Grundzüge der Informatik II

Kapitel 1: Informationen und Daten

Jun.-Prof. Dr. Melanie Schmidt

Informatik

Universität zu Köln

4. November 2020

Tippfehler, Fragen und Ähnliches

Bei der Neuerstellung von Texten treten immer kleinere Fehler oder Unklarheiten auf. Bite wenden Sie sich bei Anmerkungen per Email an `mschmidt@informatik.uni-koeln.de`.

Inhaltsverzeichnis

Kapitel 1

Informationen und Daten

1.1 Darstellung von Zeichen

Ein *Bit* ist die kleinste Informationseinheit im Computer. Es ist entweder 0 oder 1. In dieser Vorlesung abstrahieren wir davon, wie ein Bit tatsächlich gespeichert oder übertragen wird. Wir interessieren uns aber dafür, wie man mit Bits Informationen darstellen kann. Mit einem Bit lässt sich nur eine binäre Information darstellen. Mit einer Bitfolge mit 2, 4 oder n Bits lassen sich schon $2 \cdot 2 = 4$, $2 \cdot 2 \cdot 2 \cdot 216$ bzw. allgemein 2^n verschiedene Informationen darstellen. Eine Folge von 4 Bits wird Halbbyte oder Nibble genannt, eine Folge von 8 Bits Byte. In der Regel werden Informationen in Bytes oder Vielfachen von Bytes gespeichert; auch die Größe von Dateien wird üblicherweise in Bytes angegeben. Größere Dateigrößen oder Festplattenplatz werden in Megabytes, Gigabytes oder Terabytes angegeben. Dabei ist zu beachten, dass ein Megabyte ursprünglich 1024^2 Bytes bezeichnete, obwohl die Interpretation als 1000 Bytes mit der Verwendung bei physikalischen Einheiten konsistenter wäre. Dies kann zu Missverständnissen führen, z.B. bei der Größenangabe von Festplatten.

Wir kennen Dateien aus dem täglichen Gebrauch und wissen, dass sich inzwischen eine Fülle verschiedener Informationen abspeichern lassen, auch Musikstücke und Filme. Das werden wir im Rahmen dieser Vorlesung nicht besprechen, wohl aber die Grundlagen: Wie werden Texte und Zahlen abgespeichert?

Für die Darstellung von Texten ist die *Kodierung* wesentlich. Es muss für Buchstaben, Satzzeichen, Sonderzeichen und Steuerzeichen festgelegt werden, welche Bitfolge was darstellt. Tabelle 1.1 zeigt die Kodierung des bekannten ASCII-Standards, der 1963 von der American Standards Organization festgelegt wurde. Der Code verwendet 7 Bits. In üblichen Systemen ist daher das erste Bit frei. Es kann zum Beispiel zur Fehlererkennung verwendet werden: Dazu setzt man das erste Bit, das aus diesem Grund auch Parity Bit genannt wird, so, dass das Byte eine ungerade Anzahl von 1-Bits enthält. Damit kann man erkennen, wenn ein Bit fehlerbehaftet ist. Es kann aber auch verwendet werden, um mit 0 und 1 zwischen den ASCII-Zeichen und weiteren 128 Zeichen umzuschalten. So entstehen ASCII-Erweiterungen. Die ISO (International Organization for Standardization) hat verschiedene ASCII-Erweiterungen normiert,

Bits 3210/7654	P000	P001	P010	P011	P100	P101	P110	P111
0000	NULL	DC_0		0	@	P	‘	p
0001	SOM	DC_1	!	1	A	Q	a	q
0010	EOA	DC_2	”	2	B	R	b	r
0011	EOM	DC_3	#	3	C	S	c	s
0100	EOT	DC_4	$	4	D	T	d	t
0101	WRU	ERR	%	5	E	U	e	u
0110	RU	SYNC	&	6	F	V	f	v
0111	BELL	LEM	’	7	G	W	g	w
1000	FE	S_0	(	8	H	X	h	x
1001	HT/SK	S_1	)	9	I	Y	i	y
1010	LF	S_2	*	:	J	Z	j	z
1011	V/TAB	S_3	+	;	K	[	k	
1100	FF	S_4	,	<	L	\	l	ACK
1101	CR	S_5	-	=	M	]	m	UC
1110	SO	S_6	.	>	N	↑	n	ESC
1111	SI	S_7	/	?	O	←	o	DEL

Tabelle 1.1: 7-Bit ASCII Code (American Standard Code for Information Interchange). Die ersten drei Bits legen in obiger Darstellung die Spalte fest, die weiteren vier Bits die Zeile. P steht für ‘Parity Bit’, eine mögliche Verwendung des ‘freien’ ersten Bits.

insbesondere die ASCII-Erweiterung Latin-1 (ISO 8859-1), die unter anderem die in der deutschen Sprache verwendeten Umlaute enthält. Die Interpretation einer Zeichenfolge gelingt nun nur, wenn man dieselbe ASCII-Erweiterung zur Interpretation verwendet, die auch zur Kodierung verwendet wurde. Geht dies schief, sieht man zum Beispiel anstelle von Umlauten andere Zeichen beim Öffnen einer Datei.

Unicode ist ein Standard, der versucht, die relevanten Zeichen der verschiedenen Kulturkreise in dem universellen 16-Bit-Code UCS-2 zusammenzufassen. UCS-2 ist international standardisiert unter der Norm ISO-10646. Ebenso wurde eine 31-Bit-Version (UCS-4) festgelegt.

Weit verbreitet ist heutzutage der Standard UTF-8. Diese Kodierung ist mit der historischen 7-Bit-ASCII-Kodierung kompatibel und hat variable Kodierungslängen: 7-Bit-ASCII-Zeichen werden weiterhin mit 7 Bits kodiert, andere Zeichen mit 2-6 Bytes. Das UTF-8 Kodierungsschema findet sich in den Vorlesungsfolien.

