

# 64-040 Modul InfB-RS: Rechnerstrukturen

[https://tams.informatik.uni-hamburg.de/  
lectures/2015ws/vorlesung/rs](https://tams.informatik.uni-hamburg.de/lectures/2015ws/vorlesung/rs)

– Kapitel 4 –

Norman Hendrich



Universität Hamburg  
Fakultät für Mathematik, Informatik und Naturwissenschaften  
Fachbereich Informatik

**Technische Aspekte Multimodaler Systeme**

Wintersemester 2015/2016



# Kapitel 4

## Arithmetik

Addition und Subtraktion

Multiplikation

Division

Höhere Funktionen

Mathematische Eigenschaften

Literatur



## Wiederholung: Stellenwertsystem („Radixdarstellung“)

- ▶ Wahl einer geeigneten Zahlenbasis  $b$  („Radix“)
  - ▶ 10: Dezimalsystem
  - ▶ 16: Hexadezimalsystem (Sedezimalsystem)
  - ▶ 2: Dualsystem
- ▶ Menge der entsprechenden Ziffern  $\{0, 1, \dots, b - 1\}$
- ▶ 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

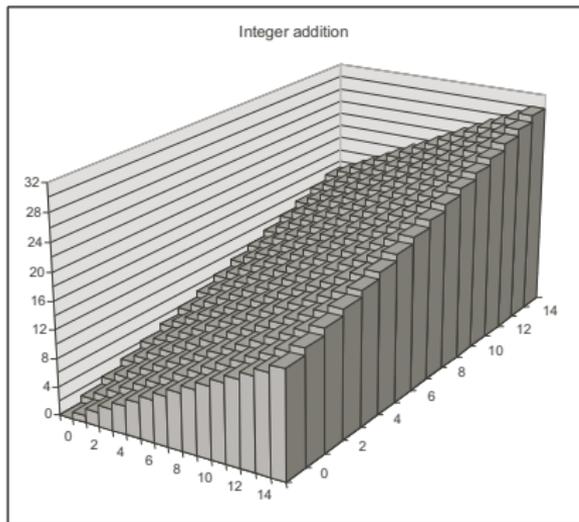
- ▶ funktioniert genau wie im Dezimalsystem
- ▶ Addition mehrstelliger Zahlen erfolgt stellenweise
- ▶ Additionsmatrix:

$$\begin{array}{r|l}
 + & 0 \ 1 \\
 \hline
 0 & 0 \ 1 \\
 1 & 1 \ 10
 \end{array}$$

- ▶ Beispiel

$$\begin{array}{r}
 1011\ 0011 \\
 + 0011\ 1001 \\
 \hline
 \overset{\cdot}{\underset{\cdot}{0}} \ 11 \ 11 \\
 \hline
 1110\ 1100
 \end{array}
 \qquad
 \begin{array}{r}
 = 179 \\
 = 57 \\
 \hline
 11 \\
 \hline
 = 236
 \end{array}$$

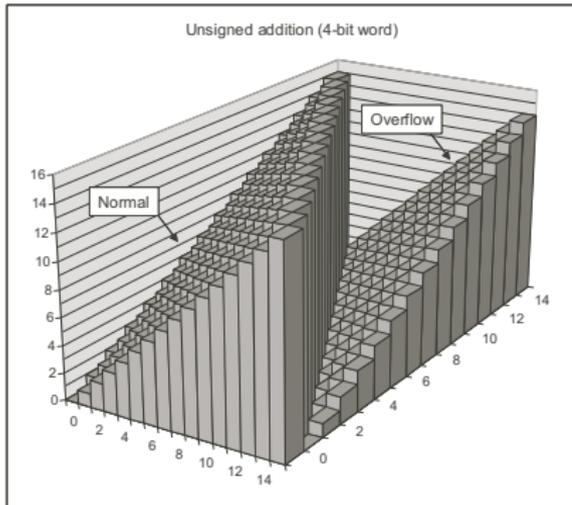
# unsigned Addition: Visualisierung



[BO14]

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

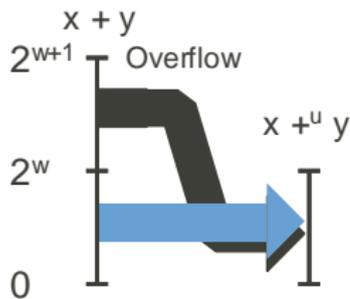
## unsigned Addition: Visualisierung (cont.)



