2 Zahlendarstellung in Rechnern

Wir wissen bereits aus der Digitaltechnik wie Ganzzahlen binär dargestellt (codiert) werden und können die Grundrechenoperationen ausführen. Im nachfolgenden Kapitel wird auf dieser Materie aufgebaut. Der Schwerpunkt der Betrachtung liegt darin, wie Zahlen im Computer codiert und verarbeitet werden. Im Speziellen wird die Darstellung rationaler Zahlen in Rechnern betrachtet. Diese Typen verlangen eine gänzlich andere Vorgehensweise als bei Ganzzahlen.

2.1 Darstellung von Zahlen in verschiedenen Zahlenbasen

Dieses Kapitel wird, da bereits in der Digitaltechnik als Repetition vermittelt. Es zeigt die grundsätzliche Darstellung von Zahlen in verschiedenen Zahlenbasen und geht auf die Umrechnung der Darstellung in verschiedenen Basen ein.

2.1.1 Zahlendarstellung zur Basis 10 (Dezimalsystem)

Das Dezimalsystem verkörpert das System mit dem der Mensch normalerweise rechnet. Zahlen werden im Dezimalsystem mit den Ziffern 0..9 und den Stellenwertigkeiten zu einer Potenz der Basis 10 gebildet.

Beispiel:
Die Zahl 32768 als Summe der Ziffern, die einen Koeffizient zur Potenz 10 darstellen:

$$32768 = 30000 + 2000 + 700 + 60 + 8 = 3 \cdot 10^4 + 2 \cdot 10^3 + 7 \cdot 10^2 + 6 \cdot 10^1 + 8 \cdot 10^0$$

Generell können wir also eine beliebige Zahl, gegeben durch ihre Ziffern, im Dezimalsystem als Summe ihrer Potenzen zur Basis 10 beschreiben:

$$a_n a_{n-1} \cdots a_1 a_0 = a_n 10^n + a_{n-1} 10^{n-1} + \cdots + a_1 10^1 + a_0 10^0 = \sum_{i=0}^{n} a_i 10^i \qquad (2.1)$$

Der obige Zusammenhang zeigt wie (positive) ganze Zahlen als Summe beschrieben werden. Genau gleich werden auch Dezimalbrüche beschrieben, indem die Stellen mit ihrer Wertigkeit summiert werden:

Beispiel:
Die Zahl 0.8432 als Summe Potenzen zur Basis 10 darstellen:

$$0.8243 = 8 \cdot 10^{-1} + 2 \cdot 10^{-2} + 4 \cdot 10^{-3} + 3 \cdot 10^{-4}$$

Oder allgemein kann ein Dezimalbruch in der Form geschrieben werden:

$$0.b_1 b_2 \cdots = b_1 10^{-1} + b_2 10^{-2} + \cdots = \sum_{i=1}^{\infty} b_i 10^{-i} \qquad (2.2)$$

Zu beachten ist, dass wir für irrationale und selbst für die meisten rationalen Zahlen keine exakte Darstellung mit einer endlichen Anzahl Nachkommastellen erreichen können.
Allgemein können also Dezimalzahlen als Summe ihrer Vor- und Nachkommastellen notiert werden:

$$a_n a_{n-1} \cdots a_1 a_0 . b_1 b_2 \cdots = \sum_{i=0}^{n} a_i 10^i + \sum_{i=1}^{\infty} b_i 10^{-i} \qquad (2.3)$$

Zahlendarstellung als Summe von Potenzen

2.1.2 Ganze Zahlen in beliebigen Zahlenbasen

Beim Umgang mit Rechnern sind Zahlenbasen von 2, 8, oder 16 üblich. Intern werden aber im Rechner die Zahlen immer als binäre Muster dargestellt. Die Binärdarstellung ist in Digitalrechnern üblich und erlaubt in den Bausteinen eine effiziente Verarbeitung der Zahlen. Die Basen 8 oder 16 werden vorwiegend zur Eingabe oder Anzeige verwendet, da Zahlen in diesen Basen wesentlich übersichtlicher sind als Binärzahlen.

Zu Anfangszeiten der Computertechnik wurde auch eine andere Codierung für Zahlen benutzt (3er-System), die zwar eine bessere Packungsdichte ermöglicht, aber von der Elektronik her gesehen, in keinem Verhältnis zum Ertrag steht. Erst ganz neue Ansätze greifen in der Technologie der Speicherbausteine wieder auf solche Prinzipien zurück. Ob sich dies aber durchsetzen wird, kann zum jetzigen Zeitpunkt nicht beantwortet werden.

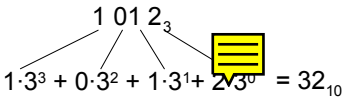
Die Wandlung (Codierung in einer anderen Zahlenbasis) beruht auf der Grundlage, dass jede Ganzzahl Z in einer beliebigen ganzzahligen Basis > 1 notiert werden kann:

$$Z = \sum_{i=0}^{n-1} b_i B^i$$

n: Anzahl Stellen
B: Basis ($\in$ N > 1)
b_i: Wertigkeit der Stelle i

Darstellung ganzer Zahlen zur Basis B (2.4)

Wir können also eine Zahl $Z=1012_3$ (die Zahl 1012, dargestellt zu Basis 3) so interpretieren:



Die Wandlung von einer Basis zur anderen wird zweckmässigerweise nach bekannter Methode der Kettendivision durchgeführt:

Bsp.: 32_{10} zur Basis 3 wandeln

32 : 3 = 10	+ Rest 2
10 : 3 = 3	+ Rest 1
3 : 3 = 1	+ Rest 0
1 : 3 = 0	+ Rest 1

Allgemein kann man das Verfahren für beliebige Zahlenbasen darstellen, wobei 'div' die ganzzahlige Division ohne Rest und 'mod' die Modulo-Divsion ist:

$$c_i = Z_i \mod B$$
$$Z_{i+1} = Z_i \operatorname{div} B$$

Z_0: Zu wandelende Zahl in beliebiger Zahlenbasis
B: Neue Zahlenbasis
c_i: Stelle der Zahl in der neuen Basis mit der Wertigkeit B^i

Kettendivision (2.5)

Besonders bequem können Zahlen gewandelt werden deren Basen Potenzen untereinander sind. Hier kann auf die Kettendivision verzichtet werden.

Durch diese Wortlänge der Speicherzelle oder auch des Prozessorregisters ist eine obere Grenze zur Darstellung einer Ganzzahl im Rechner systembedingt gegeben. Grössere Ganzzahlen kann der Rechner nicht direkt verarbeiten.