Nach Festlegung des verwendeten Codes für Zeichen werden Texte durch Aneinanderreihung der Codes für die einzelnen Zeichen dargestellt. Dabei ist wichtig, dass eindeutig ist, wie der entstehende lange Bitstring interpretiert wird. Bei der ASCII-Codierung und bei Unicode ist dies durch die feste Länge sichergestellt, bei UTF-8 wurden die Bitfolgen explizit so gewählt, dass eine eindeutige Interpretation möglich ist.

1.2 Darstellung ganzer Zahlen

Positive ganze Zahlen werden im Rechner in der Regel im *Binärsystem* (oder *Dualsystem*) gespeichert. Dieses funktioniert ganz ähnlich zu dem uns aus dem alltäglichen Gebrauch bekannten *Dezimalsystem*, verwendet jedoch eine andere *Basis*, nämlich $b = 2$ statt $b = 10$. Wir schauen uns zunächst ein Beispiel und dann die allgemeine Darstellung an. Zahlen im Binärsystem kennzeichnen wir durch eine tiefgestellte 2 am Ende der Zahl, analog verfahren wir mit anderen Basen. Die Zahl 10011_2 ist also im Binärsystem geschrieben. Das bedeutet, dass die einzelnen Ziffern der Zahl mit einer Potenz von 2 multipliziert werden müssen, um den Zahlenwert zu erhalten. Dabei geht man von hinten nach vorne vor: Die letzte Ziffer wird mit 2^0 multipliziert, die vorletzte mit 2^1 usw.:

$$\begin{aligned} 10011_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 16 + 2 + 1 \\ &= 19_{10} \end{aligned}$$

Auch wenn die Zahlen etwas länger werden als im Dezimalsystem, ist das Dualsystem ähnlich mächtig: Es lassen sich alle positiven ganzen Zahlen als Binärzahl darstellen, und die Darstellungslänge der Zahlen ist beschränkt (logarithmisch im Zahlenwert). Dieses gilt tatsächlich nicht nur für $b = 10$ und $b = 2$, sondern auch allgemein for Basen $b \geq 2$, wie das folgende Lemma zeigt.

Lemma 1. *Sind eine natürliche Zahl $b > 1$ (*Basis*) sowie b Zahlzeichen (*Ziffern*) $z_0, z_1, \ldots, z_{b-1}$ für die natürlichen Zahlen $0 = w(z_0)$, $1 = w(z_1)$, $\ldots$, $b - 1 = w(z_{b-1})$ gegeben, so entsprechen Zeichenketten mit den Zeichen $z_0, z_1, \ldots, z_{b-1}$ genau den natürlichen Zahlen:*

Die Zeichenkette $x_{n-1}x_{n-2}\ldots x_1x_0$ mit $x_i \in \{z_0, z_1, \ldots, z_{b-1}\}$ kodiert die natürliche Zahl

$$\sum_{i=0}^{n-1} w(x_i)b^i$$

und jede natürliche Zahl k kann so mit $O(\log_b k)$ Ziffern kodiert werden.

Beweis. Wird in der Übung besprochen. □

Für Basen, die größer als 10 sind, benötigt man für die Darstellung mehr als die Ziffern $0, \ldots, 9$. Dies ist insbesondere im *Hexadezimalsystem* ($b = 16$) der Fall. Es ist üblich, die fehlenden Ziffern für $10, 11, \ldots, 15$ mit den Großbuchstaben $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}$ darzustellen.

Zahlensystem	Basis b	Ziffern
Dezimalsystem	10	$0, 1, 2, \ldots, 9$
Dualsystem	2	$0, 1$
Oktalsystem	8	$0, 1, 2, \ldots, 7$
Hexadezimalsystem	16	$0, 1, 2, \ldots, 9, \mathrm{A}, \ldots, \mathrm{F}$

Es ist nützlich, sich zu überlegen, wie zwischen verschiedenen Basen umgerechnet werden kann, d.h. wie wir eine Zahl, die zur Basis b dargestellt ist, in eine Zahl zur Basis

b' umwandeln können. Wie das geht, kann man zum Beispiel an dem Beweis von Lemma 1 ablesen. Wir beschränken uns hier auf ein Beispiel (siehe auch Vorlesungsfolien für Beispiele): Wollen wir 437_{10}, also 437 interpretiert im Dezimalsystem, umrechnen in das Hexadezimalsystem ($b = 16$), so rechnen wir:

$$\begin{array}{rcrcrcl} 437 & : & 16 & = & 27 & \text{Rest} & 5 \\ 27 & : & 16 & = & 1 & \text{Rest} & 11 \\ 1 & : & 16 & = & 0 & \text{Rest} & 1 \end{array}$$

und erhalten das Ergebnis $1B5_{16}$.

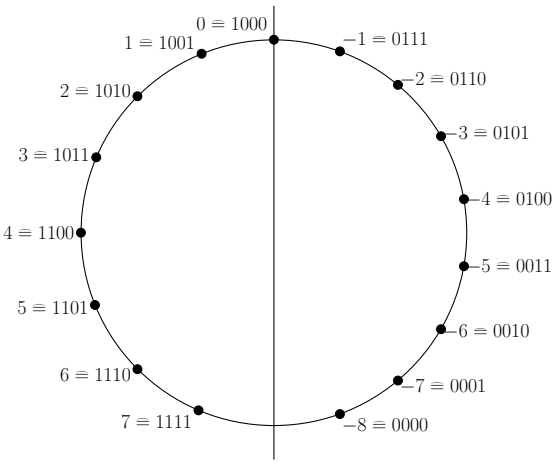
Die Darstellung nicht-negativer ganzer Zahlen durch Bitstrings / im Dualsystem ist der allgemeine Konsens in der Rechnerarchitektur. Unterschiedlich ist lediglich die Anzahl der Bits, die für die Darstellung zur Verfügung stehen. Diese beschränkt den darstellbaren Zahlenbereich.

Beobachtung 2. *Werden n Bits (also $b = 2$) so wie in Lemma 1 aufgezeigt zur Darstellung positiver ganzer Zahlen verwendet, dann sind genau die Zahlen $\{0, 1 \ldots, 2^n - 1\}$ darstellbar.*

Bei *negativen* Zahlen bzw. ganzen Zahlen ohne Einschränkung des Vorzeichens gibt es verschiedene Möglichkeiten der Darstellung, die in verschiedenen Situationen zum Einsatz kommen können.

