Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 5 – Codierung reeller Zahlen

Inhaltsverzeichnis

Version 20201120-2255

Im Kapitel 3 wurde kurz die Darstellung reeller Zahlen besprochen, genauer gesagt die Gleitkommadarstellung.

$$r = m \cdot B^e$$

Jedoch ist die kennengelernte Darstellung in der Realität nicht praktikabel, da der Exponent e sowohl eine positive, als auch negative Zahl sein kann. Das führt dazu, dass für die Implementierung der Arithmetik (also Addition, Multiplikation, etc.) aufwendiger ist.

Um sich nun diesen zusätzlichen Aufwand zu sparen, wurde die Gleitkomma*codierung* eingeführt. Dafür benötigt man jedoch noch zwei weitere Codierungen: Die *Exzeß-q-Codierung* und die *Festkommacodierung*.

5.1 Exzeß-q-Codierung ganzer Zahlen

Bei der Exzeß-q-Codierung wird die gespeicherte Ganzzahl um einen konstanten Wert gegenüber der Binärcodierung verschoben. Dadurch sind die Zahlenbereiche für positive und negative nicht unbedingt gleich groß (wie bei der 1-Komplementcodierung bzw. 2-Komplementcodierung)[1]. Außerdem muss keine Fallunterscheidung zwischen positiven (so nehmen wie sie sind) und negativen (Bits kippen) Zahlen vorgenommen werden. Die Verschiebung um einen konstanten Wert lässt sich fest in der Hardware verdrahten.

5.1.1 Was ist die Exzeß-q-Codierung?

Formal ist die Exzeß-q-Codierung wie folgt festgelegt:

> Definition: 5.1 Exzeß-q-Codierung
>
> Die **Exzeß-q-Codierung** (**EX-q-Codierung**)
> $c_{EX-q,n} : \{-q, ..., 0, 1, ..., 2^n - 1 - q\} \to \mathbb{B}^n$
> zu einer Zahl $q \in \mathbb{N}_0$ ist definiert durch
>
> $$c_{EX-q,n}(x) := c_{2,n}(x + q)$$

Die Definition beschreibt also Folgendes:
Um eine Zahl x in das Exzeß-q Format zu konvertieren, muss zu x eine natürliche Zahl q hinzu addiert werden. Wenn beispielsweise die Exzeß-5 Codierung benötigt wird, addiert man zu x den konstanten Wert 5.
Die neue Zahl wird „ganz normal“ binär codiert und mit 0 aufgefüllt, bis sie eine Länge von n Bits besitzt. Die Exzeß-q-Codierung bezeichnen wir dann als $c_{EX-q,n}$, wobei n die Anzahl der Bits angibt und q den Wert, um den verschoben wird, z.B. $c_{EX-5,n}$ für eine 8-Bit-Zahl mit einer konstanten Verschiebung um 5.

[1] Wobei die Exzeß-q-Codierung in unseren Fällen immer so gewählt wird, dass die Bereiche für positive und negative Zahlen weiterhin ungefähr gleich groß sind.

Beispiel:

Man möchte die Zahl 4 in die Exzeß-5 Darstellung mit 8 Bits bringen. Das bedeutet: $c_{EX-5,8}(4)$

Dazu addiert man zuerst die 4 zur 5 ($4 + 5 = 9$):

$$\begin{aligned} c_{EX-5,8}(4) &= c_{2,8}(5+4) \\ &= c_{2,8}(9) \end{aligned}$$

Hierbei ist zu beachten: Aus der Exzeß-q Codierung wurde eine Binärcodierung (zu erkennen am Wechsel von $c_{EX-5,8}$ hin zu $c_{2,8}$)).

Nun kann man die 9 binär codieren:

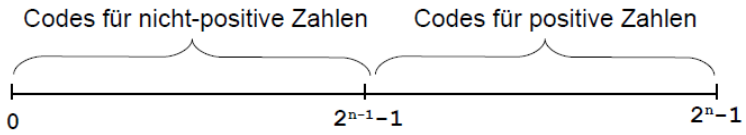
$$(9)_{10} = (1001)_2$$

und da 8 Bit gefordert sind, füllt man die Binärzahl mit 0 auf, bis man auf 8 Bit kommt: $(0000\,1001)_2$

Das ist nun die fertige Codierung.

$$c_{EX-5,8}(4) = (0000\,1001)_2$$

Die Exzeß-q-Codierung wird in der Praxis mit $q = 2^{n-1} - 1$ eingesetzt zur **Codierung des Exponenten einer normierten Gleitkommazahl**:



5.1.2 Decodierung und Beispiele

Decodierung: Es gilt (ohne Beweis, da einfache Rechnung)

$$(b_{n-1}...b_1b_0)_{EX-q,n} = (b_{n-1}...b_1b_0)_{2,n} - q = (\sum_{i=0}^{n-1} b_i \cdot 2^i) - q$$

Berechnungsschema für $c_{EX-127,8}$:

q=127	128	64	32	16	8	4	2	1	2er-Potenzen / q
-1	0	0	1	0	0	1	1	1	Ziffern

Informell bedeutet das nun:
Man decodiert „ganz normal“ die Binärzahl und subtrahiert davon den Wert q.

Beispiel:

Man will nun die Zahl $(0000\,1101)_{\mathrm{EX}-4,8}$ decodieren.
Dazu decodiert man zuerst $(0000\,1101)$:

$$(0000\,1101)_2 = (13)_{10}$$

und zieht von diesem Wert q ab, also den Wert 4:

$$(13)_{10} - (4)_{10} = (9)_{10}$$

Und fertig ist die Decodierung:

$$(0000\,1101)_{\mathrm{EX}-4,8} = (9)_{10}$$

Beispiel:

Hier noch einige Beispiele zum selbstständig nachrechnen:
Codierung:

$$c_{\mathrm{EX}-127,8}(7) = c_{2,8}(7 + 127) = c_{2,8}(134) = (1000\,0110)_2$$
$$c_{\mathrm{EX}-40,8}(0) = (0010\,1000)_2$$
$$c_{\mathrm{EX}-64,8}(-50) = (0000\,1110)_2$$
$$c_{\mathrm{EX}-3,4}(11) = (1110)_2$$