Auf dieser Grundlage erhalten folgende Wertebereiche für positive Ganzzahlen mit 8-, 16- oder 32-Bit Wortbreite:

Wortbreite	Wertebereich	Grösste darstellbare Zahl
8 Bit	$0..(2^8-1)$	255
16 Bit	$0..(2^{16}-1)$	65535
32 Bit	$0..(2^{32}-1)$	4294967295

In gewissem Umfang bieten die Prozessoren noch ein Flag, das ein Überschreiten des Bereiches um eine binäre Stelle bei einer Rechenoperation anzeigt (sog. Carry-Flag), jedoch gilt grundsätzlich, dass wir mit *n* Bits nur 2^n verschiedene Werte darstellen können.

Üblicherweise werden bei grösseren Zahlen in binärer Darstellung die Ziffern gruppiert geschrieben. Je nachdem ob man eher oktal oder hexadezimal orientiert arbeitet, schreibt man Gruppen von 3 resp. 4 binären Ziffern.

2.1.2.1 Darstellung von gebrochenen Zahlen in verschiedenen Basen

Nachkommateile können genau gleich als Summe Ihrer Stellenwertigkeiten zu einer beliebigen Basis notiert werden:

$$Z = 0.b_1 b_2 \cdots b_n = \sum_{i=1}^{n} b_i B^{-i}$$

n: Anzahl Stellen
B: Basis ($\in$ N > 1)
b_i: Wertigkeit der Stelle i

Darstellung echt gebrochener Zahlen (2.6)

Beispiel:
Wir können die Zahl $(0.C1)_{16}$ beschreiben als:

$$(0.C1)_{16} = C_{16} \cdot 16^{-1} + 1_{16} \cdot 16^{-2}$$
$$= 12 \cdot 16^{-1} + 1 \cdot 16^{-2} = \frac{12}{16} + \frac{1}{256} = \frac{193}{256} = 0.75390625$$

Die Darstellung von gebrochenen Zahlen ist in Rechnern in dieser Form nicht üblich. Gleitkommazahlen werden in einer speziellen Codierung mit Mantisse, Exponent und Vorzeichen abgespeichert.

Die Wandlung von echt gebrochenen Zahlen in eine beliebige Basis erfolgt analog der Wandlung von ganzen Zahlen, nur dass hier eine **Kettenmultiplikation** erfolgt:

Bsp.: Darstellung der Zahl $(0.C1)_{16}$ in Binärsystem:

$(0.C1)_{16} \cdot 2 = 0.75390625 \cdot 2 = 0.5078125$ + Vorkomma 1
$= 0.5078125 \cdot 2 = 0.015625$ + Vorkomma 1
$= 0.015625 \cdot 2 = 0.03125$ + Vorkomma 0
$= 0.03125 \cdot 2 = 0.0625$ + Vorkomma 0
$= 0.0625 \cdot 2 = 0.125$ + Vorkomma 0
$= 0.125 \cdot 2 = 0.25$ + Vorkomma 0
$= 0.25 \cdot 2 = 0.5$ + Vorkomma 0
$= 0.5 \cdot 2 = 0$ + Vorkomma 1

Kettenmultiplikation

$\Rightarrow (0.C1)_{16} = (0.1100\ 0001)_2$

Aus dem Beispiel kann man ersehen, dass abbrechende gebrochene Zahlen oftmals in einer anderen Basis nicht abbrechende Entwicklungen ergeben. Dies ist sogar der Regelfall, wenn Dezimalbrüche in Binärbrüche konvertiert werden. Bei beschränkter Stellenzahl können daher solche Konversionen

können nicht verlustfrei durchgeführt werden.

Gebrochene Zahlen allgemein (mit Vor- und *m*-Stellen Nachkommateil) werden gewandelt, indem man den Vorkommateil und den Nachkommateil separat wandelt und nachher zusammensetzt:

$$a_n a_{n-1} \cdots a_1 a_0 . b_1 b_2 \cdots b_m == \sum_{i=0}^{n} a_i B^i + \sum_{i=1}^{m} b_i B^{-i} \qquad (2.7)$$

Darstellung beliebiger Gleitkommazahlen

Beispiel:
Wandlung dezimal → binär

$2576.35546875 = (101\ 000\ 010\ 000.010\ 110\ 110)_2$

2.1.2.2 Wandlungen unter Zahlenbasen 2, 8,16

Besonders einfach gestalten sich Wandlungen bezüglich der Basis, wenn die eine Basis eine ganzzahlige Potenz der anderen ist. Dann können direkt eine bestimmte Anzahl Stellen zusammengefasst werden und so gruppenweise übertragen werden:

Bsp: $(0101\ 1110\ 1111)_2 \rightarrow (..)_{16}$?

$$\underbrace{(0101}_{5_{16}}\ \underbrace{1110}_{E_{16}}\ \underbrace{1111)_2}_{F_{16}}$$

Die einzelnen Konversionen werden nach der Tabelle vorgenommen:

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

Zahlen in oktaler Darstellung werden häufig bei der Systemprogrammierung von DEC-Maschinen benutzt (PDP, VAX). Hexadezimale Zahlen werden auf Systemebene in Geräten mit INTEL- oder Motorola-Prozessoren oder auch in Workstations verwendet.

2.1.3 Darstellung negativer und positiver ganzer Zahlen

Standard ist die Zweierkomplement-Darstellung für ganze Zahlen. Dabei wird der gesamte Wertebereich halbiert, indem jeweils eine Hälfte zur Darstellung positiver Zahlen und die andere Hälfte für die negativen Zahlen verwendet wird.
Die Konstruktion der Zweierkomplementcodierung für eine Zahl mit *n*-Stellen:

$$Z = -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \qquad (2.8)$$

Zweierkomplement Darstellung

Negative Zahlen werden wie folgt codiert:

$$-Z = \overline{Z} + 1 \qquad (2.9)$$

$\overline{Z}$: Einerkomplement von Z (alle b_i invertieren)
+1: Binäre Addition einer 1

Wir sehen, dass die höchste Stelle einer im Zweierkomplement codierten Zahl das Vorzeichen verkörpert. Ist die höchste Stelle eine 0 so ist die Zahl positiv, ist sie eine 1, so ist die Zahl negativ.

Beispiel: -3 soll in Zweierkomplementnotation als 16-Bit Wort Binär und Hexadezimal dargestellt werden:

$$Z = 3 = (0000\ 0000\ 0000\ 0011)_2$$
$$\overline{Z} = (1111\ 1111\ 1111\ 1100)_2$$
$$-Z = \overline{Z} + 1 = (1111\ 1111\ 1111\ 1100)_2 + 1 = (1111\ 1111\ 1111\ 1101)_2 = (FFFD)_{16}$$

2.2 Darstellung von Gleitkommazahlen

Reelle Zahlen können aufgrund der beschränkten Stellenzahl in einem Rechner nicht direkt abgespeichert werden. Man behilft sich reelle Werte zu 'rationalisieren', indem sie auf eine bestimmte Stellenzahl gekürzt werden. Man spricht hier oft von numerisch-reellen Zahlen.

