

Einführung in die Programmiertechnik

Darstellung von Zahlen

Natürliche Zahlen: Darstellungsvarianten

- Darstellung als Text
 - Üblich, wenn keine Berechnung stattfinden soll
 - z.B. „Die Regionalbahn 28023 fährt um 16:02 in Golm ab.“
 - Die Zahlen 28023, 16 und 2 werden als Textzeichen repräsentiert
 - Speicheraufwand: 1B pro Dezimalziffer
- Variante: BCD (binary coded decimal)
 - Kodierung von 10 Ziffern: 4 Bit pro Ziffer
 - 2 Ziffern pro Byte
 - Beispiel: 28023 kodiert als 0010 0100 0000 0010 0011 (Hex: 02 80 23)
 - effizienter als Textdarstellung (0,5B pro Ziffer)
 - leicht in Dezimaldarstellung umrechenbar
 - arithmetische Operationen schwierig in Hardware realisierbar

Ganze Zahlen: Binärdarstellung

- Ziel: Ausschöpfung aller Bitkombinationen
 - Mit N Bits sollen 2^N Zahlen repräsentiert werden
 - üblich: Zahlen von 0 .. 2^N-1 (nichtnegative Zahlen, *unsigned*)
 - Zahlen von -2^{N-1} .. $2^{N-1}-1$ (ganze Zahlen, *signed*)
 - Darstellung reeller Zahlen wird später diskutiert
- Binärdarstellung: Bitfolgen werden im Binärsystem interpretiert
 - Bitfolge aus Null-Bits stellt die kleinste Zahl dar (0)
 - Bitfolge aus Eins-Bits stellt die größte Zahl dar (2^N-1)
- Bitpositionen innerhalb eines Bytes fest in Hardware verdrahtet (etwa: Bits 0 .. 7)
- Bitpositionen in Mehrbytezahlen abhängig von Prozessorarchitektur
 - little-endian: geringstwertiges Byte „zuerst“ im Speicher (auf kleinster Adresse)
 - Beispiel: Intel x86
 - big-endian: geringstwertiges Byte „zuletzt“ im Speicher
 - Beispiel: Motorola/IBM PowerPC

Arithmetische Operationen

- Addition analog dem Dezimalsystem

$$\begin{array}{r} 10010 \\ + 10011 \\ \hline 111001 \end{array}$$

- Allgemeiner: Rechenoperationen auf Basis von Assoziativ- und Distributivgesetzen

– Beispiel: Multiplikation $a_2a_1a_0 * b_2b_1b_0$

– $(4a_2+2a_1+a_0) * (4b_2+2b_1+b_0) =$

$$16a_2b_2+8(a_2b_1+a_1b_2)+4(a_2b_0+a_1b_1+a_0b_2)+2(a_1b_0+a_0b_1)+a_0b_0$$

Überlauf

- Darstellbar sind Zahlen von $0.. 2^N-1$
- Überlauf: Ergebnis ist größer als 2^N
- Beispiel: $11+8$, 4-Bit-Zahlen

$$\begin{array}{r} 1011 \\ + 1000 \\ \hline 10011 \end{array}$$

- Ergebnis wieder in 4 Bit: $11+8 \Rightarrow 3$
- Prozessor erkennt den Überlauf, zeigt ihn in „carry“-Bit an
 - abhängig von Programmiersprache wird carry-Bit verworfen oder führt zu einer Programmausnahme (exception)
- Verwerfen des Carry-Bits: Alle Operationen werden modulo 2^N ausgeführt

Ganze Zahlen

- Erweiterung der natürlichen Zahlen unter Hinzunahme *negativer* Zahlen
- Variante 1: Repräsentation der Zahl durch zusätzliches Vorzeichenbit (0: Zahl ist nichtnegativ, 1: Zahl ist negativ)
 - bei N Bits stehen N-1 Bits für die Zahl zur Verfügung
 - Beispiel (4 Bit)
 - $0000 = +0$, $0100 = +4$, $1000 = -0$, $1100 = -4$
 - Darstellbare Zahlen: $-(2^{N-1}-1) \dots 2^{N-1}-1$ (insgesamt 2^N-1 Zahlen)
 - negative Zahlen erkennt man am vordersten Bit
 - Problem: Rechenoperationen sind kompliziert durchzuführen (Addition, Subtraktion: 4 verschiedene Fälle)
 - Problem: zwei verschiedene Darstellungen der Zahl 0

Ganze Zahlen (2)

- Variante 2: Einerkomplement
 - negative Zahlen entstehen durch bitweise Negation aus den positiven Zahlen ($-x = \bar{x}$)
 - Beispiel (6 Bit) 01001 = 9, 10110 = -9
 - Darstellbare Zahlen: $-2^{N-1} \dots 2^{N-1}-1$ (insgesamt 2^N-1 Zahlen)
 - negative Zahlen erkennt man am vordersten Bit
 - gleiche Probleme wie bei Vorzeichendarstellung: arithmetische Operationen sind schwierig, zwei Darstellungen der 0 (000000 = +0, 111111 = -0)