[BO14]

- ▶ Wortbreite der Operanden **und des Resultats** ist  $w$
- ⇒ Überlauf, sobald das Resultat größer als  $(2^w - 1)$
- ⇒ oberstes Bit geht verloren

## unsigned Addition: Überlauf



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



## Subtraktion im Dualsystem

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

- ▶ Beispiel

$$\begin{array}{r}
 1011\ 0011 \\
 - 0011\ 1001 \\
 \hline
 \overset{\circ}{\underset{\circ}{U}}\ 1111 \\
 \hline
 111\ 1010
 \end{array}
 \qquad
 \begin{array}{r}
 = 179 \\
 = 57 \\
 \hline
 = 122
 \end{array}$$

- ▶ Alternative: Ersetzen der Subtraktion durch Addition des  $b$ -Komplements



## Subtraktion mit $b$ -Komplement

- ▶ bei Rechnung mit fester Stellenzahl  $n$  gilt:

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

weil  $b^n$  gerade nicht mehr in  $n$  Stellen hineinpasst

- ▶ also gilt für die Subtraktion auch:

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

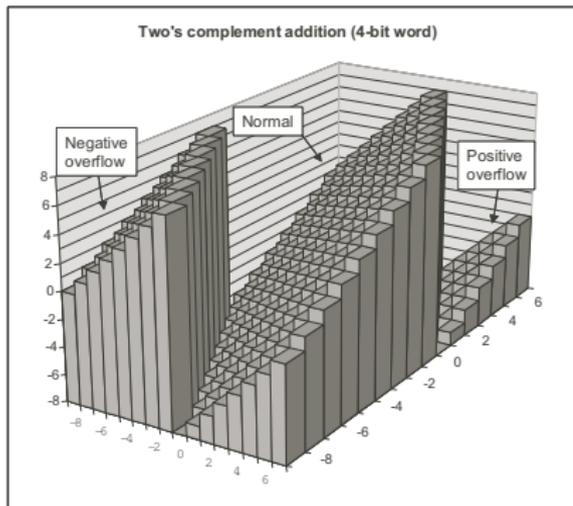
⇒ Subtraktion kann also durch Addition des  $b$ -Komplements ersetzt werden

- ▶ und für Integerzahlen gilt außerdem

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

# signed Addition: Visualisierung

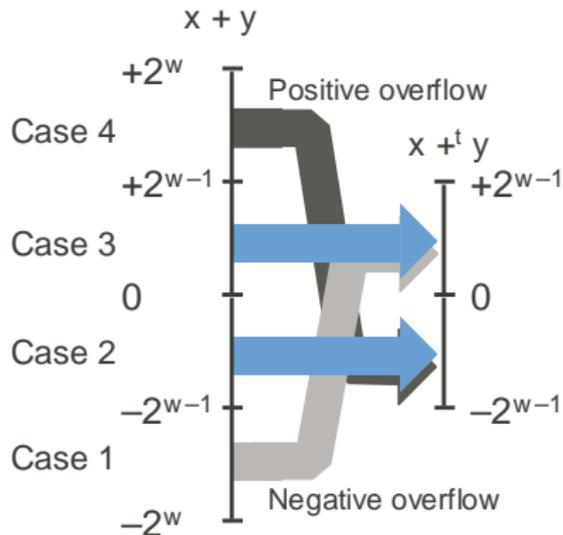
# 2-Komplement



[BO14]