Die erste Darstellungsform ist die *Vorzeichenbetragsdarstellung*. Dabei wird der Betrag der Zahl, der ja eine positive ganze Zahl ist, in $n - 1$ Bits abgespeichert, und das Vorzeichen in einem separaten Bit. Es sind die Zahlen $\{-(2^{n-1} - 1), \ldots, 2^{n-1} - 1\}$ darstellbar. Dabei hat die 0 zwei Darstellungen als $+0$ und -0.

Eine weitere Möglichkeit ist die *Excess A* - Darstellung. Dabei ist $A \geq 0$ eine Konstante (eine ganze Zahl); verbreitet ist z.B. $A = 2^{n-1}$, d.h. die Excess 2^{n-1} - Darstellung. Die Idee der Excess A - Darstellung ist es, statt x grundsätzlich $x + A$ abzuspeichern. Dadurch werden auch einige negative Zahlen darstellbar: Der darstellbare Zahlenbereich verschiebt sich z.B. im Fall von $b = 2$ und $A = 2^{n-1}$ auf $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$. Die Null hat hier nur eine Darstellung, daher kann insgesamt eine Zahl mehr dargestellt werden als in der Vorzeichenbetragsdarstellung. Für $b = 2$ und $n = 8$ sind die darstellbaren Zahlen in Excess 2^3 Darstellung in folgendem Zahlenring dargestellt.

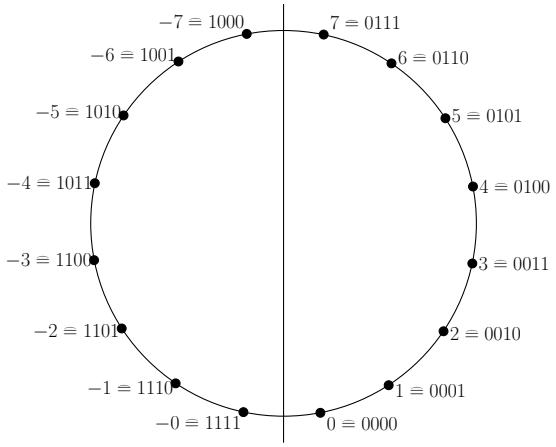


Vorteilhaft ist an der Excess Darstellung auch, dass die Ordnung der Zahlen erhalten bleibt: ist $x < y$, so ist auch $x + A < y + A$. Allerdings sind arithmetische Operationen mit Zahlen in Excessdarstellung eher schwierig.

Die dritte Möglichkeit besteht darin, eine *Komplementdarstellung* zu verwenden. Hierauf wollen wir besonders detailliert eingehen, da es sich um die am weitesten verbreitete Methode handelt, ganze Zahlen darzustellen. Die grundlegende Idee besteht darin, für negative Zahlen $x = -|x|$ grundsätzlich $C - |x|$ abzuspeichern. Dabei ist C eine große Konstante (eine ganze Zahl). Wir nennen $\bar{x} := C - |x|$ im folgenden das *Komplement* von x (das hängt natürlich von C ab und wird später noch genauer unterschieden). Positive Zahlen werden ohne Komplementbildung abgespeichert.

Die Motivation für Komplementdarstellungen ist, dass eine Subtraktion $x - y$ durch die Addition mit dem Komplement von y realisiert werden kann: Wir berechnen $x + \bar{y}$ und erhalten $x - y + C$. Ist $x - y$ negativ, ist so ist $|x - y| = y - x$ und damit auch $x - y + C = C - (y - x) = C - |x - y| = \overline{x - y}$ die Komplementdarstellung von $x - y$. Ist $x - y$ positiv, muss C noch abgezogen werden. Dies ist aber bei geschickter Wahl von C einfach zu bewerkstelligen (wir sehen dies im Folgenden).

Die erste konkrete Ausgestaltung der Komplementdarstellung, die wir betrachten wollen, ist das *Einerkomplement* bzw. allgemein das $(b-1)$-Komplement. Dabei wird $C = 2^n - 1$ bzw. allgemein $C = b^n - 1$ gesetzt. Für $b = 2$ (Einerkomplement) und $n = 4$ erhalten wir damit die folgenden Darstellungen:



Betrachtet man den Zahlenring, so scheint sich eine Regelmäßigkeit abzuzeichnen: Für eine positive Zahl x sind die Darstellungen von x und $-x$ gerade invertiert. Dies ist kein Zufall:

Beobachtung 3. *Sei $b > 1$ und n die Wortlänge. Das $(b-1)$-Komplement einer positiven Zahl $x = \sum_{i=0}^{n-1} x_i b^i > 0$ ist definiert als $\bar{x} = b^n - 1 - x$. Wir beobachten:*

$$\begin{aligned}\overline{x} &= (b^n - 1) - \sum_{i=0}^{n-1} x_i b^i \\ &= \sum_{i=0}^{n-1} b^{i+1} - \sum_{i=0}^{n-1} b^i - \sum_{i=0}^{n-1} x_i b^i\end{aligned}$$

$$= \sum_{i=0}^{n-1} ((b-1) - x_i) b^i$$

Die Darstellung von $b^n - 1 - x$ ergibt sich also aus der Darstellung von $x > 0$, indem jede Stelle x_i durch $b-1-x_i$ ersetzt wird. Diesen Vorgang bezeichnen wir als Bildung des Stellenkomplements.

Für $b = 2$ bedeutet Beobachtung 3, dass man das Einerkomplement durch 'Flippen' aller Bits erhält. Dies ist eine besonders einfache Operation, was einen Vorteil des Einerkomplements darstellt. Beispiele für das Einerkomplement haben wir in obigem Zahlenring bereits gesehen. Für $b = 10$ wird das $(b-1)$-Komplement auch *Neunerkomplement* genannt; ein Beispiel hierfür ist $\overline{0815}_{10} = 9184_{10}$. Wird also 9184_{10} als Neunerkomplementzahl interpretiert, so ist die dargestellte Zahl -815.

