Rechnerstrukturen und Betriebssysteme

64-040 Vorlesung RSB
Kapitel 4: Arithmetische Operationen

Norman Hendrich

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Technische Aspekte Multimodaler Systeme

Wintersemester 2025/2026

Gliederung

Wiederholung: Stellenwertsystem („Radixdarstellung“)

- Wahl einer geeigneten Zahlenbasis b („Radix“)
  - 10: Dezimalsystem
  - 16: Hexadezimalsystem (Sedezimalsystem)
  - 2: Dualsystem
- Menge der entsprechenden Ziffern $\{0, 1, \ldots, b-1\}$
- inklusive einer besonderen Ziffer für den Wert Null
- Auswahl der benötigten Anzahl n von Stellen

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i$$

b Basis a_i Koeffizient an Stelle i

- universell verwendbar, für beliebig große Zahlen

Integer-Datentypen in C und Java

C:

- Zahlenbereiche definiert in Headerdatei `/usr/include/limits.h`
  `LONG_MIN`, `LONG_MAX`, `ULONG_MAX` etc.
- Zweierkomplement (`signed`), Ganzzahl (`unsigned`)
- die Werte sind plattformabhängig !

Java:

- 16-bit, 32-bit, 64-bit Zweierkomplementzahlen
- Wrapper-Klassen `Short`, `Integer`, `Long`

```
Short.MAX_VALUE    =       32767
Integer.MIN_VALUE  = -2147483648
Integer.MAX_VALUE  =  2147483647
Long.MIN_VALUE     = -9223372036854775808L
...
```

- Werte sind für die Sprache fest definiert

Addition im Dualsystem

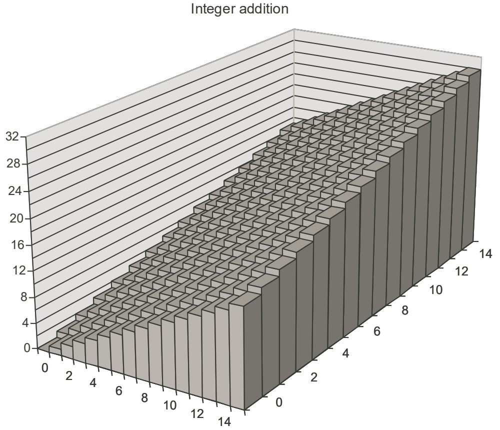
- funktioniert genau wie im Dezimalsystem
- Addition mehrstelliger Zahlen erfolgt stellenweise
- Additionsmatrix

$$\begin{array}{c|cc} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 10 \end{array}$$

- Beispiel

$$\begin{array}{rr} & 1011\,0011 \\ + & 0011\,1001 \\ \hline \text{Ü} & 11\quad\;\; 11\;\; \\ \hline\hline & 1110\,1100 \end{array} \qquad \begin{array}{rr} = & 179 \\ = & 57 \\ \hline & 11\;\; \\ \hline\hline = & 236 \end{array}$$

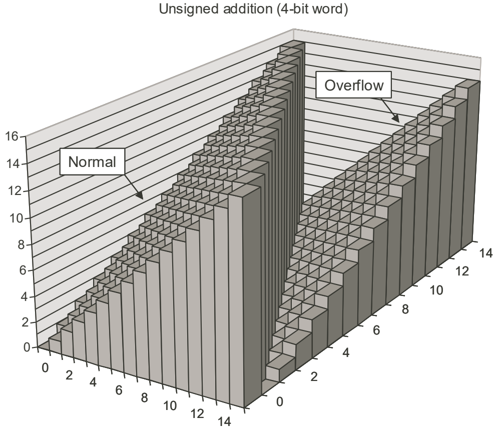
unsigned Addition: Visualisierung



[1]

- Wortbreite der Operanden ist w, hier 4-bit
- Zahlenbereich der Operanden x, y ist $0 \ldots (2^w - 1)$ (0..15)
- Zahlenbereich des Resultats s ist $0 \ldots (2^{w+1} - 2)$ (0..30)

unsigned Addition: Visualisierung (cont.)



