Dr. Jürg M. Stettbacher
Neugutstrasse 54
CH-8600 Dübendorf

Telefon: +41 43 299 57 23
E-Mail: dsp@stettbacher.ch

Zahlensysteme und Codes

Konzept und Verwendung in der Informatik

Version 2.23
2017-09-18

Zusammenfassung: Es werden die wichtigsten Stellenwert-Zahlensysteme eingeführt. Dann wird gezeigt, wie man negative Zahlen darstellen kann und wie man damit rechnet. Einschränkungen von endlichen Zahlen kommen zur Sprache. Zu Schluss folgt eine Übersicht über ein Auswahl von anderen verbreiteten Zahlen- und Zeichencodes.

Inhaltsverzeichnis

1 Zweck

Dieses Dokument verfolgt die nachstehenden Ziele:

- Einführen in das Rechnen in *Zahlensystemen*. Nebst dem 10-er System sind das insbesondere das 2-er, 8-er und das 16-er System.
- Aufzeigen, wie *negative Zahlen* mit Hilfe von Komplementen dargestellt werden.
- Erläutern der Effekte beim Rechnen mit *endlichen Zahlen*. Carry und Borrow erklären.
- Einführung in die Darstellung von *Fliesspunktzahlen*.
- Darstellung der wichtigsten *Codes.*
- Der Leser sollte nach der Lektüre in der Lage sein, Addition, Subtraktion und Multiplikation in den erwähnten Zahlensystemen von Hand durchzuführen, einschliesslich negativer Zahlen in Komplementdarstellung. Er ist sich der speziellen Effekte beim Rechnen mit endlichen Zahlen bewusst und kennt die wichtigsten Codes.

2 Einleitung

Das uns vertraute Zahlensystem ist das 10-er System. Es ist ein Stellenwert-System. Das heisst, dass jede Ziffer in einer mehrstelligen Zahl entsprechend ihrer Stelle eine Wertigkeit hat. Nehmen wir zum Beispiel die folgende Zahl aus dem 10-er System:

$$31.05 \qquad \text{(10-er System)}$$

Aufgrund der Schreibweise mit dem Dezimalpunkte erkennen wir, dass die Ziffer 3 angibt, dass die Zahl drei Zehner beinhaltet. Die Ziffer 1 sagt, dass sie zudem einen 1 Einer enthält. Weiter hat die Zahl null Zehntel und 5 Hundertstel. Oder etwas technischer ausgedrückt:

$$\begin{aligned} 31.05 &= 3 \cdot 10^{1} + 1 \cdot 10^{0} + 0 \cdot 10^{-1} + 5 \cdot 10^{-2} \\ &= 3 \cdot 10 + 1 \cdot 1 + 0 \cdot 0.1 + 5 \cdot 0.01 \end{aligned}$$

Nebst dem 10-er System sind in der Informatik einige weiter Zahlensysteme von Bedeutung. Das 2-er System drängt sich auf, weil digitale[1] Schaltungen, die nur zwei stabile Zustände zulassen, technisch besonders einfach realisierbar sind. Man bezeichnet solche Schaltungen auch als binär[2]. Jeder entsprechende Schalterausgang realisiert also ein binäre Ziffer. Das 8-er System entsteht, wenn man jeweils drei binäre Ziffern zusammenfasst und ihnen von links nach rechts die Wertigkeiten 2^2, 2^1 und 2^0 verleiht. Jede derartige Dreiergruppe kann dann Werte zwischen 0 und 7 annehmen. Das 16-er System entsteht in analoger Weise, wenn man Gruppen aus vier binären Ziffern bildet und den Ziffern die Wertigkeiten 2^3, 2^2, 2^1 und 2^0 gibt. Jede Vierergruppe kann dann dezimale Werte zwischen 0 und 15 darstellen. Dazu ein Beispiel:

$$\begin{aligned} 31.05\text{ (16-er System)} &= 3\cdot 16^1 + 1\cdot 16^0 + 0\cdot 16^{-1} + 5\cdot 16^{-2}\text{ (10-er System)} \\ &= 49.01953125\text{ (10-er System)} \end{aligned}$$

3 Zahlensysteme

Man vermutet, dass in unserem Kulturkreis das 10-er System verbreitet ist, weil wir zehn Finger haben und diese beim Zählen und Rechnen gelegentlich als Hilfe verwenden. Das 10-er System hat sonst jedoch keinen natürlichen Vorteil gegenüber anderen Zahlensystemen. Es ist einfach eines, das sich gegen andere[3] durchgesetzt hat und heute gemeinhin anerkannt und verwendet wird.

Wie wichtigsten Zahlensysteme der Informatik wollen wir im Folgenden kurz aufzählen.

3.1 10-er System

- Das Zahlensystem mit der Basis 10 heisst 10-er System, Dezimalsystem oder auch dekadisches System.
- Es umfasst 10 Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Wenn nötig, bezeichnen wir Dezimalzahlen mit einem Index *d*.
- Bei mehrstelligen Zahlen ist der Stellenwert der Ziffern von Bedeutung.

[1] *Digital* kommt vom lateinischen Wort *Digitus*, was Finger heisst. Gemeint ist, dass man mit den Fingern zählt, dass es also nur endlich viele ganze Werte gibt.

[2] *Binär* heisst zweiwertig. Der Ausdruck kommt daher, dass man den beiden Zuständen einer digitalen Schaltung die Werte null und eins zuweist.

[3] Beispiele für alternative Zahlensysteme: Das bereits im alten Sumer bekannte 60-er System hat sich bis heute in der Messung von Zeit und Winkeln erhalten. Das 12-er System kommt bei uns noch als Dutzend vor. Ausserdem steckt im Begriff *Grosshandel* das *Gros* drin, das aus 12 Dutzend besteht. Ebenso umfasst die englische Einheit *Foot* genau 12 *Inches*. Bis zur Münzreform von 1971 herrschte in Grossbritannien ein sonderbares Gemisch: 1 Pound = 4 Crown, 1 Crown = 5 Shilling, 1 Shilling = 12 Pence, also 1 Pound = 20 Shilling = 240 Pence.