Decodierung:

$$(0001)_{\mathrm{EX}-7,4} = (0001)_{2,4} - 7 = 1 - 7 = -6$$
$$(1010)_{\mathrm{EX}-5,4} = 5$$
$$(0101\,0110)_{\mathrm{EX}-1,8} = 85$$
$$(0000\,0000\,0000\,0001)_{\mathrm{EX}-264,16} = -263$$

5.1.3 Addition / Subtraktion von EX-q-Darstellungen

Definition: 5.3 Addition

$$c_{EX-q,n}(x) \oplus_{EX-q,n} c_{EX-q,n}(y) := (c_{EX-q,n}(x) + c_{EX-q,n}(y)) - c_{2,n}(q)$$

- $\oplus_{EX-q,n}$: Addition auf EX-q-Darstellungen in n Bit
- $+$: Addition auf Binärzahlen
- $-$: Subtraktion auf Binärzahlen

Das bedeutet: Man addiert „ganz normal“ die Binärzahlen und zieht vom Ergebnis nochmal den Binärwert von q ab.
Würde man vom Ergebnis nicht den Wert q abziehen, würde es das Ergebnis verfälschen. Der Grund hierfür liegt an der Art, wie die Exzeß-q Zahlen codiert werden. Da man zur Umwandlung in die Exzeß-q Darstellung für beide Zahlen den Wert q dazu addiert, würde man bei der Addition auch zweimal den Wert q im Ergebnis haben – was natürlich zuviel wäre.

Beispiel:

$$\begin{aligned} & c_{EX-127,8}(7) \oplus_{EX-127,8} c_{EX-127,8}(-7) \\ &= 1000\,0110 + 0111\,1000 - 0111\,1111 \\ &= 1111\,1110 - 0111\,1111 \\ &= 0111\,1111 \end{aligned}$$

Und damit wir auch sofort sehen, dass das Ergebnis richtig ist, rechnen wir das nochmal kurz zurück:

$$\begin{aligned} & (0111\,1111)_{EX-127,8} \\ &= (127 - 127)_{10} \\ &= (0)_{10} \\ &= c_{EX-127,8}(7 - 7) \end{aligned}$$

Definition: 5.6 Subtraktion

$$c_{EX-q,n}(x) \ominus_{EX-q,n} c_{EX-q,n}(y) := (c_{EX-q,n}(x) - c_{EX-q,n}(y)) + c_{2,n}(q)$$

- $\ominus_{EX-q,n}$: Subtraktion auf EX-q-Darstellungen in n Bit
- $+$: Addition auf Binärzahlen
- $-$: Subtraktion auf Binärzahlen

Die Subtraktion funktioniert ähnlich zur Addition. Man subtrahiert beide Binärwerte und addiert hierzu nochmal den Wert q.

Beispiel:

$$\begin{aligned}
&c_{EX-127,8}(12) \ominus_{EX-127,8} c_{EX-127,8}(7)\\
&= 1000\,1011 - 1000\,0110 + 0111\,1111\\
&= 0000\,0101 + 0111\,1111\\
&= 1000\,0100
\end{aligned}$$

Und um auch hier sich vergewissern zu können, dass das Ergebnis stimmt:

$$\begin{aligned}
&(1000\,0100)_{EX-127,8}\\
&= (132 - 127)_{10}\\
&= (5)_{10}\\
&= c_{EX-127,8}(12 - 7)
\end{aligned}$$

Bei einem **Bereichsüberlauf** (Verlassen des mit den gegebenen Bits darstellbaren Bereichs) ist das Ergebnis von Addition und Subtraktion **undefiniert**!

5.2 Festkommacodierung reeller Zahlen

Bevor wir uns der Gleitkommacodierung reeller Zahlen zuwenden, schauen wir uns erst an, wie die Festkommacodierung reeller Zahlen funktioniert und wo ihre Vor- und Nachteile liegen.

5.2.1 Was ist eine Festkommacodierung in n Bit?

Grundidee: Um eine Binärzahl mit bis zu k Nachkommastellen im Speicher eines Rechners ablegen zu können, multipliziere diese einfach mit 2^k. Dadurch wird das Komma so weit nach vorne geschoben, dass aus der reellen Zahl eine ganze Zahl wird. Falls die multiplizierte Zahl dennoch Nachkommastellen hat, runde sie zur nächsten ganzen Zahl und verwerfe die verbliebenen Nachkommastellen.

Definition: 5.8 Festkommacodierung

Die **Festkommacodierung** $c_{FK,k,n} : [0, 2^{n-k}[\to \mathbb{B}^n$ mit $n - k$ **Vorkommastellen** und k **Nachkommastellen** ist definiert durch

$$c_{FK,k,n}(x) := c_{2,n}(rd(x \cdot 2^k)),$$

wobei

$$rd : [0, \infty[\to \mathbb{N}_0,\ rd(y) := \begin{cases} \lfloor y \rfloor & \text{, falls } y - \lfloor y \rfloor < 0.5 \text{ (abrunden)} \\ \lceil y \rceil & \text{, falls } \lceil y \rceil - y \leqslant 0.5 \text{ (aufrunden)} \end{cases}$$

die **Rundung zur nächstgelegenen ganzen Zahl** ist.

Wenn man eine Zahl x in die Festkommacodierung umwandeln will, verfährt man wie folgt:
Zuerst multipliziert man x mit dem Wert 2^k. Das k muss/ist immer fest vorgegeben. Anschließend rundet man das Ergebnis, wie man es schon aus der Schule kennt. Ist die Nachkommastelle $\geqslant 0.5$, dann wird aufgerundet. Ist sie < 0.5 dann rundet man ab. Der Grund für das Runden ist, dass man im Computer kein Komma darstellen kann. Da man schon durch die Multiplikation mit 2^k die Nachkommastellen (bis zu einer gewissen Genauigkeit) mitcodiert, werden die Nachkommastellen vom Ergebnis einfach gerundet.
Anschließend wird diese Zahl „ganz normal“ in die Binärdarstellung codiert und man füllt sie mit 0 auf, bis n Bits vorliegen.

Hierbei gilt:
Eine Zahl wird auf n Bits codiert, also besitzt sie auch n Stellen. Davon werden k Bits für die Nachkommastellen, und $n - k$ Bits für die Vorkommastellen verwendet. Wir bezeichnen die Codierung mit $c_{FK,k,n}$, z.B. $c_{FK,3,8}$ für 3 Nachkommastellen bei 8 Bit. Damit stehen $8 - 3 = 5$ Bit für Vorkommastellen zur Verfügung.