Normalerweise werden positive rationale Zahlen als Verbund von Ganzzahl- und Nachkommaanteil, der durch einen Dezimalpunkt getrennt ist dargestellt. Eine andere Form ist die Exponentendarstellung. Sie wird durch eine Mantisse und Exponenten zur Basis 10 dargestellt:

Eine spezielle Art der Exponentendarstellung ist die normalisierte Darstellung (normalized scientific form):

$$32.76819 = 0.3276819 \cdot 10^2$$
$$0.0013421 = 0.13421 \cdot 10^{-2}$$
$$1782337.12 = 0.178233712 \cdot 10^7$$

Der Begriff *normalisiert* bezieht sich auf die Mantisse. Sie ist immer < 1, aber ≥ 0.1. Somit ist sie einzige normalisierte Darstellung der Zahl 7295: $0.7295 \cdot 10^5$. Die Darstellungen $0.07295 \cdot 10^6$ oder $7.295 \cdot 10^4$ sind nicht korrekt.

2.2.1 Normalisierte Darstellungen von Gleitkommazahlen

In der Informatik wird die normalisierte wissenschaftliche Darstellung als **normalisierte Gleitkommadarstellung** genannt (normalized floating-point representation).

Im Dezimalsystem kann jede reelle Zahl Z in der Form

$$Z = \pm 0.d_1 d_2 d_3 \cdots 10^n \qquad (2.10)$$

Normalisierte Gleitkommadarstellung

dargestellt werden, wobei $d_1 \neq 0$ und n ganzzahlig ist. Die Ziffern d_1, d_2, ... sind die Dezimalstellen im Wertebereich 0, ..., 9.

Verallgemeinert ist eine normalisierte Gleitkommazahl für eine beliebige reelle Zahl $Z \neq 0$:

$$Z = \pm r \cdot 10^n \qquad \left(\frac{1}{10} \leq r < 1, n \in Z\right) \tag{2.11}$$

Diese Definition besteht aus drei Teilen: Dem Vorzeichen + oder -, einer Zahl *r* im Intervall $[\frac{1}{10},1)$, sowie einer ganzzahligen Zehnerpotenz. Die Zahl *r* heisst **normalisierte Mantisse** und *n* ist der **Exponent**.

Die Darstellung von Gleitkommawerten im Binärsystem ist ähnlich wie im Dezimalsystem. Hier gilt:

$$Z = \pm q \cdot 2^m \qquad \left(\frac{1}{2} \leq q < 1, m \in Z\right) \tag{2.12}$$

Die Mantisse verkörpert hierbei eine Folge von Bits der Form $q = (0.b_1b_2b_3...)$ mit $b_1 \neq 0$. Weil $b_1 = 1$ folgt daraus dass *q* immer $\geq \frac{1}{2}$.

Eine Gleitkommazahl in einem Computer wird im Prinzip genauso gespeichert nur mit verschiedenen Einschränkungen:

Durch die Wortbreite der Speicherzelle erfolgt eine Einschränkung der maximalen Präzision der Darstellung. Somit haben numerisch reelle Zahlen in Rechnern eine **Beschränkung im Wertebereich** des Exponenten, wie auch in der Präzision der Mantisse.

Wir wissen bereits, dass keine irrationalen Zahlen mit einer beschränkten Stellenzahl exakt dargestellt werden können. Ebenso können in der Regel auch 'normale' rationale Zahlen nicht exakt in einem Computer dargestellt werden, weil viele Dezimalbrüche im Binärsystem nicht abbrechend sind.

Beispiel:
$$\frac{1}{10} = (0.1)_{10} = (0.0631463146314\ldots)_8$$
$$= (0.0\,0011\,0011\,0011\,0011\,0011\ldots)_2$$

Die meisten Zahlen in können also einem Rechner nicht exakt dargestellt werden.

Eine weitere wichtige Erkenntnis ist, dass der Zahlenbereich in einem Rechner nicht kontinuierlich ist sondern als diskrete, beschränkte Menge vorliegt. Es gibt hier eine kleinste und grösste darstellbare Zahl in einem System und die Distanz der Werte untereinander ist ungleich.

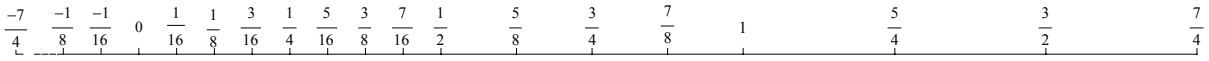
Beispiel: Zeige alle Gleitkommawerte die in der Form $z = \pm(0.b_1b_2b_3)_2 \cdot 2^{\pm m} \quad (m, b_i \in \{0,1\})$ dargestellt werden können!

Lösung:
Es existieren jeweils zwei Möglichkeiten, je nach Vorzeichen ± , zwei Möglichkeiten für b_1, zwei Möglichkeiten für b_2, zwei Möglichkeiten für b_3 und drei Möglichkeiten für den Exponenten. Eine Berechnung ergibt also $2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 48$ Möglichkeiten. Wir können also mit dieser Codierung maximal 48 Zahlenwerte darstellen, wovon allerdings gewisse Werte mehrfach erscheinen:

$$\begin{array}{lll}
0.000 \cdot 2^0 = 0 & 0.000 \cdot 2^1 = 0 & 0.000 \cdot 2^{-1} = 0 \\
0.001 \cdot 2^0 = \frac{1}{8} & 0.001 \cdot 2^1 = \frac{1}{4} & 0.001 \cdot 2^{-1} = \frac{1}{16} \\
0.010 \cdot 2^0 = \frac{2}{8} & 0.010 \cdot 2^1 = \frac{2}{4} & 0.010 \cdot 2^{-1} = \frac{2}{16} \\
0.011 \cdot 2^0 = \frac{3}{8} & 0.011 \cdot 2^1 = \frac{3}{4} & 0.011 \cdot 2^{-1} = \frac{3}{16} \\
0.100 \cdot 2^0 = \frac{4}{8} & 0.100 \cdot 2^1 = \frac{4}{4} & 0.100 \cdot 2^{-1} = \frac{4}{16} \\
0.101 \cdot 2^0 = \frac{5}{8} & 0.101 \cdot 2^1 = \frac{5}{4} & 0.101 \cdot 2^{-1} = \frac{5}{16} \\
0.110 \cdot 2^0 = \frac{6}{8} & 0.110 \cdot 2^1 = \frac{6}{4} & 0.110 \cdot 2^{-1} = \frac{6}{16} \\
0.111 \cdot 2^0 = \frac{7}{8} & 0.111 \cdot 2^1 = \frac{7}{4} & 0.111 \cdot 2^{-1} = \frac{7}{16}
\end{array}$$

Gesamthaft sind also 31 verschiedene Zahlen in dieser Codierung darstellbar. Die positiven Werte sind in obiger Tabelle aufgeführt.