- Beispiel:

$$\begin{aligned} 537.4_d &= 5 \cdot 10_d^2 + 3 \cdot 10_d^1 + 7 \cdot 10_d^0 + 4 \cdot 10_d^{-1} \\ &= 5 \cdot 100_d + 3 \cdot 10_d + 7 \cdot 1_d + 4 \cdot 0.1_d \end{aligned}$$

3.2 2-er System

- Das Zahlensystem mit der Basis 2 heisst 2-er System, Binärsystem oder Dualsystem.
- Es umfasst lediglich 2 Ziffern: 0, 1.
- Wenn nötig, bezeichnen wir Binärzahlen mit einem Index b.
- Bei mehrstelligen Zahlen ist der Stellenwert der Ziffern von Bedeutung.
- Beispiel:

$$\begin{aligned} 1011.1_b &= 1 \cdot 2_d^3 + 0 \cdot 2_d^2 + 1 \cdot 2_d^1 + 1 \cdot 2_d^0 + 1 \cdot 2_d^{-1} \\ &= 1 \cdot 8_d + 0 \cdot 4_d + 1 \cdot 2_d + 1 \cdot 1_d + 1 \cdot 0.5_d \\ &= 11.5_d \end{aligned}$$

3.3 8-er System

- Das Zahlensystem mit der Basis 8 heisst 8-er System oder Oktalsystem.
- Es umfasst 8 Ziffern: 0, 1, 2, 3, 4, 5, 6, 7.
  Die Ziffern 8 und 9 kommen nicht vor.
- Wenn nötig, bezeichnen wir Oktalzahlen mit einem Index o.
- Bei mehrstelligen Zahlen ist der Stellenwert der Ziffern von Bedeutung.
- Beispiel:

$$\begin{aligned} 537.4_o &= 5 \cdot 8_d^2 + 3 \cdot 8_d^1 + 7 \cdot 8_d^0 + 4 \cdot 8_d^{-1} \\ &= 5 \cdot 64_d + 3 \cdot 8_d + 7 \cdot 1_d + 4 \cdot 0.125_d \\ &= 351.5_d \end{aligned}$$

3.4 16-er System

- Das Zahlensystem mit der Basis 16 heisst 16-er System oder Hexadezimalsystem.
- Es umfasst 16 Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
  Da unser bekanntes Zahlensystem nur zehn Ziffern umfasst, behilft man sich mit Buchstaben um den Ziffernsatz für das 16-er System zu erweitern.
- Wenn nötig, bezeichnen wir Hexadezimalzahlen mit einem Index h.
- Bei mehrstelligen Zahlen ist der Stellenwert der Ziffern von Bedeutung.
- Beispiele:

$$\begin{aligned} 537.4_h &= 5 \cdot 16_d^2 + 3 \cdot 16_d^1 + 7 \cdot 16_d^0 + 4 \cdot 16_d^{-1} \\ &= 5 \cdot 256_d + 3 \cdot 16_d + 7 \cdot 1_d + 4 \cdot 0.0625_d \\ &= 1335.25_d \end{aligned}$$

$$\begin{aligned} AF3.C_h &= 10_d \cdot 16_d^2 + 15_d \cdot 16_d^1 + 3_d \cdot 16_d^0 + 12_d \cdot 16_d^{-1} \\ &= 10_d \cdot 256_d + 15_d \cdot 16_d + 3_d \cdot 1_d + 12_d \cdot 0.0625_d \\ &= 2803.75_d \end{aligned}$$

3.5 Vergleich

Die folgende Tabelle stellt identische Werte zwischen 0 und 15 in verschiedenen Zahlensystemen dar.

10-er System	2-er System	8-er System	16-er System
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

3.6 Umwandlungen

Für die Umwandlung von Werten in verschiedene Zahlensystem verwendet man oft das Horner-Schema. Wir wollen es anhand von einigen Beispielen erläutern.

10-er ins 8-er System Wieviel ist 100_d im 8-er System? Das Horner-Schema funktioniert wie folgt.

$$\begin{array}{rcrcrcl} 100_d & \div & 8 & = & 12 & \text{Rest} & 4 \\ 12_d & \div & 8 & = & 1 & \text{Rest} & 4 \\ 1_d & \div & 8 & = & 0 & \text{Rest} & 1 \end{array}$$

Im Horner-Schema dividieren wir durch 8 weil wir ins 8-er System umwandeln wollen. Wir beginnen mit der gesuchten Dezimalzahl, in unserem Beispiel mit 100_d. Die Division $100/8$ ergibt 12, Rest 4. Auf der zweiten Zeile fahren wir mit dem ganzzahligen Quotienten 12 weiter, dividieren wieder durch 8 und notieren Quotient und Rest. Am Ende lesen wir die Reste von *unten noch oben* aus. Dies ist das Resultat.

$$100_d = 144_o$$

Das Resultat können wir leicht überprüfen:

$$144_o = 1 \cdot 8_d^2 + 4 \cdot 8_d^1 + 4 \cdot 8_d^0 = 100_d$$

Man kann unschwer nachvollziehen, dass das Horner-Schema die gesuchte Zahl folgendermassen darstellt:

$$144_o = \overbrace{8_d \cdot (8_d \cdot (1) + 4) + 4}^{\text{Horner-Darstellung}} = 8_d \cdot 8_d \cdot 1 + 8_d \cdot 4 + 4$$

Der Vergleich mit dem Test von oben zeigt, dass beide Formen das selbe beschreiben. Die Horner-Darstellung zerlegt nun die ursprüngliche Zahl in verschachtelte Klammerausdrücke. Die äussere Klammer ist der ganzzahlige Quotient der ersten Zeile im Horner-Schema, der abgespaltene Summand 4 ist der Rest. Der Ausdruck in der Klammer entspricht der zweiten und dritten Zeile im Horner-Schema, also der weiteren Zerlegung der Zahl nach dem selben Muster.