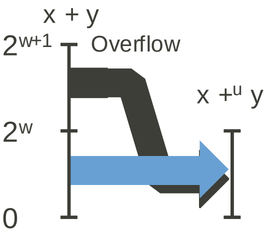
[1]

- Wortbreite der Operanden und des Resultats ist w

$\Rightarrow$ Überlauf, sobald das Resultat größer als $(2^w - 1)$

$\Rightarrow$ oberstes Bit geht verloren $(16..30) \rightarrow (0..14)$

unsigned Addition: Überlauf



- Wortbreite ist w
- Zahlenbereich der Operanden x, y ist $0 \ldots (2^w - 1)$
- Zahlenbereich des Resultats s ist $0 \ldots (2^{w+1} - 2)$
- Werte $s \geq 2^w$ werden in den Bereich $0 \ldots 2^w - 1$ abgebildet

Subtraktion im Dualsystem

- Subtraktion mehrstelliger Zahlen erfolgt stellenweise
- (Minuend - Subtrahend), Überträge berücksichtigen

- Beispiel

	1011 0011	=	179
-	0011 1001	=	57
Ü	1111		
	111 1010	=	122

- Alternative: Subtraktion durch Addition des *b*-Komplements ersetzen

Subtraktion mit *b*-Komplement

- bei Rechnung mit fester Stellenzahl n gilt:

$$K_b(z) + z = b^n \quad = 0$$

  weil b^n gerade nicht mehr in n Stellen hineinpasst

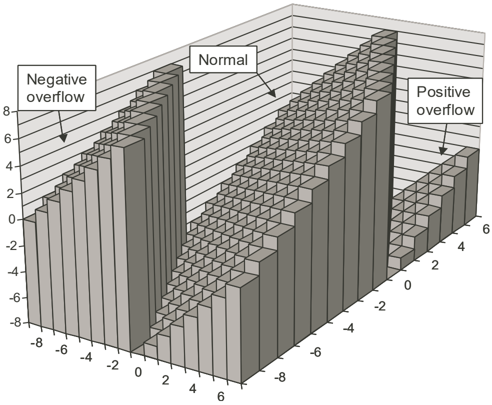
- also gilt für die Subtraktion auch:

$$x - y = x + K_b(y)$$

⇒ Subtraktion kann durch Addition des b-Komplements ersetzt werden!
Voraussetzung: begrenzte Stellenanzahl

- und für Integerzahlen gilt außerdem

$$x - y = x + K_{b-1}(y) + 1$$

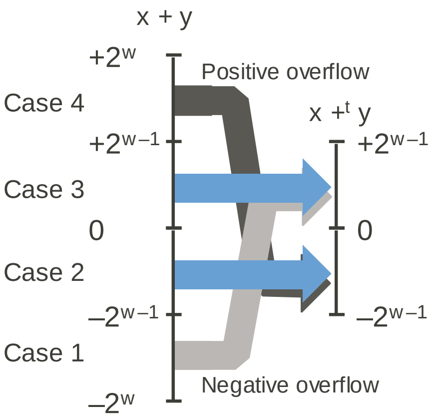


[1]

- Wortbreite der Operanden ist w, hier 4-bit
- Zahlenbereich der Operanden x, y ist $-2^{w-1} \ldots (2^{w-1} - 1)$
- Zahlenbereich des Resultats s ist $-2^{w} \ldots (2^{w} - 2)$

$\Rightarrow$ Überlauf in beide Richtungen möglich

signed Addition: Überlauf



- Wortbreite des Resultats ist w: Bereich $-2^{w-1} \ldots (2^{w-1} - 1)$
- Überlauf positiv wenn Resultat $\geq \ 2^{w-1}$: Summe negativ
  –"– negativ –"– $< -2^{w-1}$: Summe positiv

Überlauf: Erkennung

- Erkennung eines Überlaufs bei der Addition?
- wenn beide Operanden das gleiche Vorzeichen haben und sich das Vorzeichen des Resultats unterscheidet

- Java-Codebeispiel

```java
int a, b, sum;          // operands and sum
boolean ovf;            // ovf flag indicates overflow

sum = a + b;
ovf = ((a < 0) == (b < 0)) && ((a < 0) != (sum < 0));
```

Subtraktion mit Einer- und Zweierkomplement

- Subtraktion ersetzt durch Addition des Komplements

Dezimal	1-Komplement	2-Komplement
10	0000 1010	0000 1010
+(-3)	1111 1100	1111 1101
+7	1 0000 0110	1 0000 0111
Übertrag:	addieren +1	verwerfen
	0000 0111	0000 0111