Obwohl die Zahlen symmetrisch um den Nullpunkt liegen sind sie nicht gleichmässig verteilt, d.h. äquidistant gelegt:



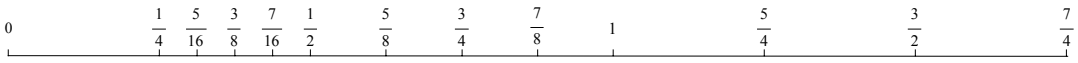
Nimmt bei einer Berechnung der Wert einer Zahl der Form $\pm q \cdot 2^m$ ausserhalb des darstellbaren Bereiches an, so sprechen wir von einem Überlauf (overflow) oder Unterlauf (underflow). Die Zahl ist in dieser Form nicht darstellbar und somit ungültig. Normalerweise werden in einem Rechner Überläufe mit einer Fehlermeldung quittiert und die laufende Berechnung abgebrochen. Bei einem Unterlauf wird bei den meisten Systemen die Zahl automatisch auf Null gesetzt, ohne irgendwelche Fehlermeldung zu erzeugen.

In unserem Beispiel bewirken alle Zahlen kleiner als $\left|\frac{1}{16}\right|$ einen Unterlauf zu Null. Alle Zahlen ausserhalb des Bereiches -1.75..+1.75 erzeugen einen Überlauf und werden in diesem Rechner als 'Unendlich' interpretiert (machine infinity).

Wenn wir nun ausschliesslich normalisierte Gleitpunktdarstellung erlauben, haben alle Zahlen (ausser der 0) in dieser Codierung die Form:

$$z = \pm(0.1b_2b_3)_2 \cdot 2^{\pm m}$$

Wir erhalten bei dieser Darstellung eine relativ grosse Lücke bei Null. Unsere darstellbaren Zahlen sind nun, wie folgt, verteilt (Betrachtung nur für positive Zahlen, für negative Zahlen symmetrisch):



Die kleinste darstellbare positive Zahl in diesem System ist also:

$$(0.100)_2 \cdot 2^{-1} = \frac{1}{4}$$

Diese auf den ersten Blick unerträgliche Genauigkeit dieser Codierung stellt in der Praxis in der Regel kein Problem dar, da die tatsächlich benutzte Codierung für Gleitkommazahlen noch weiter verfeinert ist. Der Grund wieso man mit normalisierten Darstellungen arbeitet, ist die enorme Vereinfachung der Vorschriften für die Rechenoperationen.

Tatsache ist, dass selbst mit normalen Numerikprozessoren keine bessere Genauigkeit als 10^{-15} erreicht wird.

2.2.2 Gleitkommazahlen in IEEE-754-Codierung

Standardmässig werden Gleitkommazahlen in Computer nach IEEE-754 codiert. Dieses Format wird von den Numerikprozessoren (8x87) benutzt, wie auch von Softwarealgorithmen[1].

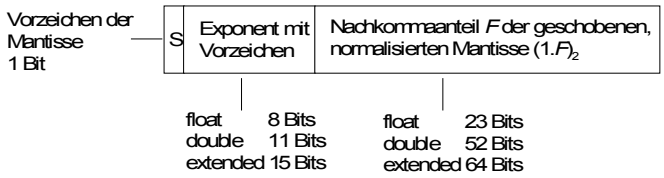
Bis zum Ende der 70er-Jahre benutzte jeder Rechnerhersteller seine eigene Codierung für Gleitkommazahlen. Erst in der Mitte der 80er-Jahre wurde von IEEE (Institute of Electrical and Electronics Engineers) die erste Norm für eine binäre Gleitkommaarithmetik erstellt. Sie legt die Codierung, Rundungsvorschriften, sowie die Behandlung von Ausnahmesituationen (Überlauf, Division durch 0, etc.) fest. Ausserdem gibt sie einen Grundstock an mathematischen Funktionen vor, die mit den Zahlen in dieser Codierung durchführbar sein müssen.

Wir werden sehen, dass die Umrechnungen von Hand recht mühsam werden. Glücklicherweise muss man sich als Programmierer praktisch nie mit Konversionen dieser Art befassen. Jedoch gehört dieser Stoff zu einer seriösen Grundausbildung zum Thema 'Zahlendarstellung in Rechnern'.

2.2.2.1 Prinzipien

Das Prinzip der Codierung ist hier, dass Gleitkommazahlen in Mantisse, Vorzeichen und Exponent zerlegt, gewandelt und in einer bestimmten Vorschrift zusammengepackt werden. Diese gepackte Information wird dann in einem Feld von 32, 64 oder 80 Bits abgespeichert, je nach Codierungsvorschrift.

Bei der Codierung muss ein Kompromiss eingegangen werden, um in einer bestimmten Wortbreite (32, 64 oder 80 Bits) einen möglichst grossen Wertebereich mit einer guten Auflösung zu codieren. Der IEEE Standard bietet hier eine gute Lösung dieser Forderungen. Gleitkommazahlen werden nach folgender Konvention codiert:



Eine Zahl des Typs float in C wird in einer Wortbreite von 32 Bits abgelegt.

[1] Algorithmus:
Begriff aus der Informatik. Ein Algorithmus beschreibt klar (rezeptual oder formal) die Vorschrift wie eine bestimmte Aufgabe zu bewältigen ist. Aus dem Algorithmus geht demzufolge eindeutig die Abfolge der einzelnen Verarbeitungsschritte hervor. Er beschreibt wie schrittweise die Eingangsdaten zu Ausgangsdaten umgewandelt werden.

Die sog. **1-plus-Form** entsteht durch Linksschieben der normalisierten Mantisse um eine Stelle:

$$(0.1b_2b_3\cdots b_m)_2 \Rightarrow (1.b_2b_3\cdots b_m)_2 = (1.F)_2 \qquad \textbf{1-plus-Form} \qquad (2.13)$$

Die 1-plus-Form wird mit 24 Bits Genauigkeit entwickelt und davon werden die hinteren 23 Bits (=*F*) abgespeichert . Die vorderste Stelle der Mantisse muss nicht gespeichert werden, da sie aufgrund der Normalisierung sowieso immer 1 ist. Das Vorzeichen der Mantisse wird im höchsten Bit abgelegt. Es hat den Wert 0, falls die Zahl positiv ist.