10-er ins 16-er System Wieviel ist 100_d im 16-er System? Wir wenden genau das selbe Schema an wie oben beschrieben, aber für die Basis 16.

$$\begin{array}{rcrcrcr} 100_d & \div & 16 & = & 6 & \text{Rest} & 4 \\ 6_d & \div & 16 & = & 0 & \text{Rest} & 6 \end{array}$$

Es folgt:

$$100_d = 64_h$$

Test:

$$64_h = 6 \cdot 16_d^1 + 4 \cdot 16_d^0 = 100_d$$

10-er ins 2-er System Wir wollen nun noch sehen, wie Zahlen mit Kommastellen unzuwandeln sind. Wir wählen den Wert 26.6875_d. Diesen Wert zerlegen wir:

$$26.6875_d = 26_d + 0.6875_d$$

Zuerst wandeln wir den ganzzahligen Teil um:

$$\begin{array}{rcrcrcr} 26_d & \div & 2 & = & 13 & \text{Rest} & 0 \\ 13_d & \div & 2 & = & 6 & \text{Rest} & 1 \\ 6_d & \div & 2 & = & 3 & \text{Rest} & 0 \\ 3_d & \div & 2 & = & 1 & \text{Rest} & 1 \\ 1_d & \div & 2 & = & 0 & \text{Rest} & 1 \end{array}$$

Wir erhalten:

$$26_d = 11010_b$$

Das Horner-Schema für die Nachkommastellen geht so:

$$\begin{array}{rcrcrcl} 0.6875_d & \cdot & 2 & = & 0.3750 & + & 1 \\ 0.3750_d & \cdot & 2 & = & 0.7500 & + & 0 \\ 0.7500_d & \cdot & 2 & = & 0.5000 & + & 1 \\ 0.5000_d & \cdot & 2 & = & 0.0000 & + & 1 \end{array}$$

Hier lesen wir die Spalte ganz rechts von oben nach unten aus:

$$0.6875_d = 0.1011_b$$

Es folgt das Resultat:

$$26.6875_d = 11010.1011_d$$

4 Negative Zahlen

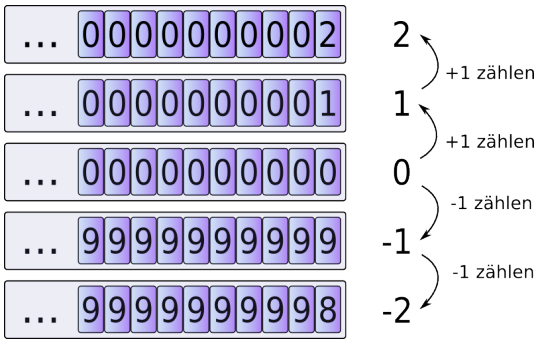
Normale negative Zahlen werden mit einem Minuszeichen vor dem Betrag markiert. Wird eine negative Zahl addiert, so sind wir gewohnt, statt der Addition eine Subtraktion des Betrags aufzuführen. Und soll eine negative Zahl subtrahiert werden, addieren wir statt dessen den Betrag.

$$\begin{array}{rcl} a+(-b) & = & a-b \\ a-(-b) & = & a+b \end{array}$$

Für Anwendungen in der Informatik sind diese Regeln denkbar ungünstig: Wenn eine Maschine den Befehl bekommt, zwei Variablen zu addieren, so soll tatsächlich eine Addition ausgeführt werden. Eine werteabhängige Substitution des Befehls ist für Maschinen nur umständlich. Daher suchen wir eine geeignete Darstellung negativer Zahlen, so dass eine Addition in jedem Fall als Addition ausgeführt werden kann.

Komplement Betrachten wir einen Kilometerzähler beim Auto. Am Anfang stehe der Zähler auf null. Wenn ich einen Kilometer gefahren bin, dann zeigt er eins, dann zwei, drei, usw. Was aber passiert, wenn ich beim Stand null einen Kilometer rückwärts fahre? Der Kilometerzähler wird dann lauter Neuner zeigen. Diese Zahl entspricht demnach dem Wert minus eins.

Der Sachverhalt ist in der folgenden Abbildung dargestellt. Wir stellen uns dabei vorerst einen unendlich breiten Kilometerzähler vor. Dies ist durch die Punkte am linken Rand angedeutet.



Die in dieser Weise gebildeten negativen Zahlen heissen *Komplemente*. Das 9-er Komplement (im 10-er System) entsteht, indem man jede Ziffer einer Zahl durch die Ergänzung auf 9 ersetzt. Das 10-er Komplement erhält man aus dem 9-er Komplement[4], indem man an der letzten Stelle, also ganz rechts eine entsprechend gewichtet eins dazu zählt.

Beispiel:

Ursprüngliche Zahl	**...**	**0**	**0**	**0**	**0**	**0**	**1** $_d$
9-er Komplement	**...**	**9**	**9**	**9**	**9**	**9**	**8** $_d$
10-er Komplement	**...**	**9**	**9**	**9**	**9**	**9**	**9** $_d$

Wenn wir das Beispiel mit der Abbildung des Kilometerzählers vergleichen, so stellen wir fest, dass das 10-er Komplement einer positiven Zahl der entsprechenden negativen Zahl entspricht. Umgekehrt ist das 10-er Komplement einer negativen Zahl ihr Betrag.

Ursprüngliche Zahl	**...**	**9**	**9**	**9**	**9**	**9**	**9** $_d$
9-er Komplement	**...**	**0**	**0**	**0**	**0**	**0**	**0** $_d$
10-er Komplement	**...**	**0**	**0**	**0**	**0**	**0**	**1** $_d$

Wir wollen hier noch ein Beispiel mit Kommastellen im Binärsystem zeigen. Gesucht sei die binäre Darstellung von -5.25_d im 2-er System:

1. Schritt, Betrag im Binärformat:

$$+5.25_d = \ldots 000101.01_b$$

2. Schritt, 2-er Komplement davon:

Ursprüngliche Zahl	**...**	**0**	**0**	**0**	**1**	**0**	**1.**	**0**	**1** $_b$
1-er Komplement	**...**	**1**	**1**	**1**	**0**	**1**	**0.**	**1**	**0** $_b$
2-er Komplement	**...**	**1**	**1**	**1**	**0**	**1**	**0.**	**1**	**1** $_b$

[4] Das 9-er Komplement hat die ungünstige Eigenschaft, dass es zwei Darstellungen der Null gibt, nämlich $\ldots 000000_d$ und $\ldots 111111_d$. Dieses Problem hat das 10-er Komplement nicht.