- **(b-1)-Komplement** der Zahl z

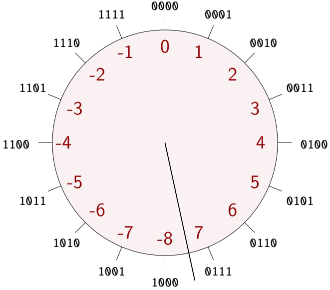
$$K_{b-1}(z) = \begin{cases} b^n - z - b^{-m} & \text{für } z \neq 0 \\ 0 & \text{für } z = 0 \end{cases}$$

b-Komplement der Zahl z

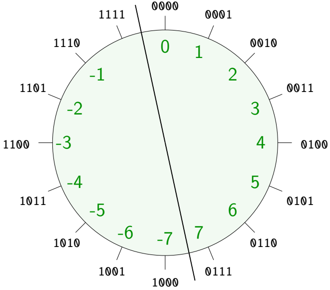
$$K_b(z) = \begin{cases} b^n - z & \text{für } z \neq 0 \\ 0 & \text{für } z = 0 \end{cases}$$

Veranschaulichung: Zahlenkreis

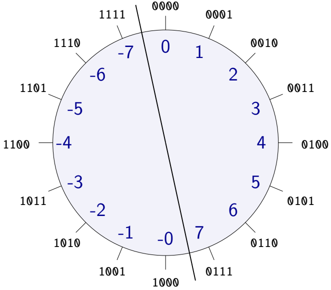
Beispiel für 4-bit Zahlen



2-Komplement



1-Komplement



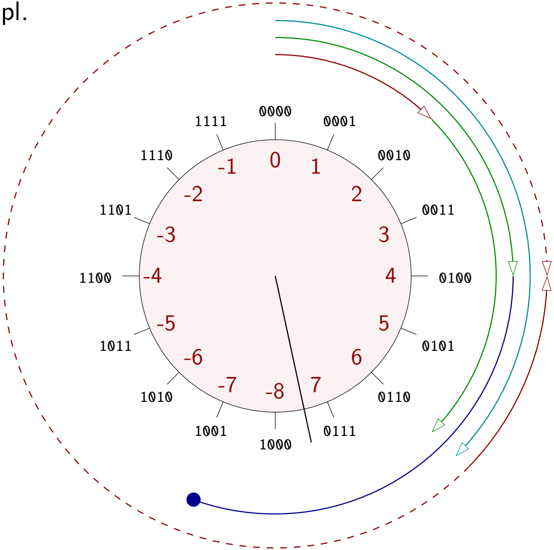
Betrag+Vorzeichen

MSB $\widehat{=}$ Vorzeichen

- Komplement-Arithmetik als Winkeladdition
- Web-Anwendung: *Visualisierung im Zahlenkreis* (JavaScript, aus [3])

Zahlenkreis: Addition, Subtraktion

2-Kompl.

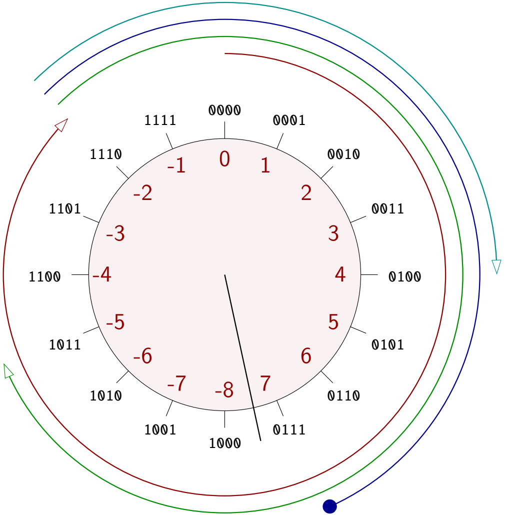


0010+0100=0110
0100+0101=1001
0110-0010=0100

0010 1110
0100
0101
0110

Zahlenkreis: Addition, Subtraktion (cont.)

2-Kompl.