Ganze Zahlen (3)

- Variante 3: Zweierkomplementdarstellung:
 - Negative Zahl durch Subtraktion von 2^N ($-x = 2^N - x$)
 - 2^N ist außerhalb des darstellbaren Bereichs; $2^N \equiv 0 \pmod{2^N}$
 - Alternative Bildungsregel: $-x = \bar{x} + 1$
 - Beispiel (4 Bit): $0011 = 3$, $-3 = 0011 + 1 = 1100 + 1 = 1101$
 - negative Zahlen erkennt man am vordersten Bit
 - Darstellbare Zahlen: $-2^{N-1} \dots 2^{N-1} - 1$ (insgesamt 2^N Zahlen)
 - Beispiel: 8 Bit: darstellbar sind die Zahlen von $-128 \dots 127$
 - Addition negativer Zahlen: entsprechend der Addition vorzeichenloser Zahlen
 - sowohl Überlauf als auch Unterlauf möglich
 - Subtraktion: Addition des negierten Werts ($a - b = a + (-b)$)
 - Problem: kleinste negative Zahl lässt sich nicht negieren

Ganze Zahlen (4)

- Variante 4: Darstellung negativer Zahlen durch Versatz (bias)
 - Bits werden zunächst als nichtnegative Zahl interpretiert
 - danach wird ein Versatz V subtrahiert
 - darstellbarer Bereich: $-V \dots 2^N - V - 1$ (insgesamt 2^N Zahlen)
 - Beispiel (4B, Versatz 7): $0000 = -7$, $0111 = 0$, $1000 = 1$, $1111 = 8$
 - Problem: Vorzeichen der Zahl u.U. nicht leicht erkennbar
 - Problem: 0 ist nicht durch eine Bitfolge von Nullbits repräsentiert
 - Verwendet u.a. als Teil der Gleitkommadarstellung

Wertebereiche

- Größe von Datentypen abhängig von Programmiersprache
 - C: $\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$
 - Beispiel: gcc, Linux, x86: short=16B, int=32B, long long=64B
 - Java: plattformunabhängig: byte=8B, short=16B, int=32B, long=64B

Größe (in B)	Wertebereich
8	-128..127
16	-32768..32767
32	$-2^{31}..2^{31}-1$ $2^{31} = 2147483648$
64	$-2^{63}..2^{63}-1$ $2^{63} = 9223372036854775808$

Große ganze Zahlen

- Idee: Aufhebung des Wertebereichs durch Verwendung einer variablen Anzahl Bytes
 - größerer Absolutwert \Rightarrow mehr Bytes
 - Wertebereich nur durch Hauptspeicher eingeschränkt
- Beispiele: **BigInt** in Java, **long** in Python
- arithmetische Operationen nicht durch Prozessor unterstützt

Reelle Zahlen

- überabzählbar unendlich viele reelle Zahlen von 0..1
 - ⇒ nur endlich viele Zahlen mit endlichem Speicher unterscheidbar
- Variante 1: Darstellung rationaler Zahlen durch Zähler und Nenner
 - etwa: Zähler ist ganze Zahl, Nenner ist natürliche Zahl
 - Problem: elementare Rechenoperationen führen leicht aus dem Bereich der darstellbaren Nenner heraus
- Variante 2: Darstellung mit festem Nenner (etwa: 100)
 - Festkommazahlen
 - arithmetische Operationen müssen auf nächste darstellbare Zahl runden
 - verbreitet in Finanzmathematik
 - Beispiel: `Decimal` in Java
- Variante 3: Gleitkommazahlen

Gleitkommazahlen

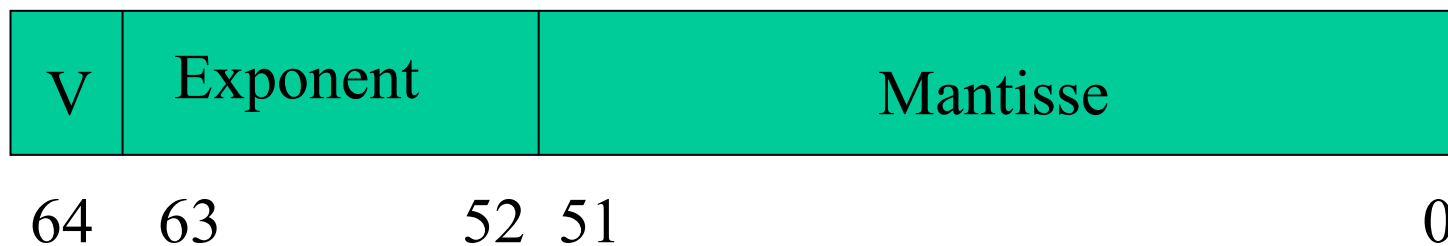
- Ziel: möglichst großes Intervall darstellbarer Zahlen, möglichst viele Nachkommastellen
 - aber nicht gleichzeitig: für „große“ Zahlen werden meist keine Nachkommastellen benötigt
 - tatsächliche Zahl wird durch Näherungswert ersetzt
- Gleitkommadarstellung heute meist nach IEEE-754 (IEC 60559:1989)
- Idee: feste Zahl signifikanter Stellen, mit Faktor skalierbar
 - negative Zahlen durch Vorzeichenbit repräsentiert