Beachte, dass in diesem Fall die Eins, die auf der rechten Seite addiert wurde, um das 2-er Komplement zu erhalten, das Gewicht 2^{-2} hat.

Kennzeichnung Woran erkennt man, dass ein gegebenes Muster von Ziffern eine positive oder eine negative Zahl ist? Beispielsweise könnte die Zahl $\ldots 111010.11_b$ eine sehr grosse positive Zahl sein, oder eben eine negative. Irgendwo in der Unendlichkeit treffen sich positive und negative Zahlen.

Als Konvention hat sich die pragmatische Regel durchgesetzt, dass eine Zahl negativ sei, wenn irgendwo weit links keine Nullen mehr stehen. Andernfalls ist sie positiv. Mit den Punkten in $\ldots 111010.11_b$ deuten wir an, dass nach links lauter Einer folgen. Die Zahl ist also negativ.

5 Rechnen in Zahlensystemen

5.1 Addition und Subtraktion

Die schriftliche Addition und Subtraktion wird in allen Zahlensystemen genau gleich ausgeführt wie wir es im 10-er System immer schon taten.

Beispiele Es folgen Beispiele zu verschiedenen Zahlensystemen.

Addition im Binärformat:

Erster Summand			**1**	**0**	**1**	**1.**	**1**	$_b$
Zweiter Summand	**+**		**1**	**1**	**0**	**1.**	**0**	$_b$
Übertrag		1	1	1	1			
Resultat		**1**	**1**	**0**	**0**	**0.**	**1**	$_b$

Subtraktion im Oktalformat:

Minuend		**7**	**0**	**2**	**3.**	**1**	$_o$
Subrahend	**-**	**1**	**4**	**1**	**4.**	**2**	$_o$
Übertrag		1		1	1		
Resultat		**5**	**4**	**0**	**6.**	**7**	$_o$

Addition im Hexadezimalsystem:

Erster Summand		**3**	**0**	**C**	**E.**	**3**	**A** $_h$
Zweiter Summand	**+**	**A**	**1**	**F**	**3.**	**1**	**2** $_h$
Übertrag			1	1			
Resultat		**D**	**2**	**C**	**1.**	**4**	**C** $_h$

Zum Schluss wollen wir noch ein Beispiel mit Komplementen vorführen. Und zwar soll die folgende Rechnung im Binärsystem ausgeführt werden: $3.5_d + (\text{-}5.0_d) = (\text{-}1.5_d)$

3.5_d	=	...	**0**	**0**	**0**	**1**	**1.**	**1** $_b$
5.0_d	=	...	**0**	**0**	**1**	**0**	**1.**	**0** $_b$

2-er Komplement:

$+5.0_d$	**...**	**0**	**0**	**1**	**0**	**1.**	**0** $_b$
1-er Komplement	**...**	**1**	**1**	**0**	**1**	**0.**	**1** $_b$
2-er Komplement	**...**	**1**	**1**	**0**	**1**	**1.**	**0** $_b$

Addition:

3.5_d		**...**	**0**	**0**	**0**	**1**	**1.**	**1** $_b$
$\text{-}5.0_d$	**+**	**...**	**1**	**1**	**0**	**1**	**1.**	**0** $_b$
Übertrag					1	1		
Resultat		**...**	**1**	**1**	**1**	**1**	**0.**	**1** $_b$

Das Resultat ist negativ, daher bilden wir das 2-er Komplement davon:

Resultat	**...**	**1**	**1**	**1**	**1**	**0.**	**1** $_b$
1-er Komplement	**...**	**0**	**0**	**0**	**0**	**1.**	**0** $_b$
2-er Komplement	**...**	**0**	**0**	**0**	**0**	**1.**	**1** $_b$

Das 2-er Komplement vom Resultat ist 1.5_d. Das bedeutet, dass das Resultat oben $\text{-}1.5_d$ entspricht, wie erwartet.

5.2 Multiplikation und Division

Die Multiplikation von positiven Zahlen folgt ebenfalls der bekannten Regel. Aufpassen muss man dagegen bei negativen Zahlen, da die Fortsetzung nach links bei negativen Zahlen berücksichtigt werden muss. Auf die Division wollen wir in diesem Zusammenhang nicht weiter eintreten.

Beispiele Multiplikationen im Binärsystem.

Positive Faktoren:
(Wir nehmen an, dass links jeder Zahl lauter Nullen stehen.)

	1	**0**	**1**	$_b$	**x**	**1**	**1**	**1**	**0**	$_b$
						1	**1**	**1**	**0**	
	+				**0**	**0**	**0**	**0**		
	+			**1**	**1**	**1**	**0**			
Übertrag				1	1					
Resultat			**1**	**0**	**0**	**0**	**1**	**1**	**0**	$_b$

Mindestens ein negativer Faktor:

$$3.5_d \cdot (\text{-}5.0_d) = (\text{-}17.5_d)$$

Wenn wir die Faktoren beispielsweise mit 4 Stellen vor dem Komma und 2 Stellen nach dem Komma darstellen, total also 6 Stellen, so müssen wir für das Resultat doppelt so viele Stellen vorsehen. Nur dann ist das Resultat sicher darstellbar. Dabei ist es ganz wichtig, dass bereits die Faktoren im doppelten Format dargestellt werden. Die Verbreiterung einer Zahl nach links nennt man *Sign-Extension*, da nicht einfach Nullen vor die Zahl gestellt werden dürfen. Statt dessen wird die Stelle, welche die Vorzeichenfunktion hat, nach links propagiert.

...	**0**	**0**	**1**	**1.**	**1**	**0**	$_b$	**x**	**...**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1.**	**0**	**0**	$_b$
									...	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1.**	**1**	**0**	
								+	**...**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1.**	**0**		
								+	**...**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0.**			
Übertrag									2	0	2	0	2	0	1	1	1	1	0	0		
Resultat									**...**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0.**	**1**	**0**	$_b$

Das Resultat ist negativ. Das sehen wir daran, dass nach links lauter Einer laufen. Um zu sehen, wie gross die Zahl ist, die im 2-er Komplement steckt, bilden wir davon wieder das 2-er Komplement:

	...	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**.**	**1**	**0**	$_b$
1-er Komplement	**...**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**.**	**0**	**1**	$_b$
2-er Komplement	**...**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**.**	**1**	**0**	$_b$

Wir erhalten $+17.5_d$, das Resultat der Multiplikation ist also -17.5_d.