Die Mantisse hat für die verschiedenen Codierungen folgende Genauigkeiten (mit der wichtigen Ausnahme der Zahl Null):

24 Bits für float mit 32 Bit Wortbreite
54 Bits für double mit 64 Bit Wortbreite
64 Bits für extended mit 80 Bit Wortbreite

Für eine `float`-codierte Zahl mit 24-Bit Mantisse wird $2^{-24} \approx 0.596 \cdot 10^{-7}$ und finden dadurch, dass diese Codierung etwa 7 signifikante Dezimalziffern Genauigkeit bietet. Eine double-codierte Zahl bietet demnach etwa 14 signifikante Dezimalstellen.

Formal lautet die Codierungsvorschrift:

$$z = (-1)^S \cdot 2^{E-B} \cdot (1.F)_2 \qquad \textbf{IEEE-754 Codierung} \qquad (2.14)$$

S: Vorzeichen der Mantisse (positiv: 0, negativ: 1)
E: Zu speichernder Exponent
B: 127 für float (32-Bit Zahl), 1023 für double (64-Bit Zahl), 16383 für extended (80-Bit Zahl)
F: Nachkommaanteil der 1 - plus Form

Das Vorzeichen der Mantisse wird immer im höchstwertigen Bit gespeichert. Das nächste Feld enthält den im Excess-B-Code codierten Exponenten, wobei *B* je nach Wortbreite eine Zahl aus 127, 1023, 16383 ist. Durch die Excess-Codierung wird das Vorzeichen des Exponenten in den Exponent selbst eincodiert. Für eine float mit 32 Bit Wortbreite wird der Exponent im Excess-127-Code notiert.

Wir betrachten am Beispiel der float-Codierung (32-Bit) die grösste darstellbare Zahl:
Für 32-Bit Codierung wird B=127. Der mögliche Wertebereich des Exponenten E ist gegeben durch die Feldbreite von 8 Bit. Das heisst, dass der Excess-127-codierte Exponent 256 Werte annehmen kann. Durch das Biasing (dt. 'Vorspannung') wird der Bereich:

$$-127 \le E - 127 \le 128$$

Ebenso können wir den Bereich der Mantisse bestimmen:

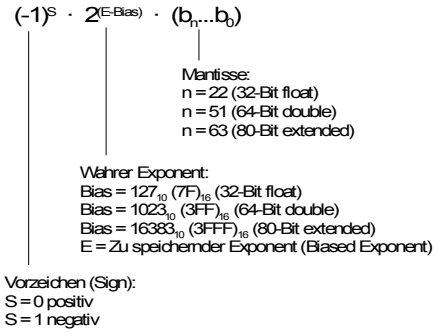
$$1 \le (1.F)_2 \le (1.11\ldots1)_2 = 2 - 2^{-23}$$

Wir erhalten somit als betragsmässig kleinste (ausser 0) und grösste darstellbare Zahl:

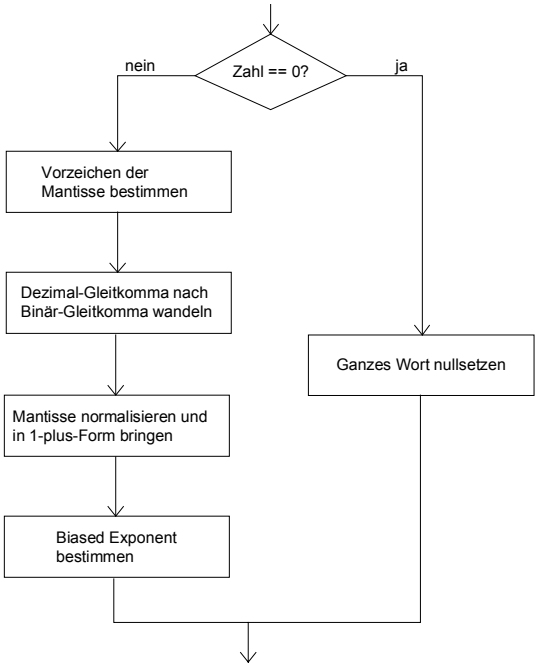
$$z_{\min} = (-1)^S \cdot 2^{-127}(1.00\ldots0)_2 = \pm 2^{-127} \qquad \approx \pm 5.87 \cdot 10^{-39}$$
$$z_{\max} = (-1)^S \cdot 2^{128}(1.11\ldots1)_2 = \pm 2^{128}\left(2 - 2^{-23}\right) \approx \pm 2^{128} \cdot 2 = \pm 2^{129} \quad \approx \pm 6.81 \cdot 10^{38}$$

2.2.2.2 Konversion nach IEEE-754

Die nachfolgenden Beispiele zeigen, wie dezimal codierte Zahlen in IEEE-Binärcodierte Zahlen umgerechnet werden und umgekehrt. IEEE-754-codierten Zahlen heissen häufig auch **Maschinenzahlen** (machine numbers). Das Vorgehen beruht auf Definition der Zahlendarstellung gemäss Seite 2-8:



Die Konversion einer Dezimalzahl in eine Maschinenzahl erfolgt nach dem Flussdiagramm[2]. Das prinzipielle Vorgehen ist für alle drei Formate gleich:



Die Rückwandlung einer Maschinenzahl in eine Dezimalzahl erfolgt indem das Diagramm sinngemäss rückwärts durchlaufen wird.

Beispiel 1: Wie lautet die 32-Bit-IEEE-Codierung für -52.234375 ausgedrückt in Hex-Zahlen?
Lösung: Die Zahl ist ≠0, also bestimmen wir das Vorzeichen der Zahl. Es liefert das Vorzeichenbit (höchstwertiges Bit D_{31}):

$$\text{sgn}(-52.234375) = -1 \quad \Rightarrow \quad D_{31} = 1$$

Als nächstes wird die Dezimal-Binärwandlung der Zahl durchgeführt. Diese erfolgt in bekannter Manier, getrennt für Ganzzahl und Nachkommateil:

[2] Flussdiagramm:
Flussdiagramme werden in der Informatik verwendet um den (Programm-)Ablauf aufzuzeigen. Die wesentlichen Elemente sind die Aktionskästchen (Rechteck) und die bedingte Verzweigung (Raute). Flussdiagramme sind aus der reinen Informatiklehre her, nicht besonders gern gesehen, da selbst der grösste Unsinn irgendwie visualisiert werden kann. Tatsache ist aber, dass diese Diagramme weltweit verstanden werden und DIN-genormt sind.

$(52)_{10} = (110100)_2$
$(0.234375)_{10} = (0.001111)_2$
$\Rightarrow (52.234375)_{10} = (110100.001111)_2$

Wir normalisieren die Zahl indem wir sie in die '1-plus-Form' bringen. Wir erhalten eine Mantisse die mit 1.xx.. beginnt und einen Exponenten zur Basis 2:

$$(52.234375)_{10} = (110100.001111)_2 = (1.10100001111)_2 \cdot 2^5$$

$(10100001111)_2$ ist nun die zu speichernde Mantisse in den Stellen $D_{22}..D_{12}$. Die restlichen Stellen werden mit 0 aufgefüllt.