- ▶ Wortbreite der Operanden ist  $w$ , hier 4-bit
  - ▶ Zahlenbereich der Operanden  $x, y$  ist  $-2^{w-1} .. (2^{w-1} - 1)$
  - ▶ Zahlenbereich des Resultats  $s$  ist  $-2^w .. (2^w - 2)$
- ⇒ Überlauf in beide Richtungen möglich

## signed Addition: Überlauf



- ▶ Wortbreite des Resultats ist  $w$ : Bereich  $-2^{w-1} .. (2^{w-1} - 1)$
- ▶ 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

```

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



## Subtraktion mit Einer- und Zweierkomplement (cont.)

- ▶ das  **$b$ -Komplement** einer Zahl  $z$  ist

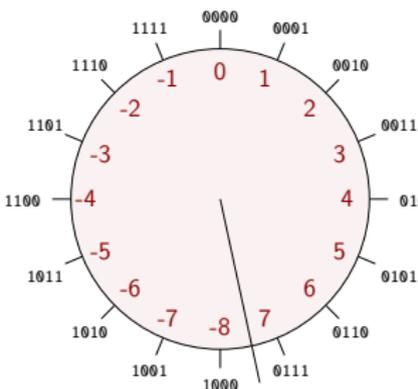
$$\begin{aligned}
 K_b(z) &= b^n - z, & \text{für } z \neq 0 \\