6 Endliche Zahlen

In Computern sind Zahlen endlich, denn Zahlen werden in Registern mit einer bestimmten Anzahl Bits dargestellt und verarbeitet. Es lassen sich also nicht beliebig grosse Zahlen repräsentieren.

Beispiele:

Dezimalzahl	**in 4 Bit**				**in 8 Bit**							
0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	1	0	0	0	0	0	0	1	1
7	0	1	1	1	0	0	0	0	0	1	1	1
8	1	0	0	0	0	0	0	0	1	0	0	0
15	1	1	1	1	0	0	0	0	1	1	1	1
-8	1	0	0	0	1	1	1	1	1	0	0	0
-1	1	1	1	1	1	1	1	1	1	1	1	1

Beachte, dass es im Beispiel bei der Darstellung mit 4 Bit zu zwei Problemen kommt[5]:

- Die Zahlen 15 und -1 sind identisch.
- Die Zahlen 8 und -8 sind identisch.

Vorzeichenlose Zahlen Die Ursache des Problems ist in der folgenden Abbildung für ein Register mit 4 Bit dargestellt. Man nennt diese Darstellung den Zahlenkreis. Beginnt man bei der Null und zählt aufwärts, so gelangt man nach einiger Zeit wieder zur Null. Geht man davon aus, dass alle Zahlen im Register vorzeichenlos, also positiv oder null sind, so passiert ein Sprung zwischen 15 und 0. Dieser Sprung geschieht, weil wir grössere Zahlen als 15 in 4 Bit nicht darstellen können. Bei der Addition kann also beispielsweise folgendes passieren:

[5] Bei der Darstellung mit 8 oder mehr Bit treten die selben Probleme bei grösseren Zahlen genauso auf.

$$7_d + 12_d = 0111_b + 1100_b = 0011_b = 3_d$$

Das Resultat ist offensichtlich falsch. Berücksichtigt man aber, dass es zusätzlich zum Resultat im Register einen Übertrag (engl. *Carry*) der Wertigkeit 16 gibt, so ist das Resultat wieder korrekt: $16_d + 3_d = 19_d$. Tritt der Übertrag nach unten auf, also beispielsweise bei der Subtraktion $7_d - 12_d = 0111_b - 1100_b = 1011_b = 11_d$, so nennt man den Übertrag englisch *Borrow*, denn er hat die Wertigkeit -16, so dass $-16+11 = -5$ dem korrekten Resultat entspricht.

Mikroprozessoren haben ein spezielles Flag, das den Übertrag speichert. Es steht dem Anwender frei, dieses Flag zu berücksichtigen oder nicht und damit Rechenfehler in Kauf zu nehmen oder abzufangen. Hochsprachen stellen dem Anwender aber in der Regel das Flag *nicht* zur Verfügung. Es ist also in jedem Fall Vorsicht geboten.

Vorzeichenbehaftete Zahlen Betrachten wir nun Zahlen im 2-er Komplement, so zählen wir von null aus nach rechts in die positive Richtung und von null aus nach links in die negative Richtung. Irgendwo stossen die beiden Richtungen zusammen und es muss ein Sprung zwischen positiven und negativen Zahlen auftreten. Aber wo?

Es gibt kein Naturgesetz, das besagt, wo dieser Sprung passiert. Man hat sich aber - wie weiter oben schon erwähnt - weltweit darauf geeinigt, dass die Sprungstelle dort auftritt, wo das Bit ganz auf der linken Seite im Register, also das höchstwertige Bit (MSB[6]), seinen Wechsel macht. Ist das höchstwertige Bit null, so ist der Registerinhalt null oder positiv. Ist das MSB eins, so ist die Zahl negativ. Es

[6] MSB steht für *Most Significant Bit*. Im Gegensatz dazu wird das niederwertigste Bit als LSB abgekürzt, was für *Least Significant Bit* steht.

gibt also beim aufwärts Zählen einen Sprung von +7 nach -8. Zählt man von null aus abwärts, so gibt es einen Sprung von -8 nach +7. Es passiert also wiederum ein Rechenfehler, wenn wir zum Beispiel dies ausrechnen:

$$3_d + 6_d = 0011_b + 0110_b = 1001_b = -7_d$$

Wir haben die Sprungstelle überschritten, es ist wiederum ein Übertrag der Wertigkeit 16 entstanden ($16_d + (\text{-}7_d) = 9_d$), allerdings an einer anderen Stelle als im oben beschriebenen Fall. Wir nennen diesen Übertrag den *vorzeichenbehafteten* Übertrag, den oben beschriebenen nennen wir den *vorzeichenlosen* Übertrag. Für die Unterscheidung der beiden Fälle hat ein Mikroprozessor je ein separates Flag.

Betrachten wir nun zum Beispiel das Bitmuster 1010_b in der Abbildung des Zahlenkreises, so kann diese Zahl entweder eine 10_d oder eine $\text{-}6_d$ sein. Welches von beiden ist richtig? Es sind eben beide richtig, aber nur eines davon ist gemeint. Der Benutzer des Registers muss sich zuerst darauf festlegen, ob er im Register vorzeichenlose Zahlen (im Bereich 0 bis 15) oder vorzeichenbehaftete Zahlen (im Bereich -8 bis +7) speichern will. Daraus folgt dann die Bedeutung eines Bitmusters. Der nicht eingeweihte Betrachter kann dagegen nicht sicher sein, welchem Wert das Bitmuster entspricht.