Beispiel:

Umwandlung der Zahl 4.65 in die Festkommacodierung, mit $k = 4$ und $n = 8$. Das bedeutet: $c_{FK,4,8}(4.65)$
Man rechnet nun zuerst:

$$4.65 \cdot 2^4 = 42.4$$

Das Ergebnis muss man nun runden:

$$rd(42.4) = 42$$

Und diese 42 wird nun binär codiert:

$$(42)_{10} = (101010)_2$$

Da eine Länge von 8 Bits verlangt ist (da $n = 8$), müssen noch zwei Nullen vorne ergänzt werden. Dann ist man fertig.

$$c_{FK,4,8}(4.65) = (0010\,1010)_2$$

5.2.2 Decodierung und Beispiel

Decodierung:
Man decodiert die Binärzahl und dividiert das Ergebnis durch 2^k.

Beispiel:

Decodierung der Zahl $(1110\,1010)_{FK,6,8}$.
Zuerst decodiert man die Binärzahl in das Dezimalsystem:

$$(1110\,1010)_2 = (234)_{10}$$

Anschließend dividiert man das Ergebnis durch 2^k:

$$\frac{234}{2^6} = (3.65625)_{10}$$

Und man ist fertig:

$$(1110\,1010)_{FK,6,8} = (3.65625)_{10}$$

Ein weiteres Beispiel:

Beispiel: Berechnung von $c_{FK,3,8}(1.2)$

1. Multiplizieren mit 2^3: $1.2 \cdot 2^3 = 9.6$
2. Runden: $\mathrm{rd}(9.6) = 10$
3. Binärcodierung mit 8 Bit: $c_{FK,3,8}(1.2) = c_{2,8}(10) = 00001010$
4. Decodierung: $(00001010)_{FK,3,8} = (1.010)_2 = 1.25$

Im Beispiel sieht man, dass die ursprüngliche Zahl und die Decodierung ihrer Codierung **nicht** übereinstimmen. Wenn man die beiden Zahlen voneinander abzieht, erhält man den *absoluten Rundungsfehler*: $|1.2 - 1.25| = 0.05$ (als Betrag, da ein Rundungsfehler immer positiv angegeben wird).

5.2.3 Arithmetik

Mit Festkommazahlen kann (und muss) man natürlich auch rechnen können. Bei der Arithmetik zu Festkommazahlen muss nichts Neues beachtet werden. Operationen wie Addition, Subtraktion, etc. funktionieren genauso wie bei den bereits bekannten Binärzahlen.

Es wird die Arithmetik auf Binärzahlen verwendet:
$c_{FK,k,n}(x) \oplus_{FK,k,n} c_{FK,k,n}(y) := c_{FK,k,n}(x) + c_{FK,k,n}(y)$
$c_{FK,k,n}(x) \ominus_{FK,k,n} c_{FK,k,n}(y) := c_{FK,k,n}(x) - c_{FK,k,n}(y)$

Beispiel:

Addition:

$$\begin{aligned}(0011\,1010)_{FK,4,8} + (0100\,0001)_{FK,4,8} \\ = (0111\,1011)_{FK,4,8}\end{aligned}$$

Subtraktion:

$$\begin{aligned}(1001\,1110)_{FK,4,8} - (0100\,0001)_{FK,4,8} \\ = (0101\,1101)_{FK,4,8}\end{aligned}$$

Hierbei können jedoch weitere Rundungsfehler hinzu kommen, da mehrfach gerundet wird: $c_{FK,k,n}(x) \oplus_{FK,k,n} c_{FK,k,n}(y) = c_{2,n}(rd(x \cdot 2^k) + rd(y \cdot 2^k)) \approx c_{2,n}(rd(x \cdot 2^k + y \cdot 2^k)) = c_{FK,k,n}(x+y)$

Beispiel:

- Beispiel für Addition mit auftauchendem Rundungsfehler:

$$\begin{aligned} & c_{FK,3,8}(1.5) \oplus_{FK,3,8} c_{FK,3,8}(4.2) \\ &= (0000\,1100)_{2,8} + (0010\,0001)_{2,8} \\ &= (0010\,1101)_{2,8} \\ &= c_{FK,3,8}(5.625) \end{aligned}$$

- Beispiel für Addition, bei der Rundungsfehler sich aufheben:

$$\begin{aligned} & c_{FK,3,8}(3.8) \oplus_{FK,3,8} c_{FK,3,8}(4.325) \\ &= (0001\,1110)_{2,8} + (0010\,0011)_{2,8} \\ &= (0100\,0001)_{2,8} \\ &= c_{FK,3,8}(8.125) \end{aligned}$$

5.2.4 Exakt darstellbare Zahlen

Definition: 5.10 Exakt darstellbare Zahlen

Eine reelle Zahl x heißt **exakt darstellbar bzgl. einer Codierung** c, falls

$$c^{-1}(c(x)) = x$$

(es tritt kein Rundungsfehler bei der Codierung auf)

Wie schon die Definition besagt: Eine exakt darstellbare Zahl ist eine Zahl, bei der für die Festkommacodierung keine Rundungsfehler auftreten. Dabei muss man beachten, dass es von k und n abhängt, ob eine Zahl exakt darstellbar ist oder nicht.

Die Daumenregel hier ist: Eine Zahl kann man, wie in Kapitel 3 definiert, in eine Binärzahl umwandeln (bspw.: $(4.625)_{10} = (100.101)_2$). Es kommt dann zum Rundungsfehler, wenn man für die Nachkommastellen der Zahl mehr als k Bits benötigt. Im kleinen Beispiel hier würde für ein $k = 3$ **kein** Rundungsfehler auftreten. Würde man jedoch $k = 2$ wählen, dann tritt ein Rundungsfehler auf, da die letzte 1 abgeschnitten wird.

Beispiel: Berechne $c_{FK,3,8}(1.75)$

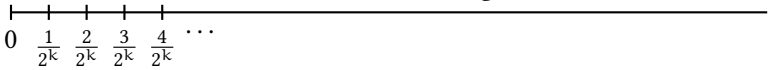
Ein Beispiel für solch eine exakt darstellbare Zahl wäre beispielsweise 1.75, mit einem $k = 3$ und $n = 8$.