Für die Angabe des Bereichs darstellbarer Zahlen müssen wir noch eine Besonderheit überlegen. Es ist zunächst einmal nicht klar, dass $x > 0$ und $\bar{x}$ im $(b-1)$-Komplement tatsächlich *verschiedene* Darstellungen haben. So ist nach obiger Definition zum Beispiel $\overline{11}_3 = 11_3$, da $b - 1 - 1$ für $b = 3$ gerade wieder 1 ergibt. Wir schränken uns daher auf Basen b ein, die gerade sind. Ist b eine gerade Zahl und die erste Ziffer der Darstellung von x höchstens $b/2-1$, so ist sichergestellt, dass das $(b-1)$-Komplement tatsächlich eine andere Darstellung hat als x. Die erste Ziffer des Komplements liegt dann im Bereich von $b - 1 - (b/2 - 1) = b/2$ bis $b - 1 - 0 = b - 1$. Der darstellbare Bereich ist dann im Allgemeinen $[-(\frac{b^n}{2} - 1), (\frac{b^n}{2} - 1)]$ und speziell für $b = 2$ ist der darstellbare Bereich $[-(2^{n-1} - 1), (2^{n-1} - 1)]$.

Wir müssen nun noch klären, wie Zahlen im $(b-1)$-Komplement addiert und subtrahiert werden können. Dabei interpretieren wir die Subtraktion als Addition des Komplements und müssen daher nur die Addition betrachten, allerdings für verschiedene Fälle. Wir beobachten zunächst, dass die Addition positiver Zahlen unproblematisch ist, da diese in Binärdarstellung vorliegen und wie üblich addiert werden können (natürlich nur innerhalb des darstellbaren Zahlenbereichs). Sei nun ein Summand positiv und der andere Summand negativ, d.h. er liegt in Komplementdarstellung vor. Wir betrachten also eine Summe $x + \bar{y}$ für $x, y > 0$. Nun gibt es zwei Fälle. Gilt $x \leq y$, so ist $x - y$ negativ und das Ergebnis muss in Komplementdarstellung gespeichert werden. Dies passiert automatisch:

$$x + \bar{y} = x + C - y = C - (y - x) = C - |x - y| = \overline{x - y}.$$

Durch das C tritt auch während der Rechnung kein Überlauf auf, da $x + \bar{y} \leq C = b^n - 1$ gilt. Dieser Fall ist also völlig unproblematisch. Ist allerdings $x > y$, so ist $x - y \geq 1$ positiv. Die Addition von x und $C - y$ würde jedoch $x - y + C$ ergeben. Dies ist aber gar nicht, was tatsächlich ausgerechnet wird! Die Summe $x - y + C$ erzeugt nämlich einen Überlauf:

$$\begin{aligned} x + \overline{y} &= x + (C - y) \\ &= x + (b^n - 1 - y) \\ &= \underbrace{x - y}_{\geq 1} - 1 + b^n \qquad\qquad \geq b^n \end{aligned}$$

Wird das bei dem Überlauf entstehende $(n+1)$. Bit einfach ignoriert, so entspricht dies einer *Subtraktion* von b^n. Der Wert der gespeicherten Summe ist dann also $x-y+b^n-1-b^n = x-y-1$. Um den korrekten Wert zu erhalten, muss noch 1 addiert werden. Man spricht hier auch von einem 'Einserrücklauf'. Hier ein Beispiel im Dezimalsystem (für ein weiteres Beispiel im Dualsystem siehe Vorlesungsfolien).

$$\begin{array}{r} 65 \\ -\ 43 \\ \hline 22 \end{array} \qquad \begin{array}{r} 65 \\ +\ 56 \\ \hline 121 \\ 1 \\ \hline 22 \end{array} \qquad \text{Einserrücklauf}$$

Der verbleibende Fall, dass beide Summanden negativ sind, verhält sich ganz ähnlich. Hier ist auch das Ergebnis negativ, das Ergebnis der Addition ist jedoch:

$$(C-x)+(C-y) = C-(x+y)+C.$$

Auch hier tritt ein Überlauf auf, der durch Einserrücklauf korrigiert werden kann.

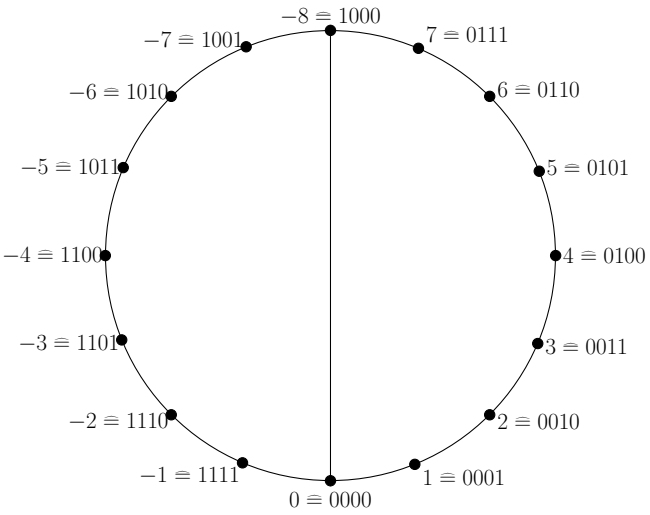
Durch das $(b-1)$-Komplement ist es möglich, Addition und Subtraktion zu vereinheitlichen. Die Addition ist dabei fast einfach, beinhaltet aber die Behandlung des Einserrücklaufs. Wir betrachten nun abschließend das *Zweierkomplement* bzw. allgemein *b-Komplement*, was diesen Nachteil nicht hat. Der einzige Unterschied zum $(b-1)$-Komplement besteht darin, dass $C = b^n$ gesetzt wird. Wir erhalten analog zu Beobachtung 3:

Beobachtung 4. *Sei $b > 1$ und n die Wortlänge. Das b-Komplement einer positiven Zahl $x = \sum_{i=0}^{n-1} x_i b^i > 0$ ist definiert als $\bar{x} = b^n - x$. Wir beobachten:*

$$\begin{aligned} \overline{x} &= b^n - \sum_{i=0}^{n-1} x_i b^i \\ &= (b^n - 1) - \sum_{i=0}^{n-1} x_i b^i + 1 \\ &= \sum_{i=0}^{n-1} ((b-1) - x_i) b^i + 1 \end{aligned}$$

Die Darstellung von $b^n - x$ ergibt sich also aus der Darstellung von $x > 0$, indem jede Stelle x_i durch $b-1-x_i$ ersetzt wird und im Anschluss 1 addiert wird.