Zusammenfassung

Bei endlichen binären Zahlen können einige Probleme auftreten:

- $\Longrightarrow$ Es ist nicht jede beliebig grosse Zahl darstellbar.
- $\Longrightarrow$ Die zahlenmässige Bedeutung eines Bitmusters hängt davon ab, ob man von vorzeichenlosen oder vorzeichenbehafteten Zahlen spricht.
- $\Longrightarrow$ Bei der Berechnung von Summen oder Produkten kommt es zu Überläufen, wenn das Resultat nicht mehr darstellbar ist.
- $\Longrightarrow$ Bei vorzeichenlosen Zahlen passieren Überläufe zwischen 0 und der grössten darstellbaren Zahl.
- $\Longrightarrow$ Bei vorzeichenbehafteten Zahlen passieren Überläufe zwischen der grössten positiven und der kleinsten negativen Zahl.
- $\Longrightarrow$ Bei Überläufen kann ein falsches Resultat entstehen, wenn das betreffende Überlaufsflag (Carry, Borrow) nicht beachtet wird (was der Normalfall ist).

7 Fliesspunktzahlen

Heute werden auf fast allen Rechnern Fliesspunktzahlen gemäss dem Standard IEEE 754 dargestellt. Demzufolge wird eine binäre Fliesspunktzahl F nach diesem Muster gebildet:

$$F = (\text{-}1)^s \cdot \left(1 + \frac{m}{2^M}\right) \cdot 2^{e-\left(2^{E-1}-1\right)}$$

Dabei ist:

s	Das Vorzeichenbit.
M	Die Anzahl Bits der Mantisse.
$\underline{m}$	Die Mantisse, dargestellt als Wort aus M Bits.
m_k	Das k-te Bit der Mantisse.
E	Die Anzahl Bits des Exponenten.
$\underline{e}$	Der Exponent, dargestellt als Wort aus E Bits.
e_k	Das k-te Bit des Exponenten.

Die Bits sind folgendermassen in einem Wort angeordnet:

s	e_{E-1}	e_{E-2}	$\ldots$	e_2	e_1	e_0	m_{M-1}	m_{M-2}	$\ldots$	m_2	m_1	m_0

Der Standard definiert verschiedene Wortgrössen:

Wortgrösse	M	E	kleinste Zahl	grösste Zahl	Inkrement
32 Bit	23	8	$2^{-126} = 10^{-38}$	$2^{128} = 10^{38}$	$6 \cdot 10^{-8}$
64 Bit	52	11	$2^{-1022} = 10^{-308}$	$2^{1024} = 10^{308}$	$1 \cdot 10^{-16}$
79 Bit	63	15	$2^{-16382} = 10^{-4932}$	$2^{16384} = 10^{4932}$	$5 \cdot 10^{-20}$

Darüber hinaus definiert der Standard einige spezielle Werte:

$\underline{m}$	$\underline{e}$	Bedeutung
$[00\ldots000]$	$[00\ldots000]$	0
$[00\ldots000]$	$[11\ldots111]$	$\pm\infty$
$\neq [00\ldots000]$	$[11\ldots111]$	NaN

NaN heisst, dass es sich um keine gültige Zahl handelt (Not a Number).

Beispiel: Wir betrachten die 32-Bit Zahl:

$$\left[0\,100'0000'0\,110'0000'0000'0000'0000'0000\right]$$

Also:

$$\begin{aligned} s &= 0 \\ \underline{e} &= 1000'0000_b = 128_d \\ \underline{m} &= 110'0000'0000'0000'0000'0000_b = 6'291'456_d \end{aligned}$$

Daraus folgt der Wert F der Zahl zu:

$$\begin{aligned} F &= (-1)^0 \cdot \left(1 + \frac{6'291'456}{2^{23}}\right) \cdot 2^{128-(2^{8-1}-1)} \\ &= 1 \cdot 1.75 \cdot 2^1 \\ &= 3.5 \end{aligned}$$

8 Allgemeine Codes

Prüfungsrelevant

Wir betrachten hier einige allgemein gebräuchliche binäre Codes. Ein Code mit Codeworten der Länge n Bits umfasst 2^n mögliche Codeworte, von denen nicht notwendigerweise alle benutzt werden.

Beispiel: Wie gross ist die Codewortlänge n (in Bits) zu wählen um die 26 Buchstaben des Alphabets zu codieren?

$$2^n \geq 26$$

Lösung:

$$n \cdot \log(2) \geq \log(26) \qquad \Longrightarrow \qquad n \geq \frac{\log(26)}{\log(2)} = \log_2(26) = 4.70$$

Man benötigt demnach mindestens $n = 5$ ganze Bits. Der Code umfasst dann $2^n = 32$ Codeworte, wovon 6 nicht benutzt werden.

8.1 BCD Code

Der BCD Code (Binary Coded Decimal) wird verwendet, wenn binäre Zahlen im Dezimalformat auf einer Anzeige dargestellt werden sollen. Jeder Dezimalziffer entspricht dann ein BCD Codewort mit 4 Bit.

BCD-Ziffer	Dezimalziffer
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	ungültig
...	ungültig
1111	ungültig

Beispiel Der Wert $10001_b = 17_d$ soll in BCD dargestellt werden.

dezimale 1er Ziffer	7_d	BCD: 0111_{BCD}
dezimale 10er Ziffer	1_d	BCD: 0001_{BCD}

Lösung: $10001_b = 0001'0111_{BCD}$

Die Umrechnung der BCD-Zahl in die normale binäre Darstellung geht so:

$$\begin{aligned} 0001'0111_{BCD} &= \overbrace{10_d}^{Wertigkeit} \cdot \overbrace{0001_{BCD}}^{10erZiffer} + \overbrace{1_d}^{Wertigkeit} \cdot \overbrace{0111_{BCD}}^{1erZiffer} \\ &= 1010_b \cdot 0001_{BCD} + 0001_b \cdot 0111_{BCD} \\ &= 1010_b \cdot 0001_b + 0001_b \cdot 0111_b \\ &= 1010_b + 0111_b \\ &= 10001_b \end{aligned}$$

Andere Codes für Dezimalziffern Nebst dem oben erwähnten BCD Code gibt es noch eine Reihe von weiteren Codes für die Darstellung von Dezimalziffern. Beispiele sind der Excess-3 Code und der 2421 Code. Beide Codes zielen darauf ab, dass mit Hilfe des 1-er Komplements das 9-er Komplement der betreffenden Dezimalziffer gebildet werden kann. Man nennt solche Codes auch komplementierend. Das war nützlich zu einer Zeit, als es Computer gab, die in BCD rechneten. Heute rechnen Computer fast ausschliesslich binär und diese Codes spielen keine grosse Rolle mehr.

