

## Codierung von Dezimalzahlen

---

- Mit dem Zweierkomplement haben wir eine Möglichkeit zur Codierung von positiven und negativen ganzen Zahlen
- Wie lassen sich Dezimalzahlen (“Kommazahlen”) speichern?

- Idee:
  - Lege fest, wie viele Bits für Dezimalzahlen verwendet werden sollen (bspw. 32)
  - Lege fest, wie viele Bits davon Nachkommastellen sein sollen
- Wir haben eine **Festkommazahl** (fixed point number) konstruiert :-)

## Einfaches Beispiel

- Wir verwenden nur 16 Bit, davon 10 Bit für die Nachkommastellen
- Wir können wir 35,5625 codieren?
  - $35 = (100011)_2$
  - $0,5625 = (0,1001)_2$  (Wieso? Weil  $0,5625 = \frac{1}{2} + \frac{1}{16}$ )
  - Im Speicher steht also **1000111001000000**

- Das Verfahren ist verhältnismäßig einfach, aber auch unflexibel
  - Für alle Zahlen wird die gleiche Anzahl an Nachkommastellen verwendet, egal ob notwendig oder nicht

# Gleitkommazahlen

- Bei einer **Gleitkommazahl** (floating point number) kann sich das Komma an “beliebiger” Stelle befinden
- Die Darstellung lehnt sich an die wissenschaftliche Schreibweise von Zahlen an:  $x = m \cdot b^e$ , mit
  - $m$ : Mantisse
  - $e$ : Exponent
  - $b$ : Basis
- Die Anzahl an Bits für Mantisse und Exponent ist fest, aber je nach gespeicherten Werten sind sehr große oder sehr kleine Zahlen möglich
  - Beispiel:
    - $1,2345 \cdot 10^{18}$ : Sehr groß, keine Nachkommastellen
    - $1,2345 \cdot 10^{-5}$ : Sehr klein, viele Nachkommastellen

## Gleitkommazahlen - “Optimierungen”

- Die Basis kann einmalig festgelegt werden, muss also nicht explizit gespeichert werden
- Die Mantisse kann auf einen bestimmten Zahlenbereich normalisiert werden
  - Beispiel: Normalisieren auf  $0 \leq m < 1$ , sodass die 0 vor dem Komma nicht gespeichert werden muss.
    - Aus  $31 \cdot 10^2$  wird dann  $0,31 \cdot 10^4$

- Gebräuchlicher Standard für den Aufbau von Gleitkommazahlen
- Definiert Zahlen mit unterschiedlicher Genauigkeit, z. B.
  - single (32 Bit)
  - double (64 Bit)
- Legt Codierungen für “besondere” Werte fest:
  - Not a Number (NaN)
  - $+\infty$  und  $-\infty$
  - $+0$  und  $-0$
- Weitere Festlegungen:
  - Verschiedene Rechenoperationen
  - Rundungsregeln



- Allgemeines:
  - Basis  $b = 2$
  - Mantisse  $m$  normalisiert auf  $1 \leq m < 2$  (mit einigen Ausnahmen)
- 32 Bit pro Zahl, davon:
  - 1 Bit für das Vorzeichen
  - 8 Bit für den Exponenten
  - 23 Bit für die Mantisse
- Biaswert  $B = 127$ 
  - Anstelle des Exponenten  $e$  wird  $E = e + B$  gespeichert
  - Dadurch wird ein Vorzeichenbit für den Exponenten gespart

## Beispiel 1 - Darstellung von 3,75 (1)

- Umwandlung ins Zweiersystem:
  - Vorkommastelle:  $3 = (11)_2$
  - Nachkommastellen:  $0,75 = \frac{1}{2} + \frac{1}{4} = (0,11)_2$
  - Insgesamt:  $0,75 = (11,11)_2 = (11,11 \cdot 2^0)_2$
- Normalisierung der Mantisse  $m$ 
  - $(11,11 \cdot 2^0)_2 = (1,111 \cdot 2^1)_2$
  - Also Mantisse  $m = (1,111)_2$
- Bestimmung des Exponenten  $E$ 
  - $E = e + B = 1 + 127 = 128$
  - Also Exponent  $E = (10000000)_2$
- Bestimmung des Vorzeichens
  - 3,75 ist positiv, also Vorzeichenbit 0