1. Multiplizieren mit 2^3: $1.75 \cdot 2^3 = 14$
2. Runden: $\mathrm{rd}(14) = 14$
3. Binärcodierung mit 8 Bit: $c_{FK,3,8}(1.75) = c_{2,8}(14) = 00001110$
4. Decodierung: $(00001110)_{FK,3,8} = (1.110)_2 = 1.75$
5. Absoluter Rundungsfehler: $|1.75 - 1.75| = 0$

Satz 1. *Eine reelle Zahl* $x \in [0, 2^{n-k}[$ *ist genau dann exakt darstellbar bzgl.* $c_{FK,k,n}$*, falls sie in der Binärdarstellung* ***höchstens*** k ***Nachkommastellen*** *hat. Dies ist gleichbedeutend mit* $x \cdot 2^k \in \mathbb{N}_0$.

Beweis. Nach der Definition wird ab der $(k+1)$-ten Nachkommastelle gerundet. □

Exakt darstellbare Zahlen auf der Zahlengeraden:



5.2.5 Rundungsfehler bei der Festkommacodierung

Da natürlich nicht immer eine exakt darstellbare Zahl vorliegt, ist (für uns jedenfalls) interessant zu erfahren, wie groß der Rundungsfehler ist. Dabei unterscheidet man zwischen absolutem und relativen Rundungsfehler.

Definition: 5.13 Absoluter und relativer Rundungsfehler

Für eine reelle Zahl x und eine Codierung c heißt:

- $|x - c^{-1}(c(x))|$ **absoluter Rundungsfehler**.
- $\frac{|x - c^{-1}(c(x))|}{|x|}$ **relativer Rundungsfehler**.

Der *absolute Rundungsfehler* gibt die Abweichung an zwischen ursprünglicher Zahl und Decodierung der codierten Zahl. Beim *relativen Rundungsfehler* wird dieser Wert noch durch die ursprüngliche Zahl geteilt, um eine prozentuale Abweichung zu erhalten.

Beispiel: $c_{FK,3,8}(1.2)$

1. $c_{FK,3,8}(1.2) = 00001010$
2. Decodierung: $(00001010)_{FK,3,8} = (1.010)_2 = 1.25$
3. Absoluter Rundungsfehler: $|1.2 - 1.25| = 0.05$
   Das bedeutet, dass der absolute Rundungsfehler zwischen codierter und ursprünglicher Zahl sich auf 0.05 beläuft.
4. Relativer Rundungsfehler: $\frac{|1.2-1.25|}{1.2} = 0.041\overline{6}$
   Das bedeutet, dass ein relativer Rundungsfehler von 4.17% vorliegt.

Abstände zwischen exakt darstellbaren Zahlen sind gleichbleibend:

Satz 2 (Maximaler absoluter Rundungsfehler der Festkommacodierung)**.** *Es gilt:*

$$|x - (c_{FK,k,n}(x))_{FK,k,n}| \leqslant \frac{1}{2^{k+1}}.$$

Dieser Satz besagt, dass der maximale absolute Rundungsfehler nach oben beschränkt werden kann. Somit kann der absolute Rundungsfehler nicht unendlich groß werden (was natürlich sehr praktisch ist).
Allerdings kann der relative Rundungsfehler für kleine x beliebig groß werden. Dies wird in den beiden folgenden Beispielen verdeutlicht.

Satz 3 (Maximaler relativer Rundungsfehler der Festkommacodierung)**.** *Es gilt:*

$$\frac{|x - (c_{FK,k,n}(x))_{FK,k,n}|}{|x|} \leqslant \frac{1}{(|x| \cdot 2^{k+1})}.$$

Beispiel: $c_{FK,3,8}(10.1)$

1. $c_{FK,3,8}(10.1) = 001010001$
2. Decodierung: $(001010001)_{FK,3,8} = (1010.001)_2 = 10.125$
3. Absoluter Rundungsfehler: $|10.1 - 10.125| = 0.025$
4. Relativer Rundungsfehler: $\frac{|10.1-10.125|}{10.1} = 0.00247..$

Ein relativer Rundungsfehler von 0.247...% mag bei einem Wert von 10.1 nicht so stark ins Gewicht fallen. Umso schlimmer kann sich ein Rundungsfehler der selben Größe aber bei kleinen Zahlen auswirken:

Beispiel: $c_{FK,3,8}(0.1)$

1. $c_{FK,3,8}(0.1) = 00000001$
2. Decodierung: $(00000001)_{FK,3,8} = (0.001)_2 = 0.125$
3. Absoluter Rundungsfehler: $|0.1 - 0.125| = 0.025$
4. Relativer Rundungsfehler: $\frac{|0.1-0.125|}{0.1} = 0.25$

Ein relativer Rundungsfehler von 25% ist in den seltensten Fällen tolerierbar.

5.2.6 Bewertung

Verwendung der Festkommacodierung von Zahlen:

- Gleichbleibender Abstand $\frac{1}{2^k}$ zwischen exakt darstellbaren Zahlen:
  Geeignet für Zahlen ähnlicher Größenordnung (Anzahl der Nachkommastellen k an Größenordnung anpassen, so dass Rundungsfehler tolerierbar)
- Problematisch für Rechnungen mit Zahlen unterschiedlicher Größenordnung (Wertebereich muss an größere Zahl angepasst werden; führt zu ggf. zu wenig Nachkommastellen für die kleinere Zahl)
- Zwar ist der *absolute* Rundungsfehler begrenzt, aber bei kleinen Zahlen kann der *relative* Rundungsfehler sehr groß werden, was die Festkommacodierung in diesen Fällen unpraktikabel macht.

5.3 Gleitkommacodierung reeller Zahlen

Nachdem wir die Probleme gesehen haben, die die Festkommacodierung mit sich bringt, lernen wir nun die Gleitkommacodierung kennen, die sich in heutigen Rechnern durchgesetzt hat. Sie kann problemlos sehr große, aber auch sehr kleine Zahlen darstellen, ohne dass dabei ein zu hoher Rundungsfehler auftritt. Somit versetzt uns die Gleitkommacodierung in die Lage, effizient reelle Zahlen im Computer darstellen und nutzen zu können. Hierfür benötigt man die gerade kennengelernten Exzeß-q- und Festkommacodierungen.