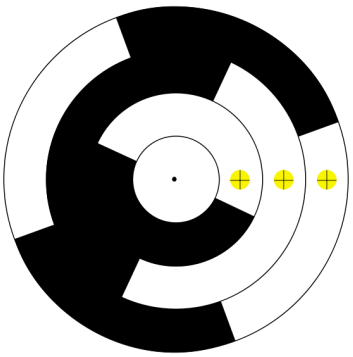
8.2 Gray Code

Der Gray Code wird oft in mechanischen Anwendungen eingesetzt. Benachbarte Codeworte unterscheiden sich nur in einem Bit, so dass bei der Abtastung (z. B. durch Schleifkontakte auf einem Code-Rad) keine Ablesefehler entstehen. Gray Codes werden daher etwa eingesetzt in Winkelsensoren, bei binären Drehschaltern, usw.

Als Beispiel betrachten wir den Gray Code mit 3 Bit:

dezimal	Gray Code
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

Die folgende Darstellung zeigt ein entsprechendes Code-Rad. Die Kreuze bezeichnen die Orte, wo der Code mit Schleifkontakten abgetastet wird. Die hellen Flächen werden als eine logische Null gelesen, die dunklen Flächen als eine Eins. Bewegt sich die Scheibe im Uhrzeigersinn während die Abtastpunkt am Ort bleiben, so lesen die Schleifkontakte den Code gemäss der Tabelle oben ab.



Beispiel Betrachten wir den Übergang von 3 auf 4 im Gray Code. Es ändert nur das Bit auf der linken Seite. Demnach können beim Wechsel nur die Zahlen 3 und 4 entstehen. Im Gegensatz dazu wechseln im normalen Binärcode ($3 = 011_b$, $4 = 100_b$) alle drei Bits ihren Zustand. Dies tun sie bei mechanischer Abtastung nie exakt gleichzeitig, so dass der Ablauf zum Beispiel so aussehen kann:

$$011_b \longrightarrow 111_b \longrightarrow 110_b \longrightarrow 100_b$$

Beim Wechsel von 3 auf 4 können also diverse falsche Zwischenwerte entstehen. Der Gray Code hat dieses Problem nicht.

8.3 ASCII Code

Der ASCII Code stammt aus der Zeit, als auf Grossrechner noch über Terminals zugegriffen wurde. Kommandos vom Terminal zum Rechner wurden seriell übertragen, wobei jedes Terminalzeichen (inkl. die Steuerzeichen des Terminals) in 7 Bit codiert war.

Später, als sich das Byte (an 8 Bit) als Speichereinheit durchgesetzt hatte, wurde der ASCII Code in 8 Bit eingebettet. Die zusätzlichen Codeworte wurden für weitere Zeichen verwendet, wobei sich aber unterschiedliche Zeichensätze etablierten[7].

ASCII Code mit 7 Bit:

Code	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
0..	NL	SH	SX	EX	ET	EQ	AK	BL	BS	HT	LF	VT	FF	CR	SO	SI
1..	DE	D1	D2	D3	D4	NK	SN	EB	CN	EM	SB	EC	FS	GS	RS	US
2..	SP	!	“	#	$	%	&	’	(	)	*	+	,	-	.	/
3..	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4..	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5..	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6..	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7..	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DL

Die Codes 00_h bis $1F_h$ und $7F_h$ enthalten Steuerzeichen, zum Beispiel HT (Horizontal Tab), LF (Line Feed), CR (Carriage Return) und DL (Delete). Die übrigen Codeworte stellen im amerikanischen Englisch typische Zeichen dar. Nicht vorhanden sind jedoch europäische Umlaute und Akzente. Diese wurden in den 8 Bit ASCII-Varianten oberhalb des Codewortes $7F_h$ untergebracht.

Beispiele

- Das Zeichen mit dem 7-Bit Code 61_h ist der Buchstabe *a*.
- Der String *Hallo* lautet codiert: $48_h\ 61_h\ 6C_h\ 6C_h\ 6F_h$.
- Das Zeichen *ü* hat in ISO 8859-1 den Code FC_h, der Buchstabe *à* den Code $E0_h$.

8.4 Unicode

Unicode beschreibt ähnlich wie ASCII einen Zeichensatz. Er ist aber moderner und fasst alle bekannten Zeichen der verschiedenen Sprachen und Schriften zusammen. Damit entfällt (theoretisch) die Inkompatibilität zwischen Dokumenten und Systemen mit unterschiedlichen Zeichensätzen.

[7] Dazu zählen unter anderem die ISO 8859-X Zeichensätze, die auch als ISO Latin-X bezeichnet werden. Dabei steht die ein- oder zweistellige Zahl X für eine Sprachregion, wie Westeuropa (1), Nordeuropa (4), Arabien (6), usw.

Die Unicode-Zeichentabelle hat Platz für rund 1 Mio. Zeichen, wovon erst etwa 10 % belegt sind. Der Zeichensatz kann daher bei Bedarf erweitert werden. Das geschieht auch. 1991 wurde die Version 1.0.0 publiziert. Seither gab es durchschnittlich alle 15 Monate eine Erweiterung.

Für die Anwendung wird die Unicode Tabelle in unterschiedlich lange Zeichencodes codiert. Und dazu gibt es (leider) mehrere Systeme. Die bekanntesten sind UTF-16 und UTF-8[8]. Von UTF-16 gibt es zudem je eine Variante mit little-endian und eine mit big-endian[9]. UTF-16 ist unter anderem verbreitet auf den Systemen Windows und Mac OS X, sowie in Java. UTF-8 wird mehrheitlich im Internet (z. B. für Webseiten) und auf praktisch allen Unix-Derivaten verwendet. Ein Vorteil von UTF-8 ist die Kompatibilität zu ASCII in den untersten 128 Codepositionen und der kleine Speicherbedarf bei Verwendung von typischen westlichen Zeichen. Im Gegensatz dazu ist der Grundzeichensatz von UTF-16 bereits viel grösser. In beiden UTF-Varianten werden seltene Zeichen[10] durch längere Codeworte abgebildet. Zu diesem Zweck gibt es im Grundzeichensatz Marker, die anzeigen, dass ein längerer Code vorliegt. In UTF-16 gibt es nur 16 und 32 Bit lange Codeworte, bei UTF-8 können Zeichencodes aus 1 bis 4 Byte bestehen.

