Zahlendarstellung und Rechnerarithmetik

Einstein-Prof. Dr.-Ing. Friedel Gerfers

Ziele

Nach dieser Vorlesung sollten Sie in der Lage sein:

- Binär-/Oktal-/Hexadezimalzahlen zu Dezimalzahlen zu konvertieren und umgekehrt
- 2-Komplement-Zahlen zu berechnen und zu negieren
- m-Bit 2-Komplement-Zahlen zu n-Bit zu konvertieren
- Arithmetische Operationen mit Binärzahlen durchzuführen
- Überlauf (*overflow*) zu erklären und zu erläutern, wann er auftritt und wie er behandelt wird
- Bruchzahlen in den „IEEE 754“ Floating-Point-Standard zu konvertieren und umgekehrt
- Arithmetische Operationen mit „IEEE 754“-Zahlen durchzuführen

Inhalt

- Zahlensysteme und ihre Konvertierung
- Vorzeichenbehaftete (*signed*) und vorzeichenlose (*unsigned*) Zahlen
- 2-Komplemente
- Arithmetische Operationen
  - Addition und Subtraktion
  - Multiplikation
  - Division
- Gleitkommazahlen (Floating Point)
- „Pentium Bug“

Zahlenbasen

- Natürliche Zahlen können in jeder Basis repräsentiert werden:

$$a_{n-1}a_{n-2}\dots a_1 a_{0(Basis\ B)} = a_{n-1}B^{n-1} + \cdots + a_1 B^1 + a_0 B^0 = \sum_{i=0}^{n-1} a_i B^i$$

- B: Basis (engl.: *radix*), z.B. 10 (dezimal), 2 (binär), 8 (oktal), 16 (hexadezimal)
- B^i: Gewicht (engl.: *weight*) der i-ten Ziffer (engl.: *digit*)
- a_i: i-te Ziffer aus der Menge {0, 1, …, B-1}
- Beispiel (binär):

$$\underset{a_3\,a_2\,a_1\,a_0}{1011_B} = \underset{a_3}{1}\text{x}\underset{B^3=8}{2^3} + \underset{a_2}{0}\text{x}\underset{B^2=4}{2^2} + \underset{a_1}{1}\text{x}\underset{B^1=2}{2^1} + \underset{a_0}{1}\text{x}\underset{B^0=1}{2^0} = (8+0+2+1)_D = 11_D$$

Binärzahlen (Dualzahlen)

- Menschen benutzen Dezimalzahlen

  $2435 = 2{\cdot}10^3 + 4{\cdot}10^2 + 3{\cdot}10^1 + 5{\cdot}10^0$

- Rechner benutzen Binärzahlen

  $1011_B = 1{\cdot}2^3 + 0{\cdot}2^2 + 1{\cdot}2^1 + 1{\cdot}2^0 = 11_D$

- Hintergrund: Zwei einfach darstellbar
  - ein / aus
  - hohes / niedriges Potenzial (VDD / GND)
  - schwarz / weiß

- Allgemein gilt:

$$\boxed{\mathbf{b_{n-1}b_{n-2}...b_1b_0 = b_{n-1}{\cdot}2^{n-1} + b_{n-2}{\cdot}2^{n-2} + ... + b_1{\cdot}2^1 + b_0{\cdot}2^0}}$$

Konvertierung

- Teile $b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \ldots + b_1 \cdot 2^1 + b_0 \cdot 2^0$ durch 2
- Das Ergebnis ist $b_{n-1} \cdot 2^{n-2} + b_{n-2} \cdot 2^{n-3} + \ldots + b_1 \cdot 2^0$ mit Rest b_0
- Dies ergibt folgenden Algorithmus für die Konvertierung von Dezimal nach dual / binär:

$167_D \rightarrow$	$167 / 2 = 83$	Rest 1
	$83 / 2 = 41$	Rest 1
	$41 / 2 = 20$	Rest 1
	$20 / 2 = 10$	Rest 0
	$10 / 2 = 5$	Rest 0
	$5 / 2 = 2$	Rest 1
	$2 / 2 = 1$	Rest 0
	$1 / 2 = 0$	Rest 1

Niederwertigstes Bit *(least significant bit* (LSB))

Höchstwertigstes Bit *(most significant bit (MSB))*

$167_D = 10100111_B$

Bits und Bytes (1)