 &= 0, & \text{für } z = 0
 \end{aligned}$$

- ▶ das  **$(b - 1)$ -Komplement** einer Zahl  $z$  ist

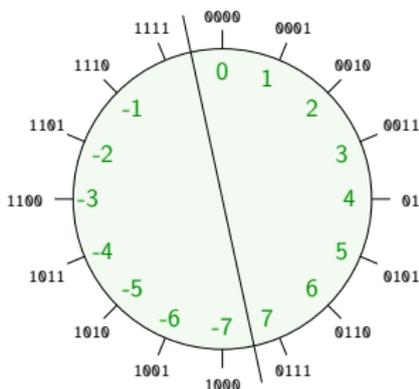
$$\begin{aligned}
 K_{b-1}(z) &= b^n - b^{-m} - z, & \text{für } z \neq 0 \\
 &= 0, & \text{für } z = 0
 \end{aligned}$$

# Veranschaulichung: Zahlenkreis

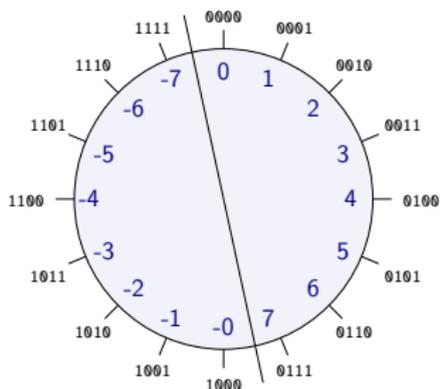
Beispiel für 4-bit Zahlen



2-Komplement



1-Komplement

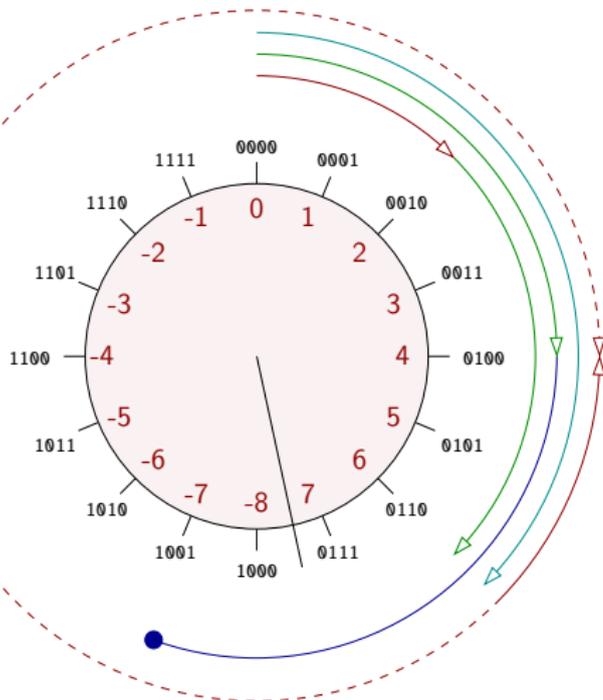


Betrag+Vorzeichen

- Komplement-Arithmetik als Winkeladdition

# Zahlenkreis: Addition, Subtraktion

2-Kompl.



$$0010 + 0100 = 0110$$

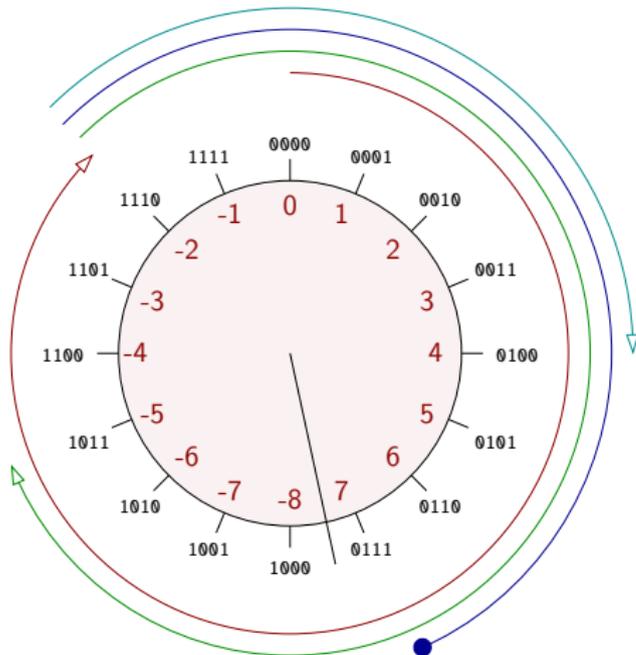
$$0100 + 0101 = 1001$$

$$0110 - 0010 = 0100$$

0010	1110
0100	
0101	
0110	