Der wahre Exponent der Zahl ist $(5)_{10}$. Er wird nach der Vorschrift mit dem Bias verrechnet und wir erhalten den zu speichernden Exponenten (Biased Exponent):

$$e = E - Bias = E - 127 = 5$$

$$\Rightarrow E = 127 + 5 = 132 = (1000\,0100)_2$$

E: Zu speichernder Exponent (Biased Exponent)
e: Wahrer Exponent

Dieser Exponent wird in den Bits $D_{30}..D_{23}$ gespeichert. Wir setzen nun alles zusammen und erhalten die IEEE-Binärzahl:

$$[1100\ 0010\ 0101\ 0000\ 1111\ 0000\ 0000\ 0000]_2 = \underline{\underline{[C250F000]_{16}}}$$

Beispiel 2: Welche Dezimalzahlen verkörpern die nachfolgenden Maschinenzahlen in IEEE-Codierung?

$[45DE4000]_{16}$ $[BA390000]_{16}$

Lösung: Die erste Zahl $[45DE4000]_{16}$ ist in Binärdarstellung:

$[0100\ 0101\ 1101\ 1110\ 0100\ 0000\ 0000\ 0000]_2$

Wir extrahieren daraus:
Vorzeichen (D_{31}): 0
Biased Exponent ($D_{30}..D_{23}$): $(1000\ 1011)_2 = 08B_{16} = 139_{10}$
Mantisse: ($D_{22}..D_0$): $(101\ 1110\ 0100\ 0000\ 0000\ 0000)_2$
$\Rightarrow$ 1-plus-Form: $(1.101\ 1110\ 0100)_2$
Das Vorzeichenbit der Zahl ist 0 $\Rightarrow$ Zahl ist positiv.
Der wahre Exponent wird: $e = (E - Bias) = 139 - 127 = 12$
Mit der Mantisse wird der Wert der gesamten Zahl:

$$(1.101\ 1110\ 0100)_2 \cdot 2^{12} = (1\ 1011\ 1100\ 1000.0)_2 = (1BC8)_{16} = \underline{\underline{7112_{10}}}$$

Die zweite Zahl $[BA390000]_{16}$ ergibt:

Binärdarstellung: $[1011\ 1010\ 0011\ 1001\ 0000\ 0000\ 0000\ 0000]_2$
Vorzeichen: 1
Biased Exponent: $(0111\ 0100)_2$
Mantisse: $(011\ 1001\ 0000\ 0000\ 0000\ 0000)_2$
$\Rightarrow$ 1-plus Form: $(1.0111\ 0010)_2$

Das Vorzeichenbit ist 1 daraus folgt, dass die Zahl negativ ist.
Wir erhalten den wahren Exponenten: $e = (E - Bias) = 116 - 127 = -11$
Die gesamte Zahl wird:

$$(1.0111\ 0010)_2 \cdot 2^{-11} = -(0.0000000001011100100)_2$$
$$= -(0.002E4)_{16} = -\left(2 \cdot 16^{-3} + 14 \cdot 16^{-4} + 4 \cdot 16^{-5}\right) \approx \underline{\underline{-7.0571894 \cdot 10^{-4}}}$$

2.3 Fehlerbetrachtung für Gleitkommadarstellungen

Den Fall, dass die Zahl nicht als Maschinenzahl in einer der entsprechenden IEEE-Formate aufgrund von Unterlauf oder Überlauf dargestellt werden kann , wollen wir hier nicht betrachten. Dieser Abschnitt soll zeigen, wie gross die zu erwartenden Fehler in der Darstellung einzelner Zahlen werden.

2.3.1 Präzision der Darstellung

Wir machen eine Betrachtung für positive in 32-Bit IEEE-Codierung. Für negative Zahlen und andere Formate erfolgt das Vorgehen analog.

Eine Zahl Z kann in normalisierter Form allgemein dargestellt werden:

$$Z = q \cdot 2^m \qquad \left(\tfrac{1}{2} \le q < 1, |m| \le Bias = 127\right) \tag{2.15}$$

Durch Rundung kann die nächste Zahl gefunden werden, die effektiv in diesem Code darstellbar ist. Uns interessiert wie gross nun dieser hierbei auftretende Rundungsfehler wird und stellen z in normalisierter binärer Notation exakt dar :

$$Z = (0.1 b_2 b_3 b_4 \ldots b_{24} b_{25} \ldots)_2 \cdot 2^m \tag{2.16}$$

Die nächste Maschinenzahl (d.h. Zahl welche in diesem Code darstellbar ist) muss der Codierungsvorschrift bezüglich Stellenzahl genügen. Die darzustellende Zahl z liegt in einem Intervall [z', z''], das durch die Genauigkeit der Mantisse bestimmt wird. Die darzustellende Zahl wird durch Rundung an das untere oder obere Ende des Intervalls gelegt und erhalten betragsmässig den relativen Fehler:

$$\left|\frac{z - z'}{z}\right| \le \frac{2^{-25+m}}{(0.1 b_2 b_3 \cdots)_2 \cdot 2^m} \le \frac{2^{-25}}{\frac{1}{2}} = 2^{-24} \tag{2.17}$$

Begründung:
Das zu betrachtende Intervall [z',z''] ergibt sich:

$z' = (0.1 b_2 b_3 b_4 \ldots b_{24})_2 \cdot 2^m$
$z'' = (0.1 b_2 b_3 b_4 \ldots b_{24})_2 \cdot 2^m + 2^{-24}$

Bei der Betrachtung können zwei Fälle auftreten: z liegt näher bei z' als bei z''. Dann gilt die Ungleichung:

$$|z - z'| \le \frac{1}{2}|z'' - z'| \le 2^{-25+m}$$

Der maximale relative Fehler ergibt sich bei $(0.1 b_2 b_3 \cdots)_2 \to (0.1)_2$. Man erhält mit (2.17):

$$\left|\frac{z - z'}{z}\right| \le \frac{2^{-25+m}}{(0.1 b_2 b_3 \cdots)_2 \cdot 2^m} = \frac{2^{-25} \cdot 2^m}{(0.1 b_2 b_3 \cdots)_2 \cdot 2^m} = \frac{2^{-25}}{\frac{1}{2}} = 2^{-24}$$

Die Betrachtung wenn z näher bei z'' liegt erfolgt analog.