- Alle Informationen werden im Rechner als Binärzahlen gespeichert.
- Kleinste Einheit: 1 Bit = eine Ziffer einer Dualzahl
- Beispiel:

höchstwertigstes (*most significant*) Bit

1Byte = 8bit = 1001 1010

niederwertigstes (*least significant*) Bit

- Mit einer n-Bit Zahl lassen sich die Werte von 0 … (2^n-1) darstellen:
  - Beispiel für n = 3:
  - $111_2 = 1*2^2 + 1*2^1 + 1*2^0 = 1*4 + 1*2 + 1*1 = 7 = 8-1 = 2^3-1$

Bits und Bytes (2)

- Vorsicht bei Präfixen:
  - In der Regel gilt: 1MB = 1 Megabyte = 10^6B = 1 000 000 B
  - In der Informatik: 1MB = 1 Megabyte = 2^{20} B = 1 048 576 B

=> Nicht immer eindeutig definiert!

Weitere Zahlensysteme

- **Problem:** Binärzahlen werden schnell sehr lang
- **Lösung:** Verwendung von weiteren Zahlensystemen mit höheren Basen, in die leicht konvertiert werden kann:

  – Oktale Zahlen (Basis 8)
  – Hexadezimale Zahlen (Basis 16)

Oktale Zahlen (Basis 8) Hexadezimale Zahlen (Basis 16)

Konvertierung ins Oktale (1)

- Oktalzahlen werden zur Basis 8 (= 2^3) gebildet
- Dezimal nach oktal:

$$167_D \text{ -> } \begin{array}{rl} 167 / 8 = 20 & \text{Rest } 7 \\ 20 / 8 = 2 & \text{Rest } 4 \\ 2 / 8 = 0 & \text{Rest } 2 \end{array} \Bigg\} \quad 167_D = 247_O$$

Konvertierung ins Oktale (2)

- Binär nach oktal:

  es werden 3 Binärzeichen benötigt, um die Zahlen 0…7 darzustellen:

$$O_0 * 8^0 = b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0$$

$$111000001011_2 = \underbrace{111}_{7}\ \overbrace{000}^{0}\ \underbrace{001}_{1}\ \overbrace{011}^{3}{}_2 = 7013_8$$

Hexadezimalzahlen (1)

- Hexadezimalzahlen werden zur Basis 16 (= 2^4) gebildet
- Ziffernmenge: { 0, 1, 2, … , 8, 9, A, B, C, D, E, F}
- Dezimal nach hexadezimal:

167_D -> 167 / 16 = 10		Rest 7
	10 / 16 = 0	Rest A

$167_D = A7_H$

In C/Java: 0xa7

Hexadezimalzahlen (2)

- Binär nach hexadezimal:

$$\underbrace{0001}_{1}\ \overbrace{0011}^{3}\ \underbrace{0101}_{5}\ \overbrace{0111}^{7}\ \underbrace{1001}_{9}\ \overbrace{1011}^{B}\ \underbrace{1101}_{D}\ \overbrace{1111}^{F}{}_2 = 13579BDF_H$$

Negative Zahlen

Negative Zahlen (1)

- Möglichkeiten negative Zahlen binär zu repräsentieren:
  - **Vorzeichen-/Betrags-Zahlen** (*Sign-magnitude numbers*)
    - MSB zeigt Vorzeichen (sign) an (0: positiv, 1: negativ).
    - Die übrigbleibenden (n-1) Bits bilden den Betrag (magnitude).
    - Beispiel: 5 (dezimal) = 0101 (binär) → -5 (dezimal) = 1101 (binär)
  - **1-Komplement-Zahlen** (*One's complement numbers*)
    - Zahl wird durch die Invertierung aller Bits negiert.
    - MSB *impliziert* das Vorzeichen.
    - Beispiel: 5 (dezimal) = 0101 (binär) → -5 (dezimal) = 1010 (binär)

Negative Zahlen (2)

- Möglichkeiten negative Zahlen binär zu repräsentieren:
  - **2-Komplement-Zahlen** (*Two's complement numbers*)

    MSB hat ein negatives Gewicht (-2^{n-1}).

    $b_{n-1}b_{n-2}\dots b_1 b_0$ (binär) = $-b_{n-1}2^{n-1}+b_{n-2}2^{n-2}+\dots+b_1 2^1+b_0 2^0$ (dezimal)

    MSB *impliziert* das Vorzeichen.