# Zahlenkreis: Addition, Subtraktion (cont.)

2-Kompl.



$$1110 + 1101 = 1011$$

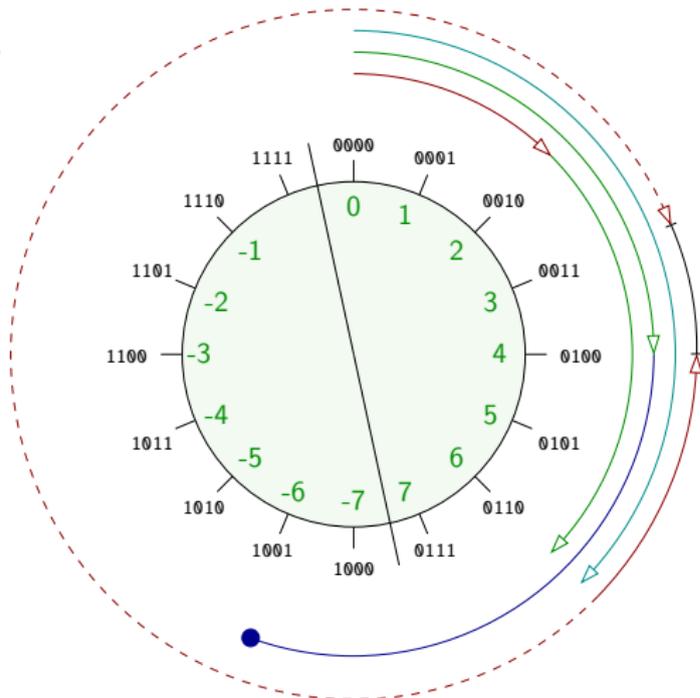
$$1110 + 1001 = \mathbf{0111}$$

$$1110 + 0110 = 0100$$

1110  
 1101  
 1001  
 0110

# Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.



$$0010 + 0100 = 0110$$

$$0100 + 0101 = 1001$$

$$0110 + 1101 + 1 = 0100$$

$$0010 \quad 1101$$

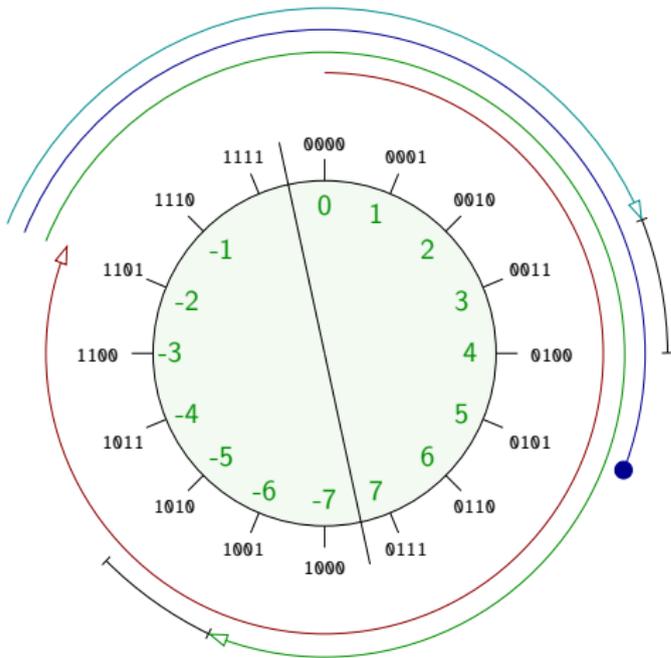
$$0100$$

$$0101$$

$$0110$$

# Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.



$$1101 + 1100 + 1 = 1010$$

$$1101 + 1000 = \mathbf{0101}$$

$$1101 + 0110 + 1 = \mathbf{0100}$$

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

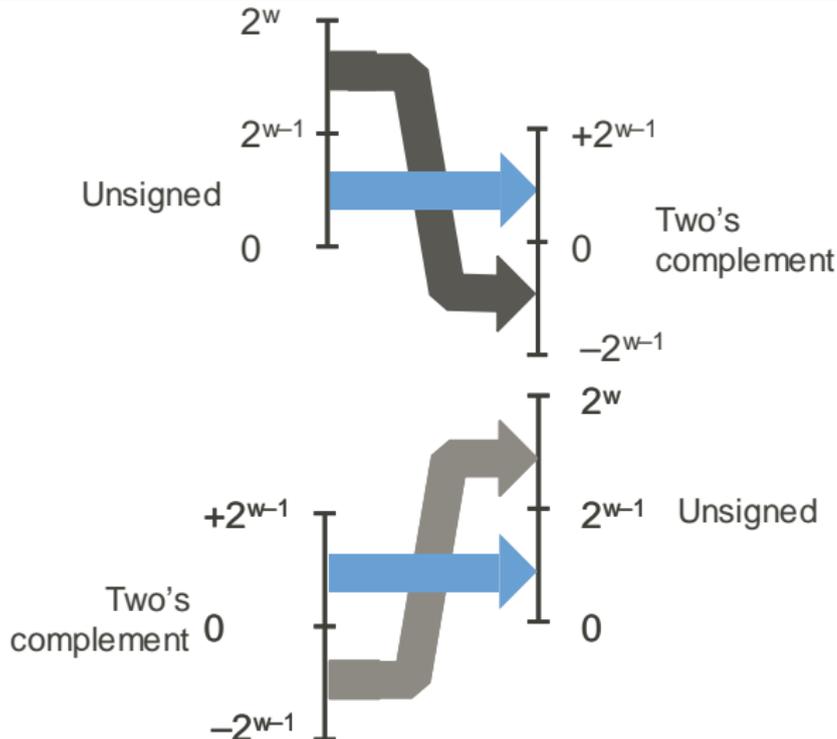
```