5.3.1 Was ist die Gleitkommacodierung?

Die Gleitkommacodierung setzt die Gleitkomma-Darstellung aus Kapitel 3 um, die am Anfang des Kapitels nochmal wiederholt wurde. Dabei ist eine Gleitkommazahl mit n Bits und $k-1$ Nachkommastellen wie folgt im Rechner hinterlegt:

1. 1 Bit Vorzeichen (gibt an, ob Mantisse positiv oder negativ ist).
2. $n-k$ Bit Exponent (auch *Charakteristik* genannt),
   in $\mathrm{EX}_{-q,n-k}$-Codierung, wobei $q = 2^{n-k-1}-1$.
3. $k-1$ Bit Mantisse. Da die Mantisse aufgrund von $1 \leqslant |m| < B$ im Binärsystem immer mit $1.\ldots$ beginnt, lässt man die 1 vor dem Komma weg und codiert nur die Nachkommastellen. Hierfür verwendet man die Festkommacodierung mit $c_{\mathrm{FK},k-1,k-1}$ (wird nach der Definition genauer erklärt).

Davon ausgehend definieren wir die Gleitkommacodierung nun formal:

Definition: 5.19 Gleitkommacodierung

Die **Gleitkommacodierung** $c_{GK,k,n} :]-2^{2^{n-k-1}}, 2^{2^{n-k-1}}[\to \mathbb{B}^n$ einer reellen Zahl x mit **normierter Gleitkomma-Darstellung** $x = m \cdot 2^e$ ist definiert durch:

$$c_{GK,k,n}(x) := x_{n-1}\underbrace{x_{n-2} \ \dots \ x_{k-1}}_{\text{Charakteristik}}\underbrace{x_{k-2} \ \dots \ x_0}_{\text{Mantisse}},$$

wobei

- $x_{n-1} := \begin{cases} 0 & \text{falls } m \geqslant 0 \\ 1 & \text{falls } m < 0 \end{cases}$ (**Vorzeichenbit**).
- $x_{n-2} \dots x_{k-1} := c_{EX-(2^{n-k-1}-1),n-k}(e)$
  EX-q-Darstellung des **Exponenten** (auch Charakteristik genannt) in $n-k$ Bit mit $q = 2^{n-k-1} - 1$.
- $x_{k-2} \dots x_0 := c_{FK,k-1,k-1}(|m| - 1)$
  Festkommacodierung der **Mantisse** mit $k-1$ Nachkommastellen und 0 Vorkommastellen (wegen der Normierung ist die Vorkommastelle immer 1)

Um eine Zahl x in die Gleitkommacodierung umzuwandeln, geht man wie folgt vor. Hierbei ist ein festes k (gibt die Anzahl der Bits für die Nachkommastellen an) und ein festes n (Gesamtzahl der Bits für die Codierung) gegeben:

1. Zuerst muss die Zahl x normiert werden (siehe Kapitel 3). Dadurch erhält man die Form $x = m \cdot 2^e$, wobei $1 \leqslant |m| < B$ (hier wird der Betrag der Mantisse eingesetzt, da das Vorzeichen extra codiert wird, siehe nächster Schritt).

2. Danach muss das Vorzeichen der Zahl bestimmt werden. Hierfür schaut man einfach, ob es eine negative oder positive Zahl ist. Für eine negative Zahl ist das erste Bit der Codierung eine 1. Für eine postive Zahl ist es eine 0.

3. Anschließend codiert man den Exponenten e mithilfe der Exzeß-q-Codierung:

   - Für die Exzeß-q-Codierung muss ein q **und** die Anzahl der Bits für die Codierung bestimmt werden.
   - q bestimmt man mit folgender Formel: $q = 2^{n-k-1} - 1$
   - Die Anzahl der Bits für q wird anhand von $n-k$ berechnet.
   - Codiere nun: $c_{EX-q,n-k}(e)$

4. Zu guter Letzt codiert man nun die Mantisse. Da diese immer mit 1. ... beginnt, codieren wir nur die Nachkommastellen von m. Beispielsweise für die Zahl 1.456 codieren wir 0.456. Hierzu nutzen wir die Festkommacodierung:

   - Für eine Festkommacodierung $c_{FK,k,n}$[2] benötigt man ein k, welches die Anzahl der Nachkommastellen angibt. Dieses entspricht den $k-1$ Bits, die in der Gleitkommacodierung für die Mantisse vorgesehen sind. Das n in der Festkommacodierung gibt

[2] **Achtung:** k und n beziehen sich hier auf die Definition der **Fest**kommacodierung!

die gesamte Anzahl der verwendeten Bits an. Da wir keine Vorkommastellen codieren, stimmt n überein mit der Anzahl der Nachkommastellen. Also berechnen wir $c_{FK,k-1,k-1}$[3].
- Zu codieren ist also $c_{FK,k-1,k-1}(m-1)$.

5. Anschließend muss man nur noch die codierte Zahl in der folgenden Reihenfolge zusammensetzen: Vorzeichenbit Exponent Mantisse

Schauen wir uns das anhand eines konkreten Beispiels an:

Beispiel:

Darstellung der Zahl -5.8 in der Gleitkommacodierung,
mit $k = 4$ und $n = 8$, d.h. 8 Bits gesamt, davon:

- 1 Bit für das Vorzeichen
- $n - k = 4$ Bits für den Exponenten
- $k - 1 = 3$ Bit für die Mantisse

1. Also wir wollen folgendes: $c_{GK,4,8}(-5.8)$
2. Normiere zuerst die Zahl: $5.8 = 1.45 \cdot 2^2$
   Also gilt: $m = 1.45$ und $e = 2$
3. Die Zahl ist negativ. Dadurch ist das Vorzeichenbit = 1
4. Berechne nun den Exponenten 2 mithilfe der Exzeß-q-Codierung:
   - Bestimme q: $$\begin{aligned} q &= 2^{n-k-1} - 1 \\ &= 2^{8-4-1} - 1 \\ &= 7 \end{aligned}$$
   - Anzahl der Bits für q: $n - k = 8 - 4 = 4$
   - Berechne nun $c_{EX-7,4}(2) = c_{2,4}(2+7) = 1001$
   - Man hat nun den Exponenten = 1001
5. Nun fehlt noch die Mantisse, welche mit Hilfe der Festkommacodierung berechnet wird:
   - Das k und die Anzahl der Bits werden durch $k - 1$ berechnet.
     Somit: $$\begin{aligned} & c_{FK,4-1,4-1}(1.45 - 1) \\ &= c_{FK,3,3}(0.45) \\ &= c_{2,3}(rd(0.45 \cdot 2^3)) \\ &= c_{2,3}(4) \\ &= 100 \end{aligned}$$
   - Somit kennt man auch die Mantisse = 100
6. Setze nun die codierte Zahl zusammen:
   $c_{GK,4,8}(-5.8) = 11001100$

[3] Hier ist k wieder das k aus der **Gleit**kommacodierung.