Wir sehen, der relative Fehler dieser Darstellung ist nie grösser als 2^{-24}. Damit ergibt sich die Anzahl der signifikanten Dezimalstellen:

$$2^{-24} = 0.596 \cdot 10^{-7}$$

Unter Betrachtung der Rundung sind hier also 7 Dezimalstellen signifikant und $0.596 \cdot 10^{-7}$ verkörpert den maximalen Rundungsfehler dieser Codierung.

2.3.2 Fehlerfortpflanzung

Besonderes Gewicht erhält bei der Rechnung mit Gleitkommazahlen die Betrachtung der Fehlerfortpflanzung. Da prinzipbedingt (fast) alle Zahlen mit einem gewissen Fehler behaftet sind, ist es notwendig kurz zu betrachten, wie sich diese Rundungsfehler auswirken. Dass diese Betrachtung gerechtfertigt ist, soll folgendes Beispiel zeigen:

Wir addieren zur Zahl 1.0 in einer Schleife immer den konstanten Wert 1.0. Das Abbruchkriterium für die Schleife soll sein, sobald die Zahl den Wert von 10000 erreicht hat, also 9999 mal durchlaufen wurde. Das Abbruchkriterium ist durch einen Vergleich der Zahl mit dem Wert 10000 zu testen. Wie gross muss das Testintervall mindestens sein?

oder:
Ergibt $10000 \cdot 1$ gleichviel wie 10000 mal 1 zu addieren?

Oft kann durch intelligente Formulierung, geschickte Wahl der Datentypen oder Operationen der Fehler stark vermindert werden.

Ein anderes Phänomen ist die Stellenauslöschung bei arithmetischen Operationen. Sie tritt immer dann auf, wenn grosse Werte mit kleinen Werten verrechnet werden. In solchen Fällen verlieren wir signifikante Stellen.

Beispiel:
Wir haben eine Maschine die mit einer Präzision fünf Dezimalstellen rechnet und addieren zwei Zahlen *x* und *y,* die in normalisierter Form vorliegen:

$$x = 0.12789 \cdot 10^{4} \qquad y = 0.51345 \cdot 10^{-1}$$

Wir gehen ferner davon aus, dass die Rechnung mit doppelter Genauigkeit ausgeführt wird, dass heisst, die Rechnung selbst wird mit 10 Stellen durchgeführt und erst das Resultat der Rechnung wird wieder auf fünf Stellen gerundet. Dieses Prinzip wird übrigens in Rechnern durchwegs so angewandt. So werden in C alle Rechnungen des Typs float intern mit `double` durchgeführt und `double` werden intern mit long double verrechnet. Somit wird unsere Addition intern:

$$\begin{array}{r} x = 0.12789\,00000 \cdot 10^{4} \\ y = 0.00000\,51345 \cdot 10^{4} \\ \hline x + y = 0.12789\,51345 \cdot 10^{4} \end{array}$$

Die nächste Maschinenzahl zur Aufnahme des Resultates ist z=$0.12790 \cdot 10^{4}$ (z auf fünf Stellen gerundet). Wir können nun den relativen Fehler dieser Addition bestimmen:

$$\frac{|x + y - z|}{|x + y|} = \frac{0.48655 \cdot 10^{-2}}{0.12789\,81345 \cdot 10^{4}} \approx 3.804 \cdot 10^{-5}$$

Betrachtet man die Vorgabe der Präzision von fünf Stellen, kann die Genauigkeit als genügend angesehen werden, obwohl das Zahlenbeispiel bewusst so gewählt worden ist, dass der Fehler recht gross wird.

Betrachten wir nun konkret die Fehlerfortpflanzung mit den uns bekannten Mitteln der Fehlerrechnung, so erhalten wir für die Addition von zwei Gleitkommazahlen in 32-Bit IEEE-Codierung:

$$\Delta\tilde{x}, \Delta\tilde{y} = 2^{-24}$$
$$\Delta\tilde{z} = \Delta\tilde{x} + \Delta\tilde{y} = 2^{-24} + 2^{-24} = 2 \cdot 2^{-24}$$
$$\tilde{z} \in [z - \Delta\tilde{z}, z + \Delta\tilde{z}]$$

Allgemein kann mit der Methode der Fehlerschranken die Fehlerfortpflanzung bestimmt werden.

2.3.2.1 Verlust an Präzision

Nicht auf den ersten Blick ersichtlich ist das Problem von Verlust an Präzision bei der Subtraktion wenn zwei Zahlen nahe beieinander liegen, also fast gleich gross sind. Die 'Distanz' der beiden Zahlen können wir durch eine Messvorschrift (sog. Metrik) beschreiben: $\left|1-\frac{y}{x}\right|$. Beachten Sie: Die | | -Klammern haben nichts mit einer Betragsbildung zu tun, sondern bezeichnen die Metrik!

Satz: (ohne Beweis)

> Sind x und y normalisierte Gleitkomma-Maschinenzahlen, so dass $x > y > 0$. Wenn $2^{-p} \le 1 - \frac{y}{x} \le 2^{-q}$ für bestimmte positive Ganzzahlen p und q, dann werden bei der Subtraktion x-y höchstens q-Stellen, aber mindestens p-Stellen an signifikanten binären Bits eingebüsst.

Beispiel: Wie viele signifikante Stellen werden bei der Subtraktion 57.383716 - 57.394116 eingebüsst?

Lösung: Wir bestimmen die Distanz der beiden Zahlen, setzen die Ungleichung des obigen Satzes ein und bestimmen die nächsten p und q so, dass die Ungleichung noch gilt:

$$x = 57.383716 \qquad y = 57.394116$$
$$1 - \frac{y}{x} = 1 - \frac{57.383716}{57.394116} = 0.000181203244$$

Dieser Wert liegt zwischen $2^{-12} = 0.000244$ und $2^{-13} = 0.000122$. Somit werden mindestens 12, aber nicht mehr als 13 Bits an Präzision eingebüsst.

2.3.2.2 Methoden zur Vermeidung von Verlust an Präzision

Nun betrachten wir Massnahmen wie man den Genauigkeitsverlust vermindern kann. Es kann jedoch kein allgemein gültiges Rezept zum Vorgehen gegeben werden. Das Prinzip ist jedoch immer dasselbe: Die 'kritischen' (und nur die kritischen) Subtraktionen müssen durch geeignete Umformungen weggeschafft werden. Das Resultat ist oftmals ein wesentlich komplizierterer Ausdruck. So fällt es manchmal schwer zu glauben, dass die Umformung tatsächlich eine Verbesserung bringt.