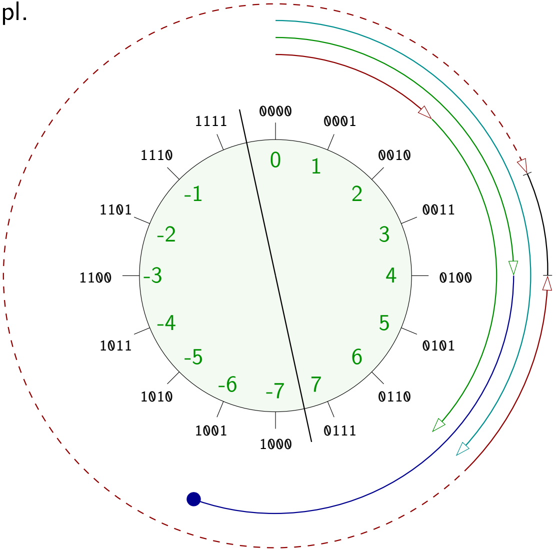


1110+1101=1011
1110+1001=0111
1110+0110=0100

1110
1101
1001
0110

Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.

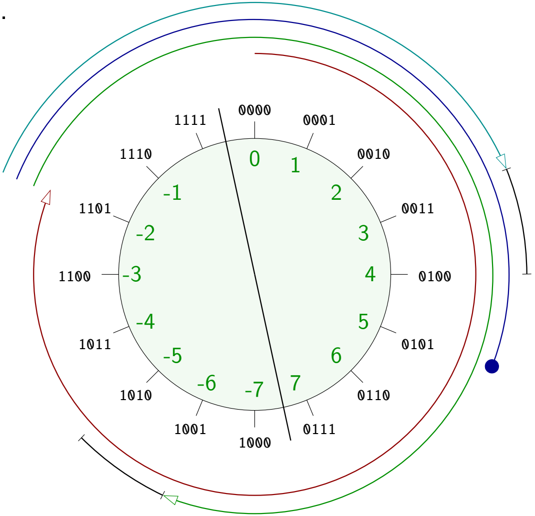


0010+0100=0110
0100+0101=1001
0110+1101+1=0100

0010 1101
0100
0101
0110

Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.



1101+1100+1=1010

1101+1000=0101

1101+0110+1=0100

1101

1100

1000

0110

in C: unsigned Zahlen

- für hardwarenahe Programme und Treiber
- für modulare Arithmetik („multi-precision arithmetic“)
- aber evtl. ineffizient (vom Compiler schlecht unterstützt)

- Vorsicht vor solchen Fehlern

```
unsigned int i, cnt = ...;
for( i = cnt-2; i >= 0; i-- ) {
  a[i] += a[i+1];
}
```

in C: Casting-Regeln

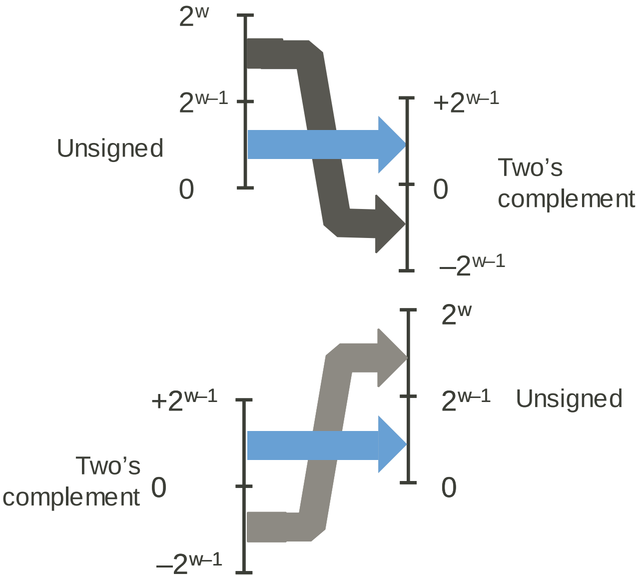
- Bit-Repräsentation wird nicht verändert
- kein Effekt auf positiven Zahlen
- Negative Werte als (große) positive Werte interpretiert

```
short int          x  =  15213;
unsigned short int ux = (unsigned short) x; // 15213

short int          y  = -15213;
unsigned short int uy = (unsigned short) y; // 50323
```

- Schreibweise für Konstanten:
  - ohne weitere Angabe: signed
  - Suffix „U“ für unsigned: `0U`, `4294967259U`

in C: unsigned / signed Interpretation



in C: Vorsicht bei Typumwandlung

- Arithmetische Ausdrücke:
  - bei gemischten Operanden: Auswertung als unsigned
  - auch für die Vergleichsoperationen <, >, ==, <=, >=
  - Beispiele für Wortbreite 32-bit:

Konstante 1	Relation	Konstante 2	Auswertung	Resultat	
`0`	`==`	`0U`	unsigned	1	
`-1`	`<`	`0`	signed	1	
`-1`	`<`	`0U`	unsigned	`0`	Fehler
`2147483647`	`>`	`-2147483648`	signed	1	
`2147483647U`	`>`	`-2147483648`	unsigned	`0`	
`2147483647`	`>`	`(int) 2147483648U`	signed	1	
`-1`	`>`	`-2`	signed	1	
`(unsigned) -1`	`>`	`-2`	unsigned	1	

Sign-Extension

- Gegeben: w-bit Integer x
- Umwandeln in $w + k$-bit Integer x' mit gleichem Wert?

- **Sign-Extension**: Vorzeichenbit kopieren

$$x' = x_{w-1}, \ldots x_{w-1}, x_{w-1}, x_{w-2}, \ldots x_0$$

- Zahlenbeispiele

0110	4-bit signed: +6
0000 0110	8-bit signed: +6
0000 0000 0000 0110	16-bit signed: +6
1110	4-bit signed: -2
1111 1110	8-bit signed: -2
1111 1111 1111 1110	16-bit signed: -2

Java Puzzlers No.5

J. Bloch, N. Gafter: *Java Puzzlers: Traps, Pitfalls, and Corner Cases.* Addison-Wesley, 2005


```
public static void main( String[] args ) {
  System.out.println(
    Long.toHexString( 0x100000000L + 0xcafebabe ));
}
```

- Programm addiert zwei Konstanten, Ausgabe in Hex-Format
- Was ist das Resultat der Rechnung?

Java Puzzlers No.5

J. Bloch, N. Gafter: *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, 2005


```
public static void main( String[] args ) {
  System.out.println(
    Long.toHexString( 0x100000000L + 0xcafebabe ));
}
```

- Programm addiert zwei Konstanten, Ausgabe in Hex-Format
- Was ist das Resultat der Rechnung?

```
  0xffffffffcafebabe   sign-extension!
  0x0000000100000000
  ------------------
  Ü 11111110
  ==================
    00000000cafebabe
```

Ariane-5 Absturz

- Erstflug der Ariane-5 („V88") am 04. Juni 1996
- Kurskorrektur wegen vermeintlich falscher Fluglage
- Selbstzerstörung der Rakete nach 36,7 Sekunden
- Schaden ca. 635 M€
  (teuerster Softwarefehler der Geschichte?)

- bewährte Software von Ariane-4 übernommen
- aber Ariane-5 viel schneller als Ariane-4
- 64-bit Gleitkommawert für horizontale Geschwindigkeit
- Umwandlung in 16-bit Integer: dabei Überlauf

- `de.wikipedia.org/wiki/Ariane_V88`

Multiplikation im Dualsystem

- funktioniert genau wie im Dezimalsystem
- $p = a \cdot b$ mit Multiplikator a und Multiplikand b
- Multiplikation von a mit je einer Stelle des Multiplikanten b
- Addition der Teilterme

- Multiplikationsmatrix – sehr einfach: $\cdot 0 \;/\; \cdot 1$

$\cdot$	0	1
0	0	0
1	0	1

- Beispiel

$$
\begin{array}{rrlll}
10110011 & \cdot & 1101 & = 179 \cdot 13 & = 2327 \\
\hline
10110011 & & 1 & & = 1001\,0001\,0111 \\
10110011 & & 1 & & = 0x917 \\
00000000 & & 0 & & \\
10110011 & & 1 & & \\
\hline
\text{Ü}\;\; 11101111 & & & & \\
\hline\hline
100100010111 & & & &
\end{array}
$$

unsigned Multiplikation

- bei Wortbreite w bit
- Zahlenbereich der Operanden: $0 \ldots (2^w - 1)$
- Zahlenbereich des Resultats: $0 \ldots (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

$\Rightarrow$ bis zu $2w$ bits erforderlich

- C: Resultat enthält nur die unteren w bits
- Java: keine unsigned Integer
- Hardware: teilweise zwei Register *high*, *low* für die oberen und unteren Bits des Resultats