5.3.2 Decodierung und Beispiel

Für die Decodierung muss man nun überlegen, welche Bits wohin gehören.
Dafür kann man sich an der Codierung orientieren und mit den selben Formeln und Regeln bestimmen, welche Bits nun zu Vorzeichen, Mantisse oder Exponent gehören:

- Das erste Bit von links ist das Vorzeichen.
- Die $n - k - 1$ Bits nach dem Vorzeichenbit gehören der Charakteristik an.
- Die $k - 1$ Bits von rechts gehören der Mantisse an.

Diese Bits der Charakteristik und Mantisse müssen nun gesondert behandelt werden.

Um eine codierte Gleitkommazahl zu decodieren, führt man folgende Schritte aus:

1. Bestimme, welche Bits der Mantisse, Charakteristik und dem Vorzeichen angehören.
2. Berechne alle Parameter für die Exzeß-q Berechnung. d.h. bestimme q und das zur Exzeß-q gehörige n (Anzahl der Bits der Codierung).
3. Decodiere den Exponenten e mithilfe der Exzeß-q Berechnung.
4. Decodiere die Mantisse m mit der Festkommacodierung, addiere 1.
5. Bestimme das Vorzeichen.
6. Füge nun wieder die Zahl zusammen: Decodierte Zahl $x = m \cdot 2^e$

Oder formeller aufgeschrieben lautet die Decodierung folgendermaßen.
Es gilt mit $q = 2^{n-k-1} - 1$:

$$(x_{n-1} \dots x_1 x_0)_{GK,k,n} = \begin{cases} (1.x_{k-2} \dots x_0)_2 \cdot 2^{(x_{n-2} \dots x_{k-1})_{EX-q,n-k}} & \text{falls } x_{n-1} = 0 \\ -(1.x_{k-2} \dots x_0)_2 \cdot 2^{(x_{n-2} \dots x_{k-1})_{EX-q,n-k}} & \text{falls } x_{n-1} = 1 \end{cases}$$

Beispiel: Codierung und Decodierung von $c_{GK,5,8}(4.6)$

Codierung:

1. Normierte Gleitkommadarstellung: $4.6 = 2.3 \cdot 2^1 = 1.15 \cdot 2^2$.
2. Vorzeichenbit: 0 (positive Mantisse)
3. q bestimmen: $q = 2^{(8-5)-1} - 1 = 3$
4. Charakteristik: $c_{EX-3,3}(2) = c_{2,3}(2 + 3) = 101$
5. Codierung der Mantisse: $c_{FK,4,4}(0.15) = c_{2,4}(\mathrm{rd}(0.15 \cdot 2^4)) = 0010$
6. Zusammensetzen: $c_{GK,5,8}(4.6) = 01010010$

Decodierung:

1. Zuordnung der Bits: Vorzeichen: 0, Charakteristik: 101, Mantisse: 0010
2. Parameter für die Exzeß-q Berechnung:
   $q = 2^{(8-5)-1} - 1 = 3$, Anzahl der Bits für q: $n - k = 8 - 5 = 3$
3. Decodiere den Exponenten e: $(101)_{EX-3,3} = (101)_2 - 3 = 2$

4. Decodiere die Mantisse m: $(0010)_{FK,4,4} = 2/2^4 = 0.125$
   Addiere 1: Mantisse = 1.125
5. Bestimme das Vorzeichen: 0 $\Rightarrow$ positiv
6. Zusammenfügen: $x = m \cdot 2^e = +1.125 \cdot 2^2 = 4.5$

Absoluter Rundungsfehler: $|4.6 - 4.5| = 0.1$
Relativer Rundungsfehler: $\frac{|4.6-4.5|}{4.6} = 0.021..$

5.3.3 Weitere Beispiele

Beispiel: Kleinste positive darstellbare normalisierte Zahl bzgl. $c_{GK,5,8}$

- Kleinster Exponent (Bitmuster 0...0 schon vergeben):
  $(001)_{EX-3,3} = (001)_{2,3} - 3 = 1 - 3 = -2.$
- Nachkommastellen der kleinsten Mantisse:
  $(0000)_{FK,4,4} = (0.0000)_2$
- *Ergebnis*:
  $(00010000)_{GK,5,8} = (1.0000)_2 \cdot 2^{-2} = (0.01)_2 = 0.25$

Beispiel: Größte positive darstellbare normalisierte Zahl bzgl. $c_{GK,5,8}$

- Größter Exponent (Bitmuster 1...1 schon vergeben):
  $(110)_{EX-3,3} = (110)_{2,3} - 3 = 6 - 3 = 3.$
- Nachkommastellen der größten Mantisse:
  $(1111)_{FK,4,4} = (0.1111)_2$
- *Ergebnis*:
  $(01101111)_{GK,5,8} = (1.1111)_2 \cdot 2^3 = (1111.1)_2 = 15.5$

5.3.4 Reservierte Bitmuster

Für bestimmte Werte und Rechenergebnisse gibt es reservierte Bitmuster, wie beispielsweise:

- Spezielle Darstellung von 0:

$$0\ \underbrace{0\ \ldots\ 0}_{\text{Charakteristik}}\ \underbrace{0\ \ldots\ 0}_{\text{Mantisse}}$$

  Die 0 spielt hier eine kleine Sonderrolle. Wie zu sehen ist, besteht die 0 aus einer Folge von n vielen Nullen. Das Problem ist allerdings, dass durch Arithmetik und Rundungsfehler das Ergebnis 0 oft nicht angenommen werden kann: Nach mehreren Rechenoperationen und damit verbundenen Rundungen kann es z.B. passieren, dass man von einem Zwischenergebnis vermeintlich die gleiche Zahl abzieht, aber das Ergebnis nicht 0 ist, sondern bspw. 0.000000012. Aus diesem Grund sollte man Werte nicht auf 0 testen, sondern auf Werte sehr

nahe der Null (Details folgen auf Seite 18).