Für die Bildung des b-Komplements müssen wir also das Stellenkomplement bilden und dann zu der entstandenen Zahl noch 1 hinzuaddieren. So ist also nun $\overline{0815}_{10} = 9185_{10}$ und $\overline{0101}_2 = 1011_2$. Für $b = 2$ und $n = 4$ erhalten wir:



Man kann sich nun überlegen, dass jedes Wort aus n Ziffern eindeutig interpretiert werden kann und erhält so für die b-Komplementdarstellung einen Zahlenbereich von $[\lceil -\frac{b^n}{2} \rceil, \lceil \frac{b^n}{2} \rceil - 1]$. Für $b = 2$ erhalten wir also als darstellbaren Bereich $[-2^{n-1}, 2^{n-1} - 1]$.

Führen wir nun analoge Überlegungen zur Addition zweier Zahlen aus wie wir dies für das $(b-1)$-Komplement getan haben, fällt auf, dass der Einserrücklauf nicht länger notwendig ist: Das Ignorieren eines Überlaufs in das $(n+1)$. Bit entspricht gerade der Subtraktion von $C = b^n$. Beispiele hierzu finden sich in den Vorlesungsfolien. Ebenso verweisen wir an dieser Stelle für Ausführungen zu Multiplikation auf die Vorlesungsfolien.

1.3 Darstellung reeller / rationaler Zahlen

Wir erinnern uns, dass jede positive ganze Zahl k bei Verwendung der Basis b mit $\mathcal{O}(\log_b k)$ Ziffern dargestellt werden kann, wobei die Ziffern als Faktoren von Potenzen von b interpretiert werden. Für nicht ganzzahlige Werte gibt es im Allgemeinen keine endliche Darstellung als Summe von Potenzen einer gewählten Basis b. Das wird schon deutlich, wenn man sich überlegt, dass $1/3$ im Dezimalsystem keine endliche Darstellung hat. Es ist allerdings sehr wohl möglich, jede reelle Zahl zur Basis b darzustellen, wenn man eine potentiell abzählbar unendliche Darstellungslänge in Kauf nimmt:

Lemma 5. *Für eine ganzzahlige Basis $b > 1$ kann jede Zahl $x \in \mathbb{R}$ mit $x > 0$ dargestellt werden als*

$$\sum_{i=-\infty}^{a} d_i b^i$$

wobei $d_i \in \{0, \ldots, b-1\}$ für alle $i \leq a$ und $a \in \mathbb{N}$.

Bevor wir uns damit beschäftigen, in Rechnerarchitekturen mit nicht ganzzahligen Zahlen umgegangen wird, sei noch das Folgende angemerkt. Wenn wir im allgemeinen Gebrauch auf Zahlen mit einer nicht endlichen Darstellung als Kommazahl stoßen, so

umgehen wird dies häufig, indem wir eine andere Darstellung wählen: Wir schreiben $1/3$ oder $\sqrt{2}$ oder auch π. Mit diesen Darstellungen können wir hervorragend rechnen, und natürlich ist es auch möglich, Software zu entwickeln, die ähnlich vorgeht. Ist es tatsächlich notwendig, exakte Arithmetik zu verwenden (auch wenn dies vielleicht ein bißchen länger dauert oder etwas mehr Speicher benötigt), so kann man zum Beispiel in C++ die *GNU Multiple Precision Arithmetic Library* (GMP) verwenden.

Umwandlung zwischen verschiedenen Basen

Die Umwandlung zwischen verschiedenen Basen funktioniert für rationale Zahlen ganz ähnlich wie für ganze Zahlen. Voraussetzung ist, dass wir eine Zahl mit endlicher Darstellung in der Ausgangsbasis wählen, und dass diese auch in der Zielbasis endlich darstellbar ist. Als Beispiel betrachten wir die Umwandlung der Zahl 6.375_{10} in eine Darstellung zur Basis $b = 2$. Zunächst wandeln wir den ganzzahligen Anteil wie gewohnt um, indem wir wiederholt Divisionen mit Rest durchführen:

$$\begin{array}{rcrcl} 6 & : & 2 & = & 3\ R\ \boxed{0} \\ 3 & : & 2 & = & 1\ R\ \boxed{1} \\ 1 & : & 2 & = & 0\ R\ \boxed{1} \end{array} \quad \uparrow$$

Der Pfeil zeigt an, in welcher Reihenfolge wir die Reste aneinanderhängen müssen, um die Darstellung des ganzzahligen Anteils im Zweiersystem zu erhalten. 6_{10} entspricht also 110_2. Für die Zahlen hinter dem Komma gehen wir folgendermaßen vor: Die aktuelle Zahl (eine Zahl zwischen 0 und 1) wird mit $b = 2$ multipliziert. Wir merken uns, ob das Ergebnis kleiner oder größer als 1 ist. Dann wird die aktuelle Zahl durch den Nachkommaanteil des Ergebnisses ersetzt. Die Berechnung endet, wenn der Nachkommaanteil Null ist:

$$\begin{array}{rcrcl} 0.375 & \cdot & 2 & = & \boxed{0}.75 \\ 0.75 & \cdot & 2 & = & \boxed{1}.5 \\ 0.5 & \cdot & 2 & = & \boxed{1}.0 \end{array} \quad \downarrow$$

Auch hier zeigt der Pfeil an, in welcher Reihenfolge wir die Bits notieren müssen, um die Darstellung um Zweiersystem zu erhalten. Wir erhalten insgesamt also, dass 6.375_{10} der Zahl 110.011_2 entspricht.

Wenn unsere Zahl in der Zielbasis nicht endlich darstellbar ist, dann bricht der beschriebene Algorithmus nie ab. Führt man den Algorithmus zum Beispiel für die Eingabezahl 0.1_{10} aus, so gerät der Algorithmus nach wenigen Schritten in eine Endlosschleife.