    Beispiel: -5 (dezimal) = -8 + 3 = 1000+0011=1011 (binär)

Beispiel (4-Bit)

Dezimal	Vorzeichen-/Betrags-Zahlen (Sign-Magnitude)	1-Komplement-Zahlen (One’s complement)	2-Komplement-Zahlen (Two’s complement)
-0	1000	1111	
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8			1000

Und der Gewinner ist ...

- **2-Komplement-Zahlen**
  - Arithmetik ist einfacher (identisch zu vorzeichenlos).
  - Es gibt nur eine Möglichkeit die 0 zu repräsentieren.
- Beispiel 32bit:

  $b_{31}b_{30}\ldots b_1b_0$ (binär) = $-b_{31}2^{31}+b_{30}2^{30}+\ldots+b_12^1+b_02^0$ (dezimal)

- Was ist die <u>kleinste</u> mit 8bit repräsentierbare Integerzahl?
  - $2^7 = -128$
- Was ist die <u>größte</u> mit 8bit repräsentierbare Integerzahl?
  - $2^6 + 2^5 + ... + 2^0 = 127$

Negation

- Negation von 2-Komplement-Zahlen:
  Invertiere alle Bits und addiere 1

- 8-Bit-Beispiel:
  $0110\ 1001_B = +105_D$

invertieren:	1001 0110	
1 addieren:	1001 0111	= -128 + 16 + 7 = -105

Rückwärts:

invertieren:	0110 1000
1 addieren:	0110 1001

Addition / Subtraktion

Binäre Addition

- Addition von rechts nach links mit Übertrag (*carry*) wie in der Grundschule
- Beispiele (4-Bit, 2er complement):

carry

1	
0010	2
+0011	+ 3
0**1**01	5

1	
0101	5
+0110	+ 6
1011	- 5

??? Überlauf (*overflow*)

Rechenregeln:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = ^{1}0

carry

Binäre Subtraktion

- Beispiele (4-Bit, 2er complement):

	`0010`	2
	`-0011`	- 3
	`111`	
borrows	`1111`	- 1

Rechenregeln:
- 0 - 0 = 0
- 0 - 1 = $_1$1 (*borrow*)
- 1 - 0 = 1
- 1 - 1 = 0

1) Beginn mit der Subtraktion an der 1. Stelle (ganz rechts). `0 - 1` ist `1` mit `1` als Übertrag.
2) Durch Übertrag sind an der 2. Stelle `1 - 1 - 1` zu berechnen. `1 - 1` ergibt `0`. Dadurch bleibt `0 - 1` übrig. Das ergibt wieder `1` mit `1` als Übertrag.
3) An der 3. Stelle sind wieder `0 - 0 - 1` zu berechnen. `0 - 1` ergibt wieder eine `1` mit `1` als Übertrag.
4) An der 4. Stelle sind wieder `0 - 0 - 1` zu berechnen. `0 - 1` ergibt wieder eine `1` mit `1` als Übertrag
5) Das Ergebnis der Subtraktion: `0010 - 0011 = 1111`

Binäre Subtraktion

- Beispiele (4-Bit, 2er complement):

```
            0010        2
           -0011      - 3
            111
           -----
borrows     1111      - 1

            1101      - 3
           -0110      - 6
            11
           -----
            0111        7
```

wieder Überlauf

Rechenregeln:

- 0 - 0 = 0
- 0 - 1 = ${}_{1}1$ (*borrow*)
- 1 - 0 = 1
- 1 - 1 = 0

- Alternative:
  Den 2. Operanden negieren und dann beide addieren.
  – Wir werden diesen Trick beim Erstellen unserer Arithmetic Logic Unit (ALU) nutzen.

Subtraktion durch Addition der Negation

- Beispiel (4-Bit, 2er complement):

```
  0010        2
- 0011      - 3
```

- Negiere `0011`
  - Invertieren: `1100`
  - 1 addieren: `1101`

- Addieren

```
  0010        2
+ 1101      - 3

  1111       -1
```

Überlauf / 1 (1)

- Überlauf (***overflow***):
  - Das Ergebnis ist zu groß für ein endliches Computer-Wort
  - Z. B. Addition von zwei n-Bit-Zahlen muss keine n-Bit-Zahl ergeben

- Kein Überlauf, wenn …
  - Addition von 2 Zahlen mit entgegengesetztem Vorzeichen
    - Betrag der Summe immer $\leq$ den Beträgen der beiden Operanden
    - Beispiel: -10 + 6 = - 4
  - Subtraktion von 2 Zahlen mit demselben Vorzeichen