- Zahlenbereich der Operanden: $-2^{w-1} \ldots (2^{w-1} - 1)$
- Zahlenbereich des Resultats: $-2^{w-1} \cdot (2^{w-1} - 1) \ldots (2^{2w-2})$

$\Rightarrow$ bis zu $2w$ bits erforderlich

- C, Java: Resultat enthält nur die unteren w bits
- Überlauf wird ignoriert

```
int i = 100*200*300*400; // -1894967296
```

- Repräsentation der unteren Bits des Resultats entspricht der unsigned Multiplikation

$\Rightarrow$ kein separater Algorithmus erforderlich
Beweis: siehe Bryant, O'Hallaron: Abschnitt 2.3.5 [1]

Java Puzzlers No. 3

J. Bloch, N. Gafter: *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, 2005

```
public static void main( String args[] ) {
  final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
  final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
  System.out.println(  MICROS_PER_DAY / MILLIS_PER_DAY );
}
```

Java Puzzlers No. 3

J. Bloch, N. Gafter: *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, 2005


```
public static void main( String args[] ) {
  final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
  final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
  System.out.println(  MICROS_PER_DAY / MILLIS_PER_DAY );
}
```

- druckt den Wert 5, nicht 1000!
- `MICROS_PER_DAY` wird mit 32-bit berechnet, dabei Überlauf
- Konvertierung nach 64-bit `long` erst bei der Zuweisung

⇒ `long`-Konstante schreiben: `24L * 60 * 60 * 1000 * 1000`

Division im Dualsystem

- $d = a/b$ mit Dividend a und Divisor b

- funktioniert genau wie im Dezimalsystem
- schrittweise Subtraktion des Divisors
- Berücksichtigen des „Stellenversetzens“

- in vielen Prozessoren nicht (oder nur teilweise) durch Hardware unterstützt
- daher deutlich langsamer als Multiplikation

Division im Dualsystem (cont.)

- Beispiele

$$100_{10}/3_{10} = 110\,0100_2/11_2 = 10\,0001_2$$

$$\begin{array}{llr}
1100100 \;/\; 11 = & 0100001 & \\
1 & 0 & \\
11 & \;1 & \\
-11 & & \\
\hline
\;\;0 & \;\;0 & \\
\;\;\;0 & \;\;\;0 & \\
\;\;\;\;1 & \;\;\;\;0 & \\
\;\;\;\;10 & \;\;\;\;\;0 & \\
\;\;\;\;100 & \;\;\;\;\;\;1 & \\
\;\;\;-11 & & \\
\hline
\;\;\;\;\;\;1 & \;\;\;\;\;\;\;1 & \text{(Rest)}
\end{array}$$

Division im Dualsystem (cont.)

$$91_{10}/13_{10} = 101\,1011_2/1101_2 = 111_2$$

$$\begin{array}{lr}
1011011 \;\;/\;\; 1101 = & 0111 \\
1011 & 0 \\
10110 & 1 \\
-1101 & \\
\hline
10011 & 1 \\
-1101 & \\
\hline
01101 & 1 \\
-1101 & \\
\hline
0 &
\end{array}$$

Höhere mathematische Funktionen

Berechnung von $\sqrt{x}$, $\log x$, $\exp x$, $\sin x$ ...?

- Approximation über Polynom (Taylor-Reihe) bzw. über rationale Funktionen
  - vorberechnete Koeffizienten für höchste Genauigkeit
  - Ausnutzen mathematischer Identitäten für Skalierung

- Sukzessive Approximation über iterative Berechnungen
  - Beispiele: Quadratwurzel und Reziprok-Berechnung
  - häufig schnelle (quadratische) Konvergenz

- Berechnungen erfordern nur die Grundrechenarten

Reziprokwert: Iterative Berechnung von $1/x$

- Berechnung des Reziprokwerts $y = 1/x$ über

$$y_{i+1} = y_i \cdot (2 - x \cdot y_i)$$

- geeigneter Startwert y_0 als Schätzung erforderlich

- Beispiel $x = 3,\ y_0 = 0{,}5$:

$$\begin{aligned}
y_1 &= 0{,}5 \cdot (2 - 3 \cdot 0{,}5) &&= 0{,}25 \\
y_2 &= 0{,}25 \cdot (2 - 3 \cdot 0{,}25) &&= 0{,}3125 \\
y_3 &= 0{,}3125 \cdot (2 - 3 \cdot 0{,}3125) &&= 0{,}33203125 \\
y_4 &= 0{,}3332824 \\
y_5 &= 0{,}3333333332557231 \\
y_6 &= 0{,}3333333333333333
\end{aligned}$$