Nun wenden wir uns den Möglichkeiten zu, die Darstellung aus Lemma 5 in einem endlichen Format möglichst sinnvoll abzubilden. Wir werden uns dabei an dem sehr lesenswerten Artikel [Gol91] von David Goldberg orientieren. Dieser ist zum weiteren Studium eindeutig empfehlenswert, da wir im Rahmen der Vorlesung nur Auszüge behandeln können.

Festkommazahlen

Bei einer endlichen Darstellung können wir nicht alle Information erhalten. Wir müssen auswählen, wie viel und welchen Teil der Zahl wir abspeichern wollen und müssen dies auf eine Weise tun, die uns auch ermöglicht, die Zahl später wieder korrekt zu interpretieren. Eine Möglichkeit besteht darin, einen bestimmten Bereich der Nachkommastellen festzulegen und die Darstellung auf diesen zu beschränken. Dadurch erhält man *Festkommazahlen*. Wir schauen uns als fortlaufendes Beispiel die (fast) willkürlich gewählten Zahlen $1/7717 = 0.000129584035246\ldots$ und $200000/19 = 10526.3157894\ldots$ an und stellen uns kurz vor, wir hätten uns überlegt, dass wir $b = 10$ wählen und pro Zahl zehn Ziffern abspeichern möchten, von denen fünf für die Nachkommastellen vorgesehen sind. Dann erhalten wir die Festkommadarstellungen 0.00013 und 10526.31579 (wenn wir auf die nächste darstellbare Zahl runden). Wir bemerken, dass der Rundungsfehler für $1/7717$ relativ gesehen zu der Zahl selbst deutlich größer ist.

Gleitkommazahlen

Eine häufig genauere Darstellung bieten *Gleitkommazahlen*. Zur Illustration betrachten wir die Zahl 0.00000000000000000003 im Dezimalsystem. Wir benötigen 20 Nachkommastellen, um diese Zahl präzise darzustellen. Wir können diese Zahl aber auch viel kompakter als $3 \cdot 10^{20}$ darstellen. Genauso können wir $1/7717$ bei einer Verwendung von 10 Stellen insgesamt durch $12958.40352 \cdot 10^{-8}$ sehr viel genauer darstellen als durch 0.0013. Hier schummeln wir allerdings etwas, denn nun müssen wir ja auch 10^{-8} irgendwie abspeichern. Wir bemerken aber, dass wir uns dafür nur -8 merken müssen, wenn wir von vorn herein festlegen, dass wir die abgespeicherte Zahl mit einem Faktor 10^e multiplizieren wollen. Dies ist die Idee der Gleitkommazahldarstellung. Ein Format für Gleitkommazahlen legt zunächst eine ganze Zahl $b \geq 1$ als Basis fest. Darstellbar sind nun Zahlen der Form

$$x = m \cdot b^e,$$

wobei m und e ganze Zahlen sind. Nur m und e werden tatsächlich abgespeichert. Wie m und e abgespeichert werden, ist wiederum eine Designentscheidung beim Entwurf eines Formats für Gleitkommazahlen. Hier kann jeweils eine der Darstellungen aus Kapitel 1.2 gewählt werden. Der weit verbreitete IEEE Standard 754 speichert m in Vorzeichenbetragsdarstellung ab und e mit einer Excessdarstellung. [Achtung: Theoretisch kann man für alle Zahlen verschiedene Formate und sogar andere Basen wählen. Wir könnten zum Beispiel m zur Basis 2 im Einerkomplement darstellen, e zur Basis 8 in Excessdarstellung, und dann auch noch $b = 16$ wählen. In unseren Beispielen vermeiden wir dies natürlich.]

Allein durch die Festlegung von b und den Formaten für m und e wird die Darstellung einer Zahl in einem Gleitkommaformat nicht eindeutig. In unseren fortlaufenden Beispielen in diesem Kapitel verwenden wir $b = 10$, und wir stellen auch alle Zahlen zur Basis 10 (in Vorzeichenbetragsdarstellung) dar. Dann haben wir aber immer noch die Wahl, $1/7717$ als $12958.40352 \cdot 10^{-8}$ abzuspeichern, als $0.1295840352 \cdot 10^{-3}$, als $1295840352 \cdot 10^{-13}$ oder vielleicht als $1.295840352 \cdot 10^{-4}$. Der IEEE Standard 754 sieht vor, dass exakt eine Stelle vor dem Komma ungleich 0 sein soll. Dies entspricht der

Bedingung $1 \leq m < b$. An diese Konvention wollen wir uns nun auch halten, und dann ist $1.295840352 \cdot 10^{-4}$ die eindeutige Darstellung.

Mit Hilfe eines Gleitkommaformats können wir die signifikantesten Stellen einer Zahl abspeichern. Dies führt dazu, dass der Fehler, der durch das Runden einer reellen Zahl auf ihre nächste darstellbare Gleitkommazahl entsteht, durch eine Konstante begrenzt ist.

Lemma 6 ([Gol91])**.** *Sei $x \in \mathbb{R}$ und $\underline{x}$ eine Gleitkommazahl zur Basis b, die p Stellen für die Mantisse verwendet (plus Vorzeichen), und die unter allen solchen Zahlen minimalen Abstand zu x hat. Dann gilt*

$$\frac{|x - \underline{x}|}{|x|} \leq \frac{1}{2} \cdot b^{-(p-1)},$$

d.h. der relative Fehler ist durch eine Konstante beschränkt, die nur von b und p abhängt. Diese Konstante bezeichnen wir mit $\epsilon_{b,p}$ oder mit $\epsilon_{maschine}$. Sie wird auch als Maschinengenauigkeit *bezeichnet.*

Addition von Gleitkommazahlen

In der normierten Darstellung fällt es uns nun schwer, die Zahlen $1.295840352 \cdot 10^{-4}$ und $1.052631579 \cdot 10^{4}$ zu addieren oder zu subtrahieren. Wir müssen dafür von der Darstellung abweichen und zumindest eine der Zahlen 'denormalisieren'. Es ist üblich (und sinnvoll, siehe dazu das Beispiel aus den Vorlesungsfolien), die Zahl mit dem kleineren Exponenten zu denormalisieren, d.h. an den Exponenten der anderen Zahl anzupassen. In unserem Beispiel lösen wir also die normalisierte Darstellung wieder auf: $1.295840352 \cdot 10^{-4} = 0.00000001295840352 \cdot 10^{4}$. Nun stellt sich die Frage, wie viel Platz wir für die denormalisierte Darstellung zur Verfügung haben und wie wir die Zahl runden.

