

64-040 Modul IP7: Rechnerstrukturen

3. Arithmetik

Norman Hendrich & Jianwei Zhang

Universität Hamburg
MIN Fakultät, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
{hendrich,zhang}@informatik.uni-hamburg.de

WS 2010/2011

Inhalt

Arithmetik

Addition und Subtraktion

Multiplikation

Division

Höhere Funktionen

Informationstreue



Rechner-Arithmetik

- ▶ Wiederholung: Stellenwertsystem
- ▶ Addition: Ganzzahlen, Zweierkomplementzahlen
- ▶ Überlauf

- ▶ Multiplikation
- ▶ Division

- ▶ Schiebe-Operationen



Wiederholung: Stellenwertsystem

- ▶ Wahl einer geeigneten Zahlenbasis b („Radix“)
 - ▶ 10: Dezimalsystem
 - ▶ 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
 - etc.
- ▶ Werte sind für die Sprache fest definiert

Addition im Dualsystem

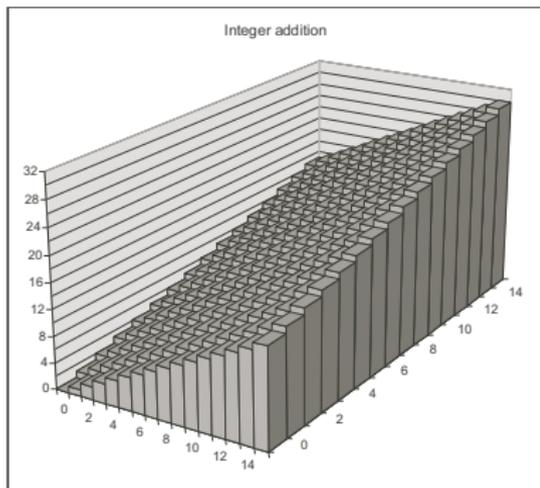
- ▶ Addition mehrstelliger Zahlen erfolgt stellenweise
- ▶ Additionsmatrix:

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

- ▶ Beispiel:

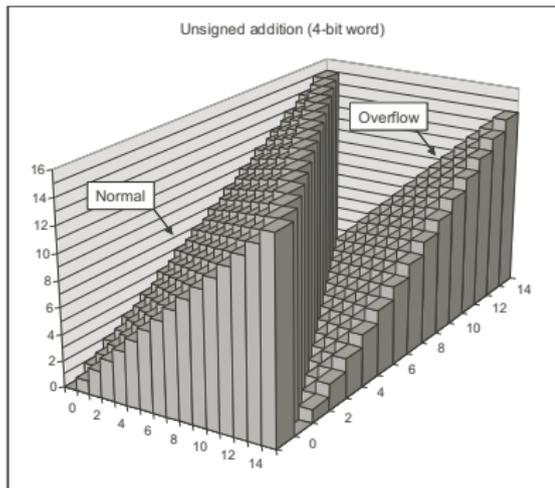
$$\begin{array}{r}
 1011 \ 0011 \\
 + 0011 \ 1001 \\
 \hline
 \ddot{U} \ 11 \ 11 \\
 \hline
 1110 \ 1100
 \end{array}$$

Visualisierung: 4-bit Addition



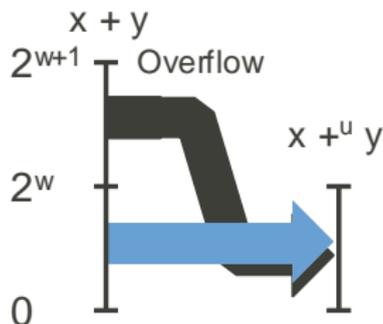
- ▶ Wortbreite w , hier 4-bit
- ▶ Zahlenbereich der Operanden x, y ist $0 \dots (2^w - 1)$
- ▶ Zahlenbereich des Resultats s ist $0 \dots (2^{w+1} - 2)$

Addition: 4-bit unsigned



- ▶ Operanden und Resultat jeweils 4-bit
- ▶ Überlauf, sobald das Resultat größer als $(2^w - 1)$
- ▶ oberstes Bit geht verloren

Überlauf: 4-bit unsigned



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

$$\begin{array}{r}
 1011\ 0011 \\
 - 0011\ 1001 \\
 \hline
 \ddot{U}\ 1111 \\
 \hline
 111\ 1010
 \end{array}$$

$$179 - 57 = 122$$

- ▶ 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 (!)

