0 auf, erhält man 1230. Das entspricht der Multiplikation mit 10. Verschiebt man die Zahl 123 nach rechts, und lässt dabei die rechteste Ziffer fallen, erhält man 12. Das entspricht der Division durch 10, wobei der Rest einfach fallen gelassen wird.

Analog entspricht eine die Verschiebung einer Binärzahl nach links der Multiplikation mit 2, und eine Verschiebung nach rechts einer Division durch 2 (ohne Rest). Durch die feste Bitlänge von 32 oder 64 Bits und die Zweierkomplementdarstellung ist es aber nicht ganz so einfach.

Die Verschiebung einer großen positiven Zahl nach links kann eine 1 an die linkeste Position bringen. Das entspricht dann einer negativen Zahl. Die Wirkung ist jedoch genau wie bei der Multiplikation mit 2.

Bei der Verschiebung nach rechts muss das linkeste Bit aufgefüllt werden. Hier gibt es zwei Möglichkeiten:

logischer Rechtsshift: Hierbei wird das linkeste Bit immer mit 0 aufgefüllt. Aus einer negativen Zahl kann dann eine positive Zahl werden.

arithmetischer Rechtsshift: Wenn das linkeste Bit 0 ist, wird mit 0 aufgefüllt, wenn es 1 ist, wird mit 1 aufgefüllt. Hierbei bleiben negative Zahlen auch negativ.

In Java kann man das folgendermaßen testen:

```
System.out.println(Integer.MIN_VALUE >>  1);
        %arithmetischer Shift: ergibt -1073741824
System.out.println(Integer.MIN_VALUE >>> 1);
        %logischer Shift: ergibt 1073741824
```

Bitweise logische Verknüpfungen: Für einzelne Bits gibt es u.A. die bekannten Booleschen Verknüpfungen und, oder, exclusives oder (xor), und nicht. Diese lassen sich auf Bitfolgen übertragen, indem sie auf zwei Bitfolgen bitweise angewendet werden. In den üblichen Programmiersprachen sind das Operatoren, die man auf die Zahldarstellungen (int, long usw.) anwenden kann.

Beispiele:

	01101101		01101101		01101101		
	10110111		10110111		10110111		10110111
und	00100101	oder	11111111	xor	11011010	nicht	01001000

Die entsprechenden Operatoren in Java können auf int oder long Zahlen angewendet werden.

Dort heißen sie:

Name	Symbol	Beispiel
und	&	$x\&y$
oder	|	$x\|y$
xor	^	$x\hat{}y$
nicht	$\sim$	$\sim x$

Maskierungen: Die logischen Operationen eigenen sich ganz besonders, um Bitfolgen auf bestimmte Muster zu testen, oder um bestimmte Teile einer Bitfolge zu extrahieren. Dazu definiert man eine Integer als sog. *Maske*, die man dann mit andere Integern verknüpft.

Beispiele: Wir illustrieren die Ideen zur Vereinfachung an 16-Bit Zahlen:

Test, ob an der 5. Position von rechts eine 1 steht: Eine Maske, die genau an der 5. Position von rechts eine 1 hat, bekommt man mit dem Ausdruck $1 << 5$ (die 1 wird um 5 Positionen nach links geschoben.) Mit der und-Operation kann man jetzt jede Zahl damit testen:

	00000000 00100000			00000000 00100000	
&	01011010 01101101		&	01011010 01001101	
	00000000 00100000			00000000 00000000	

Der Testausdruck für eine Zahl x würde in Java dann sein ((1<<5) & x) != 0

Extraktion der letzten 8 Bits aus einer Zahl: Dazu benötigt man eine Maske, die in den letzten 8 Bits 1 ist, und davor immer 0. So eine Zahl erhält man z.B. durch $(1 << 8) - 1$ (die 1 um 8 Bits verschieben, ergibt 256, und dann 1 abziehen, ergibt $255_{10} = 0000000011111111_2$. Mit der und-Operation kann man damit jetzt aus einer Zahl die letzten 8 Bit extrahieren:

```
      00000000 11111111
&     10101010 10101010
-------------------------
      00000000 10101010
```

Der Java Ausdruck, um aus der Zahl x die letzten 8 Bits zu extrahieren ist daher
((1<<8)-1) & x

Extraktion der zweitletzten 8 Bits aus einer Zahl:
Dazu braucht man die eine Maske 1111111100000000 und muss man Ende die extrahierten Bits noch nach rechts schieben. Der Java Ausdruck dazu ist dann

((((1<<8)-1)<<8) & x)>>8.

(((1<<8)-1)<<8) bildet die Maske, indem die rechtesten 8 Einsen um 8 Bit nach links verschoben werden, dann wird mit x „verundet“, und das Ergebnis um 8 Bits nach rechts geschoben.

Die letzten 3 Bits einer Zahl auf 0 setzen: Hierzu benötigt man eine Maske, die in den letzten 3 Bits eine 0 hat, und davor immer 1. Das geht folgendermaßen: $\sim ((1 << 3) - 1)$.
(1 << 3)-1 erzeugt rechts 3 Einsen. $\sim$ invertiert alle Bits. „Verundet“ man eine Zahl jetzt mit dieser Maske, dann werden die letzten 3 Bits zu 0, und alle anderen bleiben wie sie waren.

Beispiel:

```
      11111111 11111000
&     10101010 10101110
-------------------------
      10101010 10101000
```