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
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

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

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?

0xffffffffcafebabe	(sign-extension!)
0x0000000100000000	
Ü 11111110	
0000000cafebabe	

# Ariane-5 Absturz





## Ariane-5 Absturz (cont.)

- ▶ Erstflug der Ariane-5 („V88“) am 04. Juni 1996
- ▶ Kurskorrektur wegen vermeintlich falscher Fluglage
- ▶ Selbstzerstörung der Rakete nach 36,7 Sekunden
- ▶ Schaden ca. 370 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
  
- ▶ [http://de.wikipedia.org/wiki/Ariane\\_V88](http://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 ist sehr einfach:

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

## Multiplikation im Dualsystem (cont.)

► Beispiel

$$\begin{array}{r}
 10110011 \times 1101 \\
 \hline
 10110011 \quad 1 \\
 10110011 \quad 1 \\
 00000000 \quad 0 \\
 10110011 \quad 1 \\
 \hline
 \text{Ü } 11101111 \\
 \hline
 100100010111
 \end{array}
 \quad = 179 \cdot 13 = 2327$$

$$\begin{array}{l}
 = 1001\ 0001\ 0111 \\
 = 0x917
 \end{array}$$



## unsigned Multiplikation

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



## signed Multiplikation

## 2-Komplement

- ▶ Zahlenbereich der Operanden:  $-2^{w-1} .. (2^{w-1} - 1)$
  - ▶ Zahlenbereich des Resultats:  $-2^w .. (2^{2w-2})$
- ⇒ 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
- ⇒ kein separater Algorithmus erforderlich
- Beweis: siehe Bryant, O'Hallaron: Abschnitt 2.3.5 [BO14]



## 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 mit 32-bit berechnet, dabei Überlauf
- ▶ Konvertierung nach 64-bit long erst bei 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

$$100_{10} / 3_{10} = 1100100_2 / 11_2 = 100001_2$$

$$\begin{array}{r}
 1100100 \ / \ 11 = 0100001 \\
 \begin{array}{r}
 1 \qquad \qquad \qquad 0 \\
 11 \qquad \qquad \qquad 1 \\
 -11 \\
 \hline
 0 \qquad \qquad \qquad 0 \\
 0 \qquad \qquad \qquad 0 \\
 1 \qquad \qquad \qquad 0 \\
 10 \qquad \qquad \qquad 0 \\
 100 \qquad \qquad \qquad 1 \\
 -11 \\
 \hline
 1 \qquad \qquad \qquad 1 \quad (\text{Rest})
 \end{array}
 \end{array}$$

## Division im Dualsystem (cont.)

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

$$\begin{array}{r}
 1011011 \ / \ 1101 = 0111 \\
 1011 \qquad \qquad \qquad 0 \\
 10110 \qquad \qquad \qquad 1 \\
 \underline{-1101} \\
 10011 \qquad \qquad \qquad 1 \\
 \underline{-1101} \\
 01101 \qquad \qquad \qquad 1 \\
 \underline{-1101} \\
 0
 \end{array}$$

## Höhere mathematische Funktionen

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

- ▶ Approximation über Polynom (Taylor-Reihe) bzw. Approximation 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  $x = 3$ ,  $y_0 = 0,5$ :

$$\begin{aligned} y_1 &= 0,5 \cdot (2 - 3 \cdot 0,5) &&= 0,25 \\ y_2 &= 0,25 \cdot (2 - 3 \cdot 0,25) &&= 0,3125 \\ y_3 &= 0,3125 \cdot (2 - 3 \cdot 0,3125) &&= 0,33203125 \\ y_4 &= 0,3332824 \\ y_5 &= 0,333333332557231 \\ y_6 &= 0,3333333333333333 \end{aligned}$$



# 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  $y_0$

## Vorsicht mit Rundungsfehlern/Auslöschung

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

```

public class Exponential {                // 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));
    }
}
    
```