Gleitkommazahlen (2)

- jede Zahl hat 3 Bestandteile:
 - Vorzeichen V (1 Bit)
 - Mantisse M (signifikante Stellen)
 - Exponent E (für Faktor zur Basis 2), ganze Zahl in Binärdarstellung
- Dargestellte Zahl ist
$$(-1)^V * M * 2^E$$
- Normalisierung: Durch Wahl von E kann man die Mantisse in die Form $1.a_1a_2a_3a_4\dots$ bringen
 - Zur Speicherung von M genügt es, nur die Nachkommastellen zu speichern (normalisierte Zahlen)
- Exponent: Bias-Darstellung
 - Bias-Darstellung erlaubt einfacheres Vergleichen von zwei Gleitkommazahlen
 - Spezialwerte für den Exponenten: minimaler Wert (alles 0) und maximaler Wert (alles 1)

Gleitkommazahlen (3)

- Übliche Formate: 32 Bit, 64 Bit
- 32-Bit-Darstellung (Java float)
 - 1 Vorzeichenbit, 8 Bit Exponent (Bias 127), 23 Bit Mantisse
 - Wertebereich normalisierter Zahlen
 - negativ: $-(2-2^{-23})x2^{127} \dots -2^{-126}$
 - positiv: $2^{-126} \dots (2-2^{-23})x2^{127}$
- 64-Bit-Darstellung (Java double)
 - 1 Vorzeichenbit, 11 Bit Exponent (Bias 1023), 52 Bit Mantisse



Gleitkommazahlen (4)

- Spezialfälle: Exponent 0000...000, 1111...111
- Zahl 0.0: Exponent 0, Mantisse 0
 - Vorzeichen 0: +0.0 (also: 0.0 wird 32/64 Nullbits dargestellt)
 - Vorzeichen 1: -0.0
- +/- Infinity: Exponent maximal (32-bit: FF_{16}), Mantisse 0
 - „kanonische“ Rechenregeln für $\pm\infty$
 - Darstellung in Programmiersprachen uneinheitlich
 - Java: `Float.POSITIVE_INFINITY`
- NaN (not-a-number): Exponent maximal, Mantisse $\neq 0$
 - zur Anzeige von Bereichsverletzungen (etwa: Wurzel aus negativen Zahlen)
 - Berechnungen mit NaN als Eingabe liefern meist NaN als Ergebnis
- denormalisierte Zahlen: Exponent 0, Mantisse $\neq 0$
 - füllen Lücke zwischen 0 und kleinster normalisierter Zahl
 - Interpretiert als $0.a_1a_2a_3a_4\dots$

Gleitkommazahlen (5)

- Betriebsmodi von IEEE-754
 - Rundungsmodus: Anpassung des Berechnungsergebnisses an „nächste“ darstellbare Zahl
 - towards zero
 - towards negative infinity
 - towards positive infinity
 - unbiased (zur nächstliegenden Zahl, bei Zahl in der Mitte zu der Zahl, deren letztes signifikantes Bit 0 ist)
 - Fehlerbehandlung
 - Generierung von Ausnahmen
 - Generierung von Spezialwerten (infinity, NaN)
- „strikte“ Implementierung
 - Prozessorhardware weicht oft in Details von IEEE-754 ab
 - Javas Schlüsselwort **strictfp** erzwingt Konformität

Gleitkommazahlen (6)

- Rundundseffekte: Zahlen mit endlicher Darstellung im Dezimalsystem haben oft keine exakte Darstellung als Gleitkommazahl
 - Beispiel: $0,1 \equiv 1/10$ hat periodische Darstellung im Binärsystem
 - $0.00011001100110011\dots$
 - Ausnahmen: Brüche, deren Nenner eine Zweierpotenz ist
 - $0,25 \equiv \frac{1}{4} \equiv 0.01_2$
- Gleitkommadarstellung sucht nächstliegende darstellbare Zahl
 - Beispiel: 0.1 , float

V	Exponent	Mantisse	Dezimalwert
0	01111011	10011001100110011001100	0,0999999940...
0	01111011	10011001100110011001101	0,1000000015

Andere Zahlen

- komplexe Zahlen: Darstellung als Paar (Realteil, Imaginärteil)
- Intervallarithmetik: Darstellung einer Zahl als Intervall
 - Berücksichtigung von Rundungen in vorigen Operationen
- ...