- Nehmen wir zunächst an, dass wir für die denormalisierte Darstellung nicht mehr Platz zur Verfügung stellen als für die Speicherung der Zahlen selbst vorgesehen ist, in unserem Beispiel also zehn Stellen für die Mantisse. Wir können nun auf- oder abrunden und entscheiden uns, grundsätzlich abzurunden. Dies entspricht einem Verschieben der Zahl nach rechts und einem Abschneiden aller Stellen, die aus dem darstellbaren Bereich herausgeschoben werden. Wir rechnen also:

  $$\begin{aligned} &1.295840352 \cdot 10^{-4} \\ +&0.000000012 \cdot 10^{-4} \\ =&1.295840364 \cdot 10^{-4}. \end{aligned}$$

  Der Fehler, den wir hier durch das Abschneiden der hinteren Ziffern machen, ist im Vergleich zum Ergebnis der Rechnung klein. Leider ist dies nicht immer der Fall. Betrachten wir nun die Addition der Zahlen $1 = 1.000000000 \cdot 10^{0}$ und $-0.999999999 = -9.999999999 \cdot 10^{-1}$. Das richtige Ergebnis bei beliebig großer Darstellungsgenauigkeit ist $0.0000000001 = 1.0 \cdot 10^{-10}$. Mit unserer Beschränkung auf zehn Bits für die Darstellung rechnen wir:

  $$1.000000000 \cdot 10^{0}$$

$$
\begin{aligned}
&+0.999999999 \cdot 10^0 \\
&=0.000000001 \cdot 10^0 \qquad\qquad = 1.0 \cdot 10^{-9}.
\end{aligned}
$$

Der relative Fehler beträgt also $\frac{|1.0 \cdot 10^{-10} - 1.0 \cdot 10^{-9}|}{1.0 \cdot 10^{-10}} = 9$. (Man bemerke, dass der relative Fehler identisch ist zu dem in den Vorlesungsfolien bestimmten, obwohl das Beispiel dort mit einer kleineren Darstellungsgenauigkeit durchgeführt wird). Wir stellen fest, dass für zwei Zahlen x, y im Gleitkommaformat der relative Fehler der Darstellung von $x - y$ im Gleitkommaformat bis zu $b - 1$ betragen kann, wenn man nach obiger Vorgehensweise verfährt (siehe auch Theorem 1 in [Gol91]). Dieser Fehler ist wesentlich größer als die Maschinengenauigkeit und sinkt insbesondere durch die Verwendung von mehr Bits *nicht*.

- Einen Ausweg bietet die Verwendung eines *guard bits*. Dies bedeutet, dass für die Zeit der Berechnung die denormalisierte Zahl mit einem Bit mehr dargestellt wird als die eigentliche Darstellungsgenauigkeit der Architektur. In obigem Beispiel liefert die Berechnung des Ergebnisses unter Verwendung eines Guard Bits das korrekte Ergebnis. In den Vorlesungsfolien finden sich weitere Beispiele, bei denen nur ein kleiner Fehler auftritt. Wir sehen dort jedoch auch, dass der Fehler durchaus größer sein als $\epsilon_{b,p}$ (aber nicht viel).

Wir entscheiden uns also dafür, die denormalisierte Zahl mit $p + 1$ Bits darzustellen und das Ergebnis wieder auf das verwendete Format zu normalisieren. Dann gilt:

Theorem 7 (Theorem 2 in [Gol91])**.** *Seien x, y Gleitkommazahlen nach obigem Modell mit Parametern b und p. Wenn für die Darstellung der kleineren Zahl nach Denormalisierung $p + 1$ Bits verwendet werden und das Ergebnis e dann auf die nächste Gleitkommazahl $\underline{e}$ gerundet wird, dann gilt*

$$\frac{|(x - y) - \underline{e}|}{|x - y|} \leq 2 \cdot \epsilon_{b,p}.$$

Die Verwendung eines Guard Bits sorgt also dafür, dass der Fehler der Darstellung eines Additionsergebnisses (oder Subtraktionsergebnisses) wieder in der Größenordnung der Maschinengenauigkeit liegt.

Summen mit vielen Operanden

Als Abschluss des Kapitels beschäftigen wir uns noch kurz mit der Addition vieler Zahlen. Wir haben in den Vorlesungsfolien gesehen, dass die Addition mehrerer Zahlen in einem Gleitkommaformat das Assoziativgesetz nicht erfüllt: Die Reihenfolge der Berechnung beeinflusst die Rundungsfehler und damit das Gesamtergebnis. Es kann sich daher lohnen, Zahlen in sinnvoller Reihenfolge zu addieren.

Dies ist aber im Allgemeinen nicht ausreichend, um zu verhindern, dass sich die entstehenden Fehler bei der Addition sehr vieler Zahlen immer mehr verstärken. Wir haben dazu in den Vorlesungsfolien Überlegungen angestellt, die im Wesentlichen aufzeigen, dass der erste Summand in der Gesamtsumme mit einem Fehler bis zu $n\epsilon_{b,p}$ belegt ist,

- `S := 0, F := 0`
- `Für alle` $i = 1, \ldots, n$`:`
  - `Y :=` x_i `- F`
  - `Z := S + Y`
  - `F := (Z - S) - Y`
  - `S := Z`

Abbildung 1.1: Der Algorithmus von Kahan.