Wir zeigen das Vorgehen anhand von einigen Beispielen:

Beispiel 1
Wir untersuchen dazu die Funktion

$$f(x) = \sqrt{x^2 + 1} - 1$$

mit Werten für x nahe bei Null so, dass das Resultat der Rechnung $\sqrt{x^2+1} \approx 1$. Wenn $x \approx 0$ sehen wir, dass die Bedingung für den Genauigkeitsverlust gegeben ist (Subtraktion zweier fast gleich grosser Zahlen). Die Lösung des Problems besteht darin, dass die Funktion algebraisch so weit umgeformt wird, dass keine solchen 'kritischen' Subtraktionen mehr vorkommen. Eine Möglichkeit ist die Funktion als rationale Funktion zu formulieren, indem geeignet erweitert wird:

$$f(x) = \sqrt{x^2+1} - 1 = \left(\sqrt{x^2+1} - 1\right)\left(\frac{\sqrt{x^2+1}+1}{\sqrt{x^2+1}+1}\right) = \frac{x^2}{\sqrt{x^2+1}+1}$$

Den Erfolg der Rechnung prüfen wir mit x=10^{-6} . Die Rechnungen werden mit dem HP48 durchgeführt. Wir erhalten im ersten Fall das falsche Resultat 0 und im zweiten Fall den korrekten Wert von $0.5 \cdot 10^{-12}$.

Beispiel 2
In einem Programm wird folgende Berechnung durchgeführt:

$$y = \cos^2(x) - \sin^2(x)$$

Beurteilen Sie, ob diese Berechnung für bestimmte x kritisch werden kann und machen Sie falls ja einen Vorschlag zur Verbesserung der Genauigkeit!

Lösung:
Für $x \approx \frac{\pi}{4}$ wird die Subtraktion $\cos^2(x) - \sin^2(x)$ kritisch, da beide Zahlen etwa gleich gross sind. Durch die trigonometrische Identität: $\cos(2\varphi) = \cos^2\varphi - \sin^2\varphi$ können wir die Genauigkeit der Rechnung verbessern:

$$y = \cos(2x)$$

Beispiel 3
Beurteilen Sie, ob nachfolgende Subtraktion kritisch ist!

$$y = \ln(x) - 1$$

Man sieht sofort, dass für $x \approx e$ die Subtraktion kritisch wird und mit Genauigkeitsverlust verbunden ist. Unter Verwendung der Rechenregeln für Logarithmen können wir die Gleichung umformen:

$$f(x) = \ln(x) - 1 = \ln\left(\frac{x}{e}\right)$$

2.4 Aufgaben

Zahlenbasen:

1. Stellen Sie folgende Zahlen in den verlangten Basen dar:
   1714_{10} → $(\ )_2$
   1714_8 → $(\ )_{10}$
   324_{10} → $(\ )_5$
   1000_{10} → $(\ )_{18}$
   1000_{24} → $(\ )_{15}$

2. Stellen Sie folgende Zahlen in den verlangten Basen dar, unter Ausnutzungen der Dualität der Basen:
   101011101111_2 → $(\ )_8$
   101011101111_2 → $(\ )_{16}$
   874_{16} → $(\ )_2$
   874_{16} → $(\ )_8$

3. Programmieren Sie das Verfahren der Kettendivision als kleines C- oder Taschenrechner-Programm.

4. Wandeln Sie die Zahl 0.4 zuerst nach hexadezimal und dann nach binär. Überprüfen Sie das Resultat, indem Sie nachher direkt nach binär wandeln.

5. Zeigen Sie, dass 1/5 nicht als endlicher Ausdruck im Binärsystem dargestellt werden kann.

6. Welches Resultat liefert der Computer wenn er 3 * 1/3 rechnet? Wie sieht es für 2 * ½ oder 10* 1/10 aus?

Darstellung negativer Zahlen:

7. Welche anderen Verfahren, ausser der Zweierkomplementcodierung, wären zur Darstellung negativer Zahlen denkbar? Beurteilen Sie diese Verfahren indem Sie Forderungen aufstellen (Komplexität des Rechenwerkes für +/-, Eindeutigkeit des Nullelementes, etc.) und prüfen Sie, inwieweit diese von den Methoden erfüllt werden.

8. Wie gross ist der Wertebereich für Zweierkomplement-codierte Zahlen mit der Wortbreite?
   a.) 8 Bit
   b.) 16 Bit
   c.) 32 Bit
   d.) 64 Bit

9. Stellen Sie folgende Zahlen im Zweierkomplement dar bezüglich 8 Bit Wortbreite:
   a.) 12 b.) -24
   c.) -1 d.) $(-7A)_{16}$
   e.) $(2G)_{18}$ f.) $(-2G)_{18}$

10. Geben Sie den dezimalen Wert der im Zweierkomplement codierten Zahlen an!
    a.) $(0080)_{16}$ b.) $(80)_{16}$
    c.) $(FEDF)_{16}$ d.) $(8713)_{16}$
    e.) $(00128)_{16}$ f.) $(1000\ 0000)_2$
    g.) $(0100\ 0001)_2$ h.) $(1111\ 1111\ 1111\ 1110)_2$

11. Wie würde -2 als vierstellige Ternärzahl aussehen? Überprüfen Sie, ob mit dieser Codierung 2 +(-2) tatsächlich Null ergibt!

IEEE-754 Codierung:

12. Bestimmen Sie durch Rechnung die Genauigkeitsbereiche für die Gleitkommadatentypen der Sprache C. Beachten Sie, dass diese Werte unter Berücksichtigung der Rundung entstehen!

13. Bestimmen Sie die Dezimalwerte der nachfolgenden IEEE-codierten Zahlen:
    a.) $[42CAE000]_{16}$
    b.) $[B4B24B7A]_{16}$
    c.) $[4B092D30]_{16}$

14. Bestimmen Sie die 32-Bit IEEE-Codierung der nachfolgenden Dezimalzahlen:
    a.) 1.0, -1.0
    b.) +0.0, -0.0
    c.) -9876.54321
    d.) 0.23475
    e.) 10^{-3}

15. Zählen Sie alle Werte auf die mit der einfachen Gleitkommacodierung der Form $\pm(0.b_1b_3) \cdot 2^m$ möglich sind, wobei *m*:
    a.) $m \in \{0,1\}$
    b.) $m \in \{-1,1\}$
    c.) $m \in \{-1,0,1\}$

Fehlerfortpflanzung:

16. Wie sähe das Resultat und Fehler der Addition *x+y* mit

$$x = 0.12789 \cdot 10^4 \qquad y = 0.51345 \cdot 10^{-1}$$

aus, wenn man durchwegs mit 5 Dezimalstellen Genauigkeit rechnet?

17. In welchem Bereich liegt das Resultat der Rechnungen (32-Bit IEEE-Codierung)?
    a.) $z = \sum_{1}^{10000} 1.0$
    b.) $z = 10000.0 \cdot 1.0$
    c.) $z = \sqrt[3]{3.0 \cdot x^2 + y}$