Beachte, dass Unicode noch nicht sicherstellt, dass die betreffenden Zeichen auf einem PC-Bildschirm auch angezeigt werden können. Dazu sind weiterhin entsprechende Font-Dateien notwendig, welche für jedes Codewort (Zeichen) eine entsprechende grafische Repräsentation enthalten.

Beispiele

- Das Zeichen *a* (Unicode 0061_h) hat in UTF-8 den Code 61_h (8 Bit), in UTF-16 den Code 0061_h (16 Bit).
- Das Zeichen *ü* (Unicode $00FC_h$) hat in UTF-8 den Code $C3BC_h$ (16 Bit), in UTF-16 den Code $00FC_h$ (16 Bit).
- Das griechische Zeichen Ω (Unicode $03A9_h$) hat in UTF-8 den Code $CEA9_h$ (16 Bit), in UTF-16 den Code $FEFF03A9_h$ (32 Bit).
- Das mathematische Zeichen $\lll$ (Unicode $22D8_h$) hat in UTF-8 den Code $E28B98_h$ (24 Bit), in UTF-16 den Code $FEFF22D8_h$ (32 Bit).

Nicht befriedigend gelöst ist die Markierung von reinen (unformatierten) Textdateien, wie zum Beispiel dem Source-Code von Programmen. Werden derartige Dateien zwischen verschiedenen Systemen ausgetauscht, so muss ein Editor beim Öffnen erkennen können, in welchem Encoding die betreffende Datei gespeichert wurde, also ob sie beispielsweise in UTF-8, UTF-16 oder gar ASCII oder ISO 8859-X codiert ist. Für UTF-16 gibt es eine spezielle Startsequenz, die unter anderem auch little- und big-endian erkenntlich macht. Diese Präambel ist aber fakultativ und fehlt oft. Daher sind gute Editoren mit ausgeklügelten Erkennungsalgorithmen ausgestattet, welche allerdings bezüglich

[8] UTF steht für *UCS Transformation Format*, und UCS für *Universal Character Set*. Die angehängte Zahl 8 oder 16 steht für die minimale Codewortbreite in Bit.

[9] Little-endian heisst, dass von mehreren zusammen gehörenden Bytes zuerst das niederwertigste übertragen oder gespeichert wird und am Ende das höchstwertige. Bei big-endian ist es umgekehrt.

[10] Man könnte auch sagen: Weniger bevorzugte Zeichen.

der verwendeten Sprache und Zeichen auf Annahmen angewiesen sind. XML und HTML Dateien enthalten oft einen Tag, der das Encoding angibt. Aber auch dieser Tag ist nur lesbar, wenn zuerst eine geeignete Wahl des Encodings getroffen wurde.

Beispiele Es wird eine Datei mit dem Inhalt *Salü* in verschiedenen Encodings erzeugt. Dann betrachten wir den Inhalt der Datei mit einem Hex-Editor. Beachte die Marker $FEFF_h$ für UTF-16 und $EFBBBF_h$ für UTF-8, die von einigen Editoren eingefügt werden.

- Notepad unter Windows in der Einstellung *Unicode* (UTF-16):
  `Datei-Inhalt (hex): ff fe 53 00 61 00 6c 00 fc 00`
- Notepad unter Windows in der Einstellung *Unicode big endian* (UTF-16):
  `Datei-Inhalt (hex): fe ff 00 53 00 61 00 6c 00 fc`
- Notepad unter Windows in der Einstellung *UTF-8*:
  `Datei-Inhalt (hex): ef bb bf 53 61 6c c3 bc`
- jEdit unter Linux in der Einstellung *UTF-16*:
  `Datei-Inhalt (hex): fe ff 00 53 00 61 00 6c 00 fc`
- jEdit unter Linux in der Einstellung *UTF-16*LE:
  `Datei-Inhalt (hex): 53 00 61 00 6c 00 fc 00`
- jEdit unter Linux in der Einstellung *UTF-8*:
  `Datei-Inhalt (hex): 53 61 6c c3 bc`
- Bash Shell unter Linux (UTF-8) mit dem Befehl *echo Salü > file*:
  `Datei-Inhalt (hex): 53 61 6c c3 bc`

9 Übungsaufgaben

9.1 Rechnen in Zahlensystemen

(a) Führen Sie die folgende Addition schriftlich und von Hand im 2-er System aus:
$-9_d + 21_d$

(b) Führen Sie die folgende Subtraktion schriftlich und von Hand im 16-er System aus:
$1101'0010'1001_b - 10'0101'1000_b$

(c) Führen Sie die folgende Multiplikaton schriftlich und von Hand im 2-er System aus:
$7_d \cdot (-11_d)$

(d) Schreiben Sie die folgenden Zahlen in Komplementdarstellung:
$-11'0111'1000_b$
-87.25_d

9.2 Endliche Zahlen

(a) Was bedeutet die folgende binäre Zahl, die in einem 8-Bit Register steht, im Dezimalformat?
$1100'1001_b$

(b) In einem 8-Bit Register wird die folgende Addition ausgeführt:
$100_d + 100_d$
Welches Problem kann auftreten?

(c) Sie subtrahieren in einem 8-Bit Register die beiden vorzeichenlosen Zahlen:
$0 - 1$
Was geschieht?

9.3 Allgemeine Codes

(a) Wie lautet die folgende Zahl in BCD-Darstellung?
110011_b

(b) Welcher Binärzahl entspricht die folgende BCD-Zahl?
$0010'1001_{BCD}$

(c) Ein Code-Rad mit 3-Bit Gray Code steht auf dem Wert 011. Wenn sich das Rad nun leicht bewegt, welche neuen Werte können auftreten?

(d) Für welches Zeichen steht der ASCII Code 100_d?