Quadratwurzel: Heron-Verfahren für $\sqrt{x}$
Babylonisches Wurzelziehen

- Sukzessive Approximation von $y = \sqrt{x}$ gemäß

$$y_{n+1} = \frac{y_n + x/y_n}{2}$$

- quadratische Konvergenz in der Nähe der Lösung
- Anzahl der gültigen Stellen verdoppelt sich mit jedem Schritt

- aber langsame Konvergenz fernab der Lösung
- Lookup-Tabelle und Tricks für brauchbare Startwerte y_0

Transzendente Funktionen: $\sin(x)$ usw.

- Reduktion auf geeigneten (kleinen) Wertebereich, wenn möglich
- dort dann Approximation durch Polynom oder rationale Funktion

  - $\sin(x + n \cdot 2\pi) = \sin(x)$, $\sin(x) = -\sin(-x)$
  - also im ersten Schritt $x \to x \mod 2\pi$
  - Approximation für $\sin(y)$, Wertebereich $0 \leq y \leq \pi$

  - $\exp(a + b) = \exp(a) \cdot \exp(b)$
  - $\exp x = \exp(\lceil x \rceil) \cdot \exp(\lfloor x \rfloor)$ (Split x in Vor-/Nachkommastellen)
  - Tabelle mit Werten für $\exp(n)$ mit Integer n
  - Polynom für $\exp(y), 0 \leq y \leq 1$

- vorberechnete Koeffizienten für maximale Genauigkeit
- Genauigkeit typisch ± 2ULP über gesamten Wertebereich

Vorsicht mit Rundungsfehlern/Auslöschung

- $\exp(x) = 1 + x + x^2/2! + x^3/3! + \ldots$

```
public class NaiveTaylorExponential {                // don't use this!!!
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double sum  = 0.0;
        double term = 1.0;
        for (int i = 1; sum != sum + term; i++) {
            sum  = sum + term;
            term = term * x / i;
            System.out.println(i + " " + term + " " + sum);
        }
        System.out.println(sum);
        System.out.println(Math.exp(x));
    }
}
```

http://introcs.cs.princeton.edu/java/91float/Exponential.java.html

Vorsicht mit Rundungsfehlern/Auslöschung


```
/* Naively compute the e^x using the Taylor series:
*  e^x = 1 + x + x^2/2! + x^3/3! + ...
*  and compare with Math.exp(). Catastrophic cancelation occurs
*  if x is a large negative number.  */
% java NaiveTaylorExponential 1
2.718281828459045
2.718281828459045

% java NaiveTaylorExponential 50
5.184705528587075E21
5.184705528587072E21

% java NaiveTaylorExponential -1
0.36787944117144245
0.36787944117144233

% java NaiveTaylorExponential -50
11072.93338289197
1.9287498479639178E-22
```

Welche mathematischen Eigenschaften gelten bei der Informationsverarbeitung / in der gewählten Repräsentation?

Beispiele:

- Gilt $x^2 \geq 0$?
  - float: ja
  - signed integer: nein
- Gilt $(x + y) + z = x + (y + z)$?
  - integer: ja
  - float: nein

    1.0E20 + (-1.0E20 + 3.14) = 0

Festkomma Addition

unsigned Arithmetik

- Wortbreite auf w begrenzt
- kommutative Gruppe / Abel'sche Gruppe

  - Abgeschlossenheit $0 \leq a \oplus_w^u b \leq 2^w - 1$
  - Kommutativgesetz $a \oplus_w^u b = b \oplus_w^u a$
  - Assoziativgesetz $a \oplus_w^u (b \oplus_w^u c) = (a \oplus_w^u b) \oplus_w^u c$
  - neutrales Element $a \oplus_w^u 0 = a$
  - Inverses $a \oplus_w^u \overline{a} = 0; \overline{a} = 2^w - a$

Festkomma Addition (cont.)

signed Arithmetik 2-Komplement

- Wortbreite auf w begrenzt
- signed und unsigned Addition sind auf Bit-Ebene identisch