## Beispiel 1 - Darstellung von 3,75 (2)

- Zusammenfassung:
  - Mantisse  $m = (1,111)_2$
  - Exponent  $E = (10000000)_2$
  - Vorzeichenbit 0
- Darstellung im Speicher

0 10000000 111000000000000000000000

- Von  $m$  werden nur die Nachkommastellen gespeichert, denn vor dem Komma steht nach der Normalisierung ohnehin immer eine 1
- Rundungsregeln wurden bei der Umrechnung nicht beachtet
- Die Leerzeichen dienen natürlich lediglich der besseren Lesbarkeit

## Beispiel 2 - Darstellung von $5,4$ (1)

- Umwandlung ins Zweiersystem:
  - Vorkommastelle:  $5 = (101)_2$
  - Nachkommastellen:  $0,4 = (0,\overline{0110})_2$ , denn
    - $0,4 \cdot 2 = 0,8 < 1 \Rightarrow 0$
    - $0,8 \cdot 2 = 1,6 \geq 1 \Rightarrow 1$
    - $0,6 \cdot 2 = 1,2 \geq 1 \Rightarrow 1$
    - $0,2 \cdot 2 = 0,4 < 1 \Rightarrow 0$
    - $0,4 \cdot 2 = 0,8 \dots$
  - Insgesamt:  $5,4 = (101,\overline{0110})_2 = (101,\overline{0110} \cdot 2^0)_2$
- Normalisierung der Mantisse  $m$ 
  - $(101,\overline{0110} \cdot 2^0)_2 = (1,01\overline{0110} \cdot 2^2)_2$
  - Also Mantisse  $m = (1,01\overline{0110})_2$

## Beispiel 2 - Darstellung von $5,4$ (2)

- Bestimmung des Exponenten  $E$ 
  - $E = e + B = 2 + 127 = 129$
  - Also Exponent  $E = (10000001)_2$
- Bestimmung des Vorzeichens
  - $5,4$  ist positiv, also Vorzeichenbit  $0$

## Beispiel 2 - Darstellung von $5,4$ (3)

- Zusammenfassung:
  - Mantisse  $m = (1,01\overline{0110})_2$
  - Exponent  $E = (10000001)_2$
  - Vorzeichenbit 0
- Darstellung im Speicher

0 10000001 01011001100110011001100

- Rundungsregeln wurden bei der Umrechnung nicht beachtet

## Aufgabe 1

- Gib für die folgenden Zahlen die Codierung gemäß IEEE-754 Single an:
  - a) 0,1
  - b)  $-14,28$
- Lösung:
  - a) 0 01111011 10011001100110011001101
  - b) 1 10000010 11001000111101011100001

## Aufgabe 2

- Gib jeweils als Dezimalzahl an

a) 0 10000001 01010100011110101110001

b) 1 10001011 00000000001110001010010

- Lösung:

a)  $\approx 5,32$

b)  $\approx -4099,54$



- Die Darstellung von Dezimalzahlen als Fließkommazahl ist oft nicht exakt
  - Selbst “harmlose” Zahlen wie 0,4 sind in binärer Darstellung periodisch
  - Die speicherbare Stellenzahl ist immer begrenzt
- Das Rechnen mit Gleitkommazahlen führt unweigerlich zu (Rundungs)fehlern
  - Lies den [hier](#) verlinkten Artikel
- Tests auf Gleichheit können fehlschlagen, obwohl aus mathematischer Sicht gleiche Ergebnisse zu erwarten wären
  - Besser: Prüfen, ob das Ergebnis innerhalb eines gewissen Toleranzbereichs liegt

## Aufgabe 3

- Welche Ausgabe erwartest Du?

```
System.out.println(0.123);  
System.out.println(0.123 * 100.0 / 100.0 );  
System.out.println(0.123 * 100.0 / 100.0 == 0.123);
```

# Möglichkeiten für höhere Genauigkeit

- Verwendung einer Gleitkummacodierung mit der Basis 10 statt 2
  - Vermeidet periodische Darstellungen und resultierende Rundungsfehler
- Verwendung zusammengesetzter Datentypen
  - Beispiel: `BigInteger` und `BigDecimal` in Java
- BCD-Codierung (Binary Coded Decimal): Codiere jede der Ziffern 0 bis 9 einzeln binär (mit 4 Bit pro Ziffer)
  - Verschwendet Speicherplatz, wird heute kaum noch verwendet