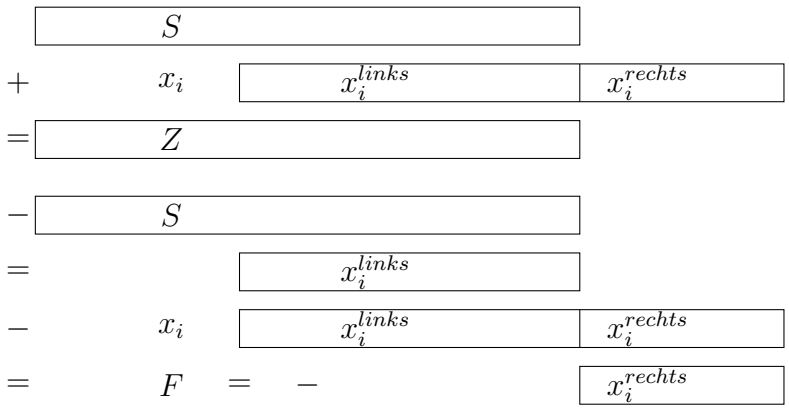


Abbildung 1.2: Schematische Darstellung des Algorithmus von Kahan..

während der letzte Summand nur mit einem Fehler in der Größenordnung $\epsilon_{b,p}$ belastet ist.

Zur Reduktion des aufschaukelnden Rundungsfehlers haben wir den Algorithmus von Kahan kennengelernt (siehe Abbildung 1.1). Der Fehler wird in diesem Algorithmus in einer separaten Variable mitgeführt (die einen anderen Genauigkeitsbereich haben kann) und sammelt sich dort, bis er relevant für den Wert der Summe wird. In Abbildung 1.2 sehen wir eine schematische Darstellung des ersten Schritts ($F = 0$).

Durch Verwendung dieses Algorithmus und dank der Beobachtung, dass $\epsilon_{b,p}^2$ im Vergleich zur Maschinengenauigkeit sehr klein ist, sinkt der Fehler, mit dem der erste Summand behaftet ist, nun auf die Größenordnung der Maschinengenauigkeit, wie das folgende Theorem zeigt.

Theorem 8 (Theorem 8 in [Gol91])**.** *Sei S mit dem Algorithmus in Abbildung 1.1 berechnet. Dann ist*

$$S = \sum_{i=1}^{n} x_i(1 + \delta_i) + \mathcal{O}(n \cdot \epsilon^2) \cdot \sum_{i=1}^{n} |x_i|$$

wobei $|\delta_i| \leq 2\epsilon_{b,p}$ für alle $i \in \{1, \ldots, n\}$.

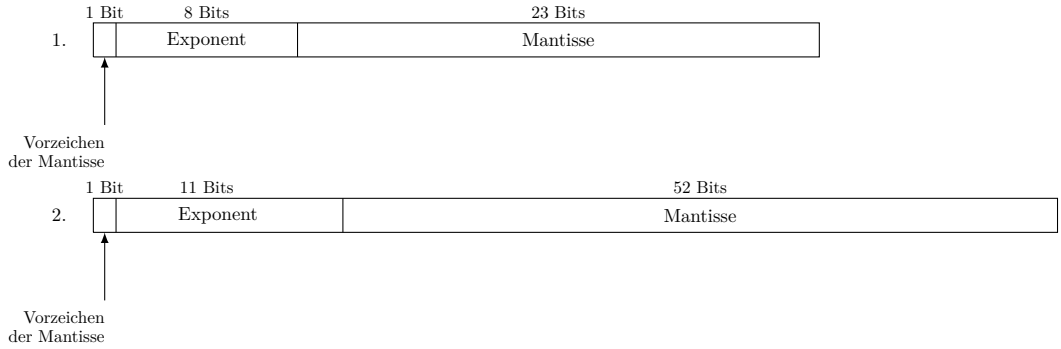


Abbildung 1.3: Aufteilung der Bits im IEEE Standard 754.

Der IEEE Standard

Der IEEE Standard 754 hat sich als allgemein verwendeter Standard zur Darstellung von Gleitkommazahlen durchgesetzt. Er definiert Formate für verschiedene Genauigkeiten,

- *single precision* mit 32 Bits,
- *double precision* mit 64 Bits, sowie
- *extended precision* mit 79 Bits,

wobei der letztgenannte für den internen Gebrauch zur Reduzierung von Rundungsfehlern gedacht ist. Abbildung 1.3 zeigt, wie die 32 bzw. 64 Bits aufgeteilt werden. Wir erklären den single precision Standard genauer. Der Standard verwendet eine Vorzeichenbetragsdarstellung für die Mantisse. Das erste Bit der Darstellung entspricht dem Vorzeichen der Mantisse, die letzten 23 Bits entsprechen der Mantisse. Dazwischen befinden sich 8 Bits für den Exponenten, der in Excess 127 Darstellung dargestellt wird. Der Standard nutzt aus, dass die Normalisierung der Mantisse dazu führt, dass immer genau eine Stelle vor dem Komma ungleich 0 ist, was aufgrund der Basis 2 bedeutet, dass diese Stelle immer 1 ist. Diese 1 wird bei der Speicherung weggelassen.

Der Exponent wird auf den Zahlenbereich $[-126, 127]$ eingeschränkt. Dies hat den Grund, dass der Nullstring und der Einsstring für Spezialanwendungen verwendet werden. Werden die acht Exponentenbits alle auf 0 gesetzt, so stellt der Bitstring keine normalisierte Gleitkommazahl dar. Ist die Mantisse ebenfalls gleich 0, so wird der String als Null interpretiert. Ist die Mantisse ungleich 0, so wird die Zahl als *denomalisierte* Zahl interpretiert: Die Mantisse wird als Zahl mit 0 vor dem Komma interpretiert. So können kleinere denormalisierte Zahlen (etwas ungenauer) dargestellt werden. Werden die acht Exponentenbits alle auf 1 gesetzt, so wird entweder ∞ dargestellt, wenn die Mantisse auf 0 steht, oder `NaN` (not a number), wenn die Mantisse ungleich 0 ist.

Überlegungen zum darstellbaren Bereich des resultierenden Standards finden sich in den Vorlesungsfolien.

Literaturverzeichnis

[Gol91] David Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Survey **23** (1991), no. 1, 5–48.