  $a \oplus^s_w b = U2S(S2U(a) \oplus^u_w S2U(b))$

$\Rightarrow$ isomorphe Algebra zu $\oplus^u_w$

- kommutative Gruppe / Abel'sche Gruppe
  - Abgeschlossenheit $-2^{w-1} \leq a \oplus^s_w b \leq 2^{w-1} - 1$
  - Kommutativgesetz $a \oplus^s_w b = b \oplus^s_w a$
  - Assoziativgesetz $a \oplus^s_w (b \oplus^s_w c) = (a \oplus^s_w b) \oplus^s_w c$
  - neutrales Element $a \oplus^s_w 0 = a$
  - Inverses $a \oplus^s_w \overline{a} = 0; \quad \overline{a} = -a, \; a \neq -2^{w-1}$

    $a, \; a = -2^{w-1}$

Festkomma Multiplikation

unsigned Arithmetik

- Wortbreite auf w begrenzt
- Modulo-Arithmetik $a \otimes_w^u b = (a \cdot b)\, mod\, 2^w$
- $\otimes_w^u$ und $\oplus_w^u$ bilden einen kommutativen Ring
  - $\oplus_w^u$ ist eine kommutative Gruppe
  - Abgeschlossenheit $0 \leq a \otimes_w^u b \leq 2^w - 1$
  - Kommutativgesetz $a \otimes_w^u b = b \otimes_w^u a$
  - Assoziativgesetz $a \otimes_w^u (b \otimes_w^u c) = (a \otimes_w^u b) \otimes_w^u c$
  - neutrales Element $a \otimes_w^u 1 = a$
  - Distributivgesetz $a \otimes_w^u (b \oplus_w^u c) = (a \otimes_w^u b) \oplus_w^u (a \otimes_w^u c)$

Festkomma Multiplikation (cont.)

signed Arithmetik

- signed und unsigned Multiplikation sind auf Bit-Ebene identisch
- ...

isomorphe Algebren

- unsigned Addition und Multiplikation; Wortbreite w
- signed Addition und Multiplikation; Wortbreite w 2-Komplement
- isomorph zum Ring der ganzen Zahlen *modulo* 2^w

- Ordnungsrelation im Ring der ganzen Zahlen
  - $a > 0 \longrightarrow a + b > b$
  - $a > 0, b > 0 \longrightarrow a \cdot b > 0$
  - diese Relationen gelten nicht bei Rechnerarithmetik! Überlauf

Gleitkomma Addition
Vergleich mit kommutativer Gruppe

- Abgeschlossen? — Ja
- Kommutativ? — Ja
- Assoziativ? — Nein
  (Überlauf, Rundungsfehler)
- Null ist neutrales Element? — Ja
- Inverses Element existiert? — Fast
  (außer für NaN und Infinity)

- Monotonie? $a \geq b \longrightarrow (a + c) \geq (b + c)$ — Fast
  (außer für NaN und Infinity)

Gleitkomma Multiplikation
Vergleich mit kommutativem Ring

- Abgeschlossen? Ja
  (aber Infinity oder NaN möglich)
- Kommutativ? Ja
- Assozativ? Nein
  (Überlauf, Rundungsfehler)
- Eins ist neutrales Element? Ja
- Distributivgesetz? Nein

- Monotonie? $a \geq b; c \geq 0 \longrightarrow (a \cdot c) \geq (b \cdot c)$ Fast
  (außer für NaN und Infinity)

Literatur

[1] Randal E. Bryant and David R. O'Hallaron.
Computer systems – A programmers perspective.
Pearson Education Limited, Harlow, third, global ed. edition, 2015.

[2] Israel Koren.
Computer Arithmetic Algorithms.
CRC Press, Boca Raton, Fla., second edition, 2001.

[3] Laszlo Korte.
Bsc thesis: Tams tools for elearning.
Bachelorarbeit, Universität Hamburg, Fachbereich Informatik, AB TAMS, 2016.

[4] Amos R. Omondi.
Computer Arithmetic Systems – Algorithms, Architecture and Implementations.
Prentice-Hall International (UK) Limited, Hemel Hempstead, Herfordshire, 1994.

Literatur (cont.)

[5] Otto Spaniol.
Arithmetik in Rechenanlagen.
B. G. Teubner, Stuttgart, 1976.

[6] Andrew S. Tanenbaum and Todd Austin.
Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.
Pearson Deutschland GmbH, Hallbergmoos, sixth edition, 2014.