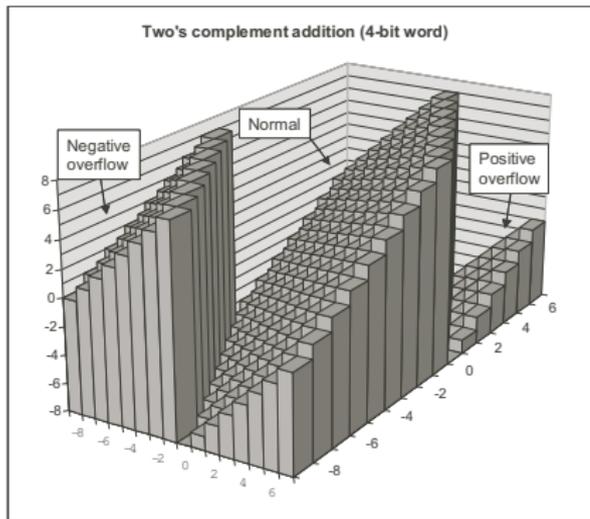
- ▶ also gilt für die Subtraktion auch:

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

- ▶ Subtraktion kann also durch Addition des b -Komplements ersetzt werden
- ▶ und für Integerzahlen gilt außerdem

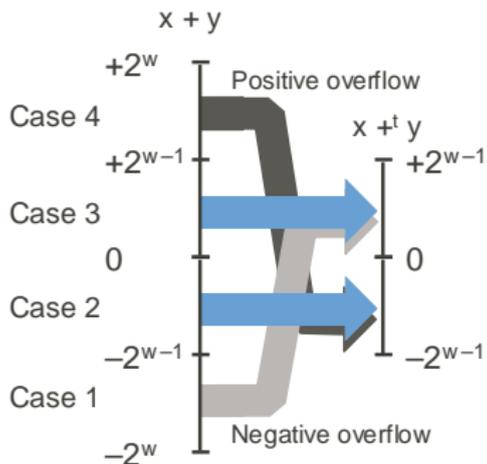
$$x - y = x + K_{b-1}(y) + 1$$

Visualisierung: 4-bit Zweierkomplement-Addition



- ▶ Zahlenbereich der Operanden ist $-2^{w-1} .. (2^{w-1} - 1)$
- ▶ Zahlenbereich des Resultats ist $-2^w .. (2^w - 2)$
- ▶ Überlauf in beide Richtungen möglich

Überlauf: 4-bit signed-Addition



- ▶ Zahlenbereich der Operanden ist $-2^{w-1} .. (2^{w-1} - 1)$
- ▶ Zahlenbereich des Resultats ist $-2^w .. (2^w - 2)$
- ▶ Überlauf in beide Richtungen möglich



Überlauf: Erkennung

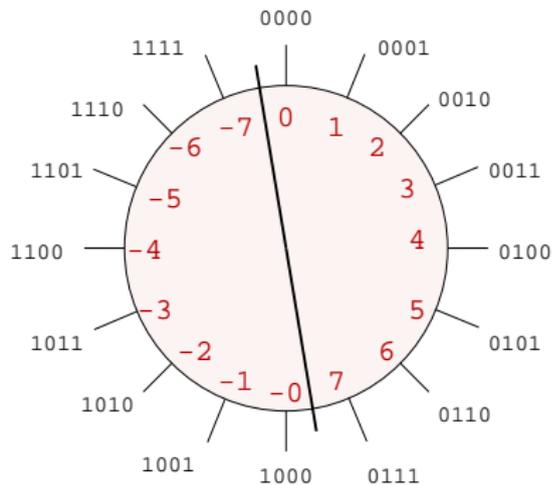
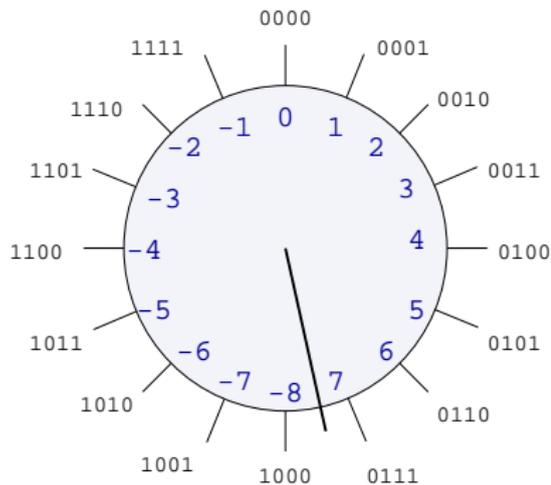
- ▶ Erkennung eines Überlaufs bei der Addition?
- ▶ wenn beide Operanden das gleiche Vorzeichen haben
- ▶ und Vorzeichen des Resultats sich unterscheidet