- Spezielle Darstellung von ∞ (z.B. als Ergebnis von $1/0$):

$$0\ \underbrace{1\ \ \ldots\ \ 1}_{\text{Charakteristik}}\ \underbrace{0\ \ \ldots\ \ 0}_{\text{Mantisse}}$$

- Spezielle Darstellung von $-\infty$ (z.B. als Ergebnis von $-1/0$):

$$1\ \underbrace{1\ \ \ldots\ \ 1}_{\text{Charakteristik}}\ \underbrace{0\ \ \ldots\ \ 0}_{\text{Mantisse}}$$

- Darstellung von NaN (**Not a number**, z.B. als Ergebnis der Rechnung $0/0$ oder $\infty - \infty$):

$$x_{n-1}\ \underbrace{1\ \ \ldots\ \ 1}_{\text{Charakteristik}}\ \underbrace{x_{k-2}\ \ \ldots\ \ x_0}_{\text{Mantisse}}$$

  ($x_{n-1}x_{k-2}\ldots x_0$ ist ein plattformabhängiger Fehlercode)
- Darstellung von **denormalisierten Zahlen** x der Form $|x| = 0.x_{k-2}\ldots x_0 \cdot 2^{min}$, wobei $min = -(2^{n-k-1} - 2)$ der kleinstmögliche Exponent ist:

$$x_{n-1}\ \underbrace{0\ \ \ldots\ \ 0}_{\text{Charakteristik}}\ \underbrace{x_{k-2}\ \ \ldots\ \ x_0}_{\text{Mantisse}}$$

Zusammengefasst stehen die Bitmuster $0\ldots0$ und $1\ldots1$ **nicht** für die Darstellung des **Exponenten** zur Verfügung.

5.3.5 Arithmetik

Will man nun Addition und Subtraktion auf Gleitkommazahlen ausführen, ist das für uns Menschen sehr aufwendig, da wir nicht direkt auf den codierten Bitmustern rechnen können wie ein Rechner. Liegt eine codierte Gleitkommazahl vor, muss diese erst in die $x = m \cdot 2^e$ Darstellung decodiert werden.

Wie wir schon in Kapitel 3 gelernt haben, erfolgen Addition und Subtraktion zweier normierter Gleitkommazahlen $m_1 \cdot 2^{e_1}$ und $m_2 \cdot 2^{e_2}$ auf Mantisse und Exponent:

1. Exponentenangleich der kleineren Zahl ggf. mit Rundung (der Exponentenangleich entspricht einer Verschiebung der Stellen nach rechts – das kann zu einer nicht exakt darstellbaren Zahl führen).
2. Addition / Subtraktion der Mantissen und ggf. Normierung mit Rundung des Ergebnisses (bei der Normierung werden wieder Stellen verschoben – das kann hier noch einmal zu einer nicht exakt darstellbaren Zahl führen)

> Beispiel:
>
> Addition der $1.52 \cdot 2^5$ mit der Zahl $1.63 \cdot 2^3$.

Dafür müssen wir zuerst beide Exponenten auf den selben Wert bringen:

$$1.52 \cdot 2^5 \text{ und } 0.0163 \cdot 2^5$$

Nun können wir die Mantissen addieren:

$$1.52 + 0.0163 = 1.5363$$

Und wir bekommen folgendes Ergebnis: $1.5363 \cdot 2^5$

Das Ergebnis können wir nun wieder in eine Gleitkommazahl umrechnen.

Problem:
Addition sehr unterschiedlich großer Zahlen und Subtraktion fast gleich großer Zahlen kann zu Stellenauslöschungen durch Rundung führen.

Rundung und Stellenauslöschung durch Exponentenangleich:
Betrachte $m_1 \cdot 2^{e_1} + m_2 \cdot 2^{e_2}$ mit $e_2 < e_1$ in einer $c_{GK,k,n}$-Codierung. Durch den Exponentenangleich werden die Stellen der kleineren Zahl $m_2 \cdot 2^{e_2}$ um $e_2 - e_1$ Stellen rechts verschoben:

$$m_1 \cdot 2^{e_1} + m_2 \cdot 2^{e_2} = (m_1 + m_2 \cdot 2^{e_2 - e_1}) \cdot 2^{e_1}$$

Bzgl. der k-ten Nachkommastelle von $m_2 \cdot 2^{e_2 - e_1}$ wird gerundet, die Nachkommastellen ab der $k+1$-ten werden gelöscht (da diese nicht mehr darstellbar sind). Im Falle $e_2 - e_1 < -k$ gilt also sogar:

$$(m_1 + m_2 \cdot 2^{e_2 - e_1}) \cdot 2^{e_1} = m_1 \cdot 2^{e_1}$$

Die sehr kleine Zahl wird durch die Addition mit der großen „verschluckt“. Ein ähnlicher Effekt tritt bei Subtraktion fast gleich großer Zahlen auf.

In der Gleitkommarechnung hängt das Rechenergebnis wegen unterschiedlicher Rundungsfehler von der Auswertungsreihenfolge ab!

Beispiel: Rechenergebnis hängt von der Auswertungsreihenfolge ab

Betrachte folgende Rechnungen in einfacher Genauigkeit, d.h. in der $c_{GK,24,32}$-Codierung.

- $(1.0 \cdot 2^{-9} + 1.0 \cdot 2^{23}) - 2^{23} = (2^{-32} + 1.0) \cdot 2^{23} - 2^{23} = 0$
- $2^{-9} + (1.0 \cdot 2^{23} - 1.0 \cdot 2^{23}) = 2^{-9} + (1.0 - 1.0) \cdot 2^{23} = 2^{-9}$

Nachkommastellen ab der 24-ten Stelle werden abgeschnitten bzw. nicht berücksichtigt!

- $2^{23} \cdot ((2^{-9} + 2^{23}) - 2^{23}) = 0$
- $2^{23} \cdot (2^{-9} + (2^{23} - 2^{23})) = 2^{14}$

Sonderrolle der 0

- Hat keine Gleitkomma-Darstellung, da Mantisse immer mit 1. ... beginnt.

- Standardisierte Gleitkommacodierung durch reservierte Bitmuster (siehe Seite 16)
  $\Rightarrow$ Arithmetische Sonderbehandlung
- Exaktes Ergebnis 0 wird in der Regel wegen Rundungsfehlern nicht angenommen.