## Vorsicht mit Rundungsfehlern/Auslöschung

```

/* Naively compute the e^x using the Taylor series:
 * e^x = 1 + x + x^2/2! + x^3/3! + ...
 * and also using Math.exp. Catastrophic cancelation
 * occurs if x is a large negative number. */
% java Exponential 1
2.7182818284590455
2.7182818284590455
% java Exponential 50
5.184705528587075E21
5.184705528587072E21
% java Exponential -1
0.36787944117144245
0.36787944117144233
% java Exponential -50
11072.93338289197
1.9287498479639178E-22
  
```



## Informationstreue

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

$$1.0\text{E}20 + (-1.0\text{E}20 + 3.14) = 0$$



# 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  $a \oplus_w^u \bar{a} = 0; \bar{a} = 2^w - a$



## Festkomma Addition (cont.)

signed Arithmetik

2-Komplement

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

$$a \oplus_w^s b = U2S(S2U(a) \oplus_w^u S2U(b))$$

⇒ isomorphe Algebra zu  $\oplus_w^u$

- ▶ kommutative Gruppe / Abel'sche Gruppe

- ▶ Abgeschlossenheit  $-2^{w-1} \leq a \oplus_w^s b \leq 2^{w-1} - 1$

- ▶ Kommutativgesetz  $a \oplus_w^s b = b \oplus_w^s a$

- ▶ Assoziativgesetz  $a \oplus_w^s (b \oplus_w^s c) = (a \oplus_w^s b) \oplus_w^s c$

- ▶ neutrales Element  $a \oplus_w^s 0 = a$

- ▶ Inverses  $a \oplus_w^s \bar{a} = 0; \quad \bar{a} = -a, \quad a \neq -2^{w-1}$   
 $a, \quad a = -2^{w-1}$



# Festkomma Multiplikation

## unsigned Arithmetik

- ▶ Wortbreite auf  $w$  begrenzt
- ▶ Modulo-Arithmetik  $a \otimes_w^u b = (a \cdot b) \bmod 2^w$
- ▶  $\otimes_w^u$  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-Kompl.
- ▶ isomorph zum Ring der ganzen Zahlen *modulo*  $2^w$
  
- ▶ Ordnungsrelation im Ring der ganzen Zahlen
  - ▶  $a > 0 \quad \longrightarrow \quad a + b > b$
  - ▶  $a > 0, b > 0 \longrightarrow a \cdot b > 0$
  - ▶ 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?<br>(aber Infinity oder NaN möglich)   | Ja   |
| ▶ Kommutativ?  | Ja   |
| ▶ Assoziativ?<br>(Überlauf, Rundungsfehler)  | Nein |
| ▶ Eins ist neutrales Element?  | Ja   |
| ▶ Distributivgesetz?   | Nein |
| ▶ Monotonie? $a \geq b; c \geq 0 \rightarrow (a \cdot c) \geq (b \cdot c)$<br>(außer für NaN und Infinity) | Fast |



## Literatur

- [BO14] R.E. Bryant, D.R. O'Hallaron:  
*Computer systems – A programmers perspective.*  
 2nd new intl. ed., Pearson Education Ltd., 2014.  
 ISBN 978-1-292-02584-1. [csapp.cs.cmu.edu](http://csapp.cs.cmu.edu)
- [TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.*  
 6. Auflage, Pearson Deutschland GmbH, 2014.  
 ISBN 978-3-86894-238-5



## Literatur (cont.)

- [Omo94] A.R. Omondi: *Computer Arithmetic Systems – Algorithms, Architecture and Implementations*. Prentice-Hall International, 1994. ISBN 0–13–334301–4
- [Kor01] I. Koren: *Computer Arithmetic Algorithms*. 2nd edition, CRC Press, 2001. ISBN 978–1–568–81160–4.  
[www.ecs.umass.edu/ece/koren/arith](http://www.ecs.umass.edu/ece/koren/arith)
- [Spa76] O. Spaniol: *Arithmetik in Rechenanlagen*. B. G. Teubner, 1976. ISBN 3–519–02332–6