- ▶ Java-Codebeispiel:

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

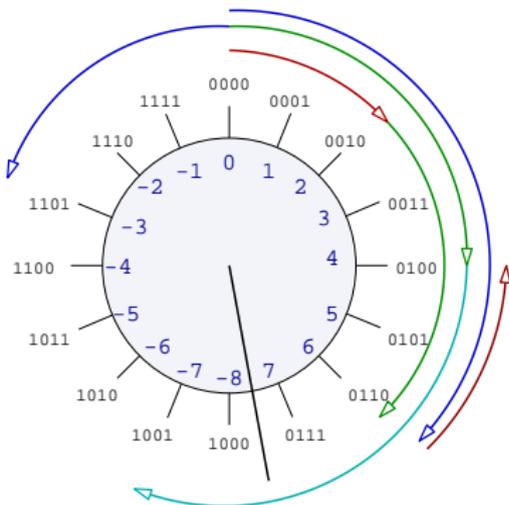
sum = a + b;
ovf = ((a < 0) == (b < 0)) && ((a < 0) != (sum < 0));
```

Visualisierung: Zahlenkreis



► links: Zweierkomplement rechts: Betrag und Vorzeichen

Zahlenkreis: Addition, Subtraktion



$$0010 + 0100 = 0110, \quad 0100 + 0101 = 1001, \quad 0110 - 0010 = 0100$$



Unsigned-Zahlen in C

- ▶ 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];
}
    
```



Unsigned-Typen in C: Casting-Regeln

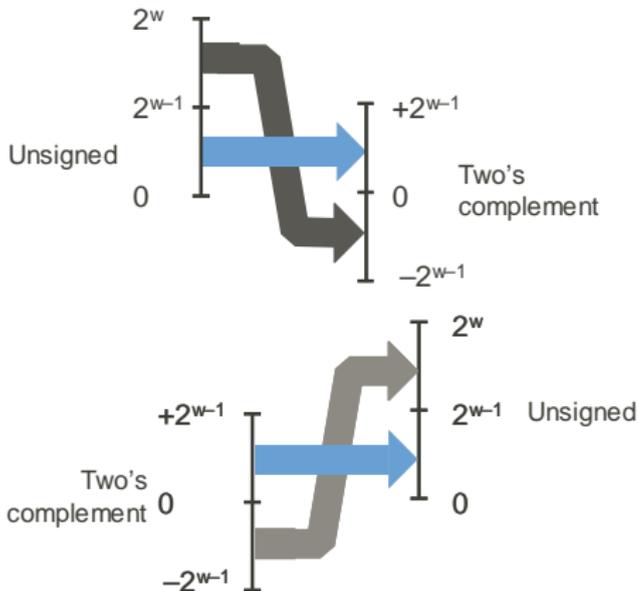
- ▶ Bit-Repräsentation wird nicht verändert
- ▶ kein Effekt auf positiven Zahlen
- ▶ Negative Werte als (große) positive Werte interpretiert:

```
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

Interpretation: unsigned/signed



Typumwandlung in C: Vorsicht

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

Konstante1	Relation	Konstante2	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

```
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



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. 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)