Daher geht man in Abfragen/Bedingungen wie folgt vor:

- `double`-**Variablen nicht auf** `0` **testen**
  Verwende `-r < x && x < r` statt `x == 0` für eine kleine positive Fehlerschranke `r`
- `double`-**Variablen nicht auf Gleichheit testen**
  Verwende `-r < x-y && x-y < r` statt `x == y` für eine kleine positive Fehlerschranke `r`

Fazit:

- Ergebnisse von Gleitkommaberechnungen können u.U. erheblich von dem exakten Wert abweichen
- Übliche Rechengesetze gelten im Allgemeinen nicht (Assoziativ-/Distributiv-/Kommutativgesetz)

Auswege:

- Exakte Arithmetik (z.B. mit spezieller Software, die exakt mit Brüchen rechnen kann)
- Gruppierung von Zahlen nach Größenbereichen

Für die meisten Anwendungen reicht es aber aus, Gleitkomma-Zahlen mit ausreichend vielen Bits zu verwenden, siehe Kapitel 5.3.8.

5.3.6 Exakt darstellbare Zahlen

Eine Zahl ist exakt darstellbar, wenn wir beim Speichern der Mantisse ohne Runden auskommen und der Exponent ebenso vollständig gespeichert werden kann.

Satz 4. *Eine reelle Zahl* $x \in]-2^{2^{n-k-1}}, 2^{2^{n-k-1}}[$ *in der normierten Gleitkomma-Darstellung* $x = m \cdot 2^e$ *ist genau dann exakt darstellbar bzgl.* $c_{GK,k,n}$*, falls*

- *Die Mantisse* m *in der Binärdarstellung* ***höchstens*** $k-1$ ***Nachkommastellen*** *hat*
- *und für den Exponenten* $-(2^{n-k-1}-2) \leqslant e \leqslant 2^{n-k-1}-1$ *gilt.*

Beispiel:

- Mantisse: Nach der Definition wird ab der k-ten Nachkommastelle gerundet.
- Exponent: Wegen der reservierten Bitmuster hat der
  - größte Exponent das Bitmuster $1\ldots10$:
    das entspricht $e = 2^{n-k-1}-1$
  - kleinste Exponent das Bitmuster $0\ldots01$:
    das entspricht $e = -(2^{n-k-1}-2)$
    (siehe Definitionsbereich der EX-q-Darstellung)

Die Idee hinter einer exakt darstellbaren Zahl ist also die selbe, wie sie auch in der Festkommacodierung beschrieben wurde. Das bedeutet also, dass keine Rundungsfehler auftreten.

5.3.7 Rundungsfehler

Abstände zwischen exakt darstellbaren Zahlen werden für große Exponenten sehr groß:

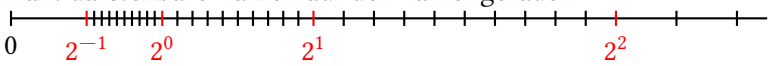
Satz 5 (Maximaler absoluter Rundungsfehler der Gleitkommacodierung)**.** *Es gilt:*

$$|m \cdot 2^e - (c_{GK,k,n}(m \cdot 2^e))_{GK,k,n}| \leqslant \frac{2^e}{2^k}.$$

Beispiel:

$|m \cdot 2^e - (c_{GK,k,n}(m \cdot 2^e))_{GK,k,n}|$
$= |m - (c_{FK,k-1,n}(m))_{FK,k-1,n}| \cdot 2^e$
$\leqslant {}^{2^e}/_{2^k}$

Exakt darstellbare Zahlen auf der Zahlengeraden:



5.3.8 IEEE-Standard 754

Das *Institute of Electrical and Electronics Engineers* (IEEE) als weltweiter Berufsverband der Ingenieure stellt Standards für technische Systeme auf. So ist im IEEE-754 Standard definiert, wie genau die Gleitkommadarstellung im Computer berechnet wird. Dazu ist auch die Größe und Genauigkeit unterschiedlicher Zahlenwerte angegeben. Hier ein paar Details zu den Genauigkeiten, Anzahl Bits etc. und was deren Äquivalente in der Programmiersprache C sind.

Einfache Genauigkeit (32 Bit, $k = 24$, Datentyp `float` in C)

- *8-Bit Charakteristik ($q = 127$)*:
  Exponent zwischen -126 (Bitmuster 0...01) und 127 (Bitmuster 1...10)
- *23-Bit Mantisse*:
  Werte zwischen 1 (Bitmuster 0...0) und $2 - 2^{-23}$ (Bitmuster 1...1)
- Maximaler absoluter Fehler: ${}^{2^{127}}/_{2^{24}}$
- Maximaler relativer Fehler: ${}^{1}/_{2^{24}}$

Doppelte Genauigkeit (64 Bit, Datentyp `double` in C)

- *11-Bit Charakteristik ($q = 1023$)*:
  Exponent zwischen -1022 (Bitmuster 0...01) und 1023 (Bitmuster 1...10)
- *52-Bit Mantisse*:
  Werte zwischen 1 (Bitmuster 0...0) und $2 - 2^{-52}$ (Bitmuster 1...1)

Erweiterte Genauigkeit (80 Bit, Datentyp `long double` in C)

- *15-Bit Charakteristik (*$q = 16383$*)*:
  Exponent zwischen $-(2^{-14} - 2)$ (Bitmuster 0...01) und $(2^{14} - 1)$ (Bitmuster 1...10)
- *64 -Bit Mantisse*:
  Werte zwischen 1 (Bitmuster 0...0) und $2 - 2^{-64}$ (Bitmuster 1...1)

5.3.9 Allgemeines zu Speicherbedarf und Wertebereich

- Der Speicherbedarf und die Grenzen des Wertebereichs der oben genannten Datentypen können über Bibliotheks-Konstanten in `float.h` abgefragt werden.
- Wenn in einer Berechnung oder durch eine Benutzereingabe der Wertebereich verlassen wird, kommt es zu Programm- oder Rechenfehlern.
- Der Speicherbedarf einer Variable `x` kann mit `sizeof(x)` abgefragt werden.
- Der Speicherbedarf eines Datentyps `T` kann mit `sizeof(T)` abgefragt werden.
- Der Compiler ordnet jeder Variablen in einem Programm einen festen Speicherbereich zu, und zwar durch Festlegung der **Adresse der ersten Speicherzelle** dieses Bereichs.

5.4 Literaturverzeichnis

siehe Foliensatz