Multiplikation: 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:

$$\begin{array}{r|ll}
 * & 0 & 1 \\
 \hline
 0 & 0 & 0 \\
 1 & 0 & 1
 \end{array}$$

Multiplikation: Beispiel

$$10110011 * 1101 = 179 * 13 = 2327 = 0x917$$

```

-----
10110011    1
 10110011    1
 00000000    0
 10110011    1
-----
Ü 11 1111
-----
100100010111
    
```



Multiplikation: Wertebereich unsigned

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

Multiplikation: Zweierkomplement

- ▶ Zahlenbereich der Operanden ist $-2^{w-1} .. (2^{w-1} - 1)$
- ▶ Zahlenbereich des Resultats ist $-2^w \cdot (2^{w-1} - 1) .. (2^{2w-2})$
- ▶ bis zu $2w$ bits erforderlich

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


```
int i = 100*200*300*400; // -1894967296
```
- ▶ Wichtig: Bit-Repräsentation der unteren Bits des Resultats entspricht der unsigned Multiplikation
- ▶ kein separater Algorithmus erforderlich
- ▶ Beweis: siehe Bryant/O'Hallaron, 2.3.5



Java Puzzlers No. 3

```
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

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

Division: 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: Beispiel im Dualsystem

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

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

Division: Beispiel im Dualsystem

$$91_{10}/13_{10} = 1011011_2/1101_2 = 111_2$$

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

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
 - ▶ Bsp. 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 $x = 3$, $y_0 = 0.5$:

$$y_1 = 0.5 \cdot (2 - 3 \cdot 0.5) = 0.25$$

$$y_2 = 0.25 \cdot (2 - 3 \cdot 0.25) = 0.3125$$

$$y_3 = 0.3125 \cdot (2 - 3 \cdot 0.3125) = 0.33203125$$

$$y_4 = 0.3332824$$

$$y_5 = 0.3333333332557231$$

$$y_6 = 0.3333333333333333$$

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

Eigenschaften: Rechnerarithmetik

- ▶ Addition ist kommutative Gruppe
- ▶ Abgeschlossenheit der Multiplikation:
 $0 \leq \text{UMult}_v(u, v) \leq 2^{w-1}$
- ▶ Multiplikation ist kommutativ: $\text{UMult}_v(u, v) = \text{UMult}_v(v, u)$
- ▶ Multiplikation ist assoziativ:
 $\text{UMult}_v(t, \text{UMult}_v(u, v)) = \text{UMult}_v(\text{UMult}_v(t, u), v)$
- ▶ Eins ist neutrales Element: $\text{UMult}_v(u, 1) = u$
- ▶ Distributivgesetz:
 $\text{UMult}_v(t, \text{UAdd}_v(u, v)) = \text{UAdd}_v(\text{UMult}_v(t, u), \text{UMult}_v(t, v))$



Eigenschaften: Rechnerarithmetik

Isomorphe Algebren

- ▶ Unsigned Addition und Multiplikation (Wortbreite w Bits)
- ▶ Zweierkomplement-Addition und Multiplikation (w Bits)
- ▶ Isomorph zum Ring der ganzen Zahlen modulo 2^w

- ▶ Ring der ganzen Zahlen: Ordnungsrelationen
 - ▶ $u > 0 \longrightarrow u + v > v$
 - ▶ $u > 0, v > 0 \longrightarrow u \cdot v > 0$
 - ▶ diese Relationen gelten nicht bei Rechnerarithmetik (Überlauf)

Eigenschaften: Gleitkomma-Addition

- | | |
|----------------------------------------------------------------------------------------------|------|
| ▶ Abgeschlossen (Addition)? | Ja |
| ▶ Kommutativ? | Ja |
| ▶ Assoziativ?
(Überlauf, Rundungsfehler) | Nein |
| ▶ Null ist neutrales Element? | Ja |
| ▶ Inverses Element existiert?
(außer für NaN und Infinity) | Fast |
| ▶ Monotonie? $(a \geq b) \rightarrow (a + c) \geq (b + c)$?
(außer für NaN und Infinity) | Fast |

Eigenschaften: Gleitkomma-Multiplikation

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

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