Überlauf / 1 (2)

- Overflow tritt auf, wenn…

Operation	**A**	**B**	**Overflow wenn (Vorzeichen) des Ergebnisses**
A+B	$\geq$ 0	$\geq$ 0	< 0 (MSB = 1)
A+B	< 0	< 0	$\geq$ 0 (MSB = 0)
A-B	$\geq$ 0	< 0	< 0 (MSB = 1)
A-B	< 0	$\geq$ 0	$\geq$ 0 (MSB = 0)

Überlauf / 2

- Vergleich der Operationen A + B und A – B
  - Kann ein Überlauf auftreten, wenn B = 0 ist?
  - Kann ein Überlauf auftreten, wenn A = 0 ist?

Nein!

Ja! Falls 0 – kleinste negative Zahl, für 2 bit 0 – (– 2) = 2. Ergebniss nicht darstellbar

- Überlauf tritt nur auf, wenn das Übertragsbit (*carry in*) für das MSB ≠ dem entstehenden Übertragsbit (*carry out*) aus der Operation der beiden MSBs ist.

$$\boxed{\textit{carry in}_{MSB} \neq \textit{carry out}_{MSB}}$$

- Beispiel (4-Bit): carry out → 01 ← carry in

```
  0101
+ 0110
  ----
  1
  1011
```

Multiplikation / Division

Multiplikation

- Für den Moment betrachten wir nur positive Zahlen, d.h. MSB ist kein sign Bit

- Schulmathematik: 13 x 11 = ...

```
Multiplikand       1101                 13
Multiplikator x    1011               x 11
                   1101 (1101x1)
                  1101  (1101x1)
                 0000   (1101x0)
            +   1101    (1101x1)
      Produkt 10001111                 143
```

- Beachte: Produkt erfordert doppelte Stellenanzahl!

Division / 1

- Schulmathematik: 74 / 8 = ...

Dividend		Divisor		Quotient
1001010	/	1000	=	1

```
1000
----
   1
```

Division / 2

- Schulmathematik: 74 / 8 = ...

Dividend		Divisor		Quotient
1001010	/	1000	=	10

```
1001010   /   1000    =    10
1000
   10
```

Division / 3

- Schulmathematik: 74 / 8 = ...

Dividend		Divisor		Quotient
1001010	/	1000	=	100

```
1001010
1000
   101
```

Division / 4

- Schulmathematik: 74 / 8 = ...

Dividend / Divisor = Quotient

```
1001010  /  1000  =  1001
1000                  = 9
   1010
   1000
     10   → Rest
    = 2
```

$$1001010 \,/\, 1000 = \underbrace{1001}_{=9}$$

$$\underbrace{10}_{=2} \rightarrow \text{Rest}$$

→ 74 / 8 = 9 Rest 2

Gebrochene Zahlen / Gleitkommazahlen

Gebrochene Zahlen

- Darstellung gebrochener Zahlen:

$$a_{n-1}a_{n-2}\ldots a_1 a_0 , a_{-1}a_{-2}\ldots a_{-m}$$

$$= a_{n-1}B^{n-1} + \cdots + a_1 B^1 + a_0 B^0 + a_{-1}B^{-1} + a_{-2}B^{-2} + \cdots + a_{-m}B^{-m} =$$

$$= \sum_{i=0}^{n-1} a_i B^i + \sum_{i=-m}^{-1} a_i B^i$$

- Beispiel (binär):

$11{,}1010_B = 3 + 2^{-1} + 2^{-3}$

$= 3 + 0{,}5 + 0{,}125 = 3{,}625_D$

Konvertierung

- Dezimal $(0\ ,\ b_{-1}b_{-2}...b_{-m}\ )_D$ nach Dual
  → $2 \times (\ 0\ ,\ b_{-1}b_{-2}...b_{-m}\ )_D = (\ b_{-1}\ ,\ b_{-2}b_{-3}...b_{-m}\ )$
- Beispiel: $0{,}24_D$

$0{,}24_D$ ->

$$0{,}24 \cdot 2 = 0{,}48 + 0$$

MSB!

$$0{,}48 \cdot 2 = 0{,}96 + 0$$
$$0{,}96 \cdot 2 = 0{,}92 + 1$$
$$0{,}92 \cdot 2 = 0{,}84 + 1$$
$$0{,}84 \cdot 2 = 0{,}68 + 1$$
$$0{,}68 \cdot 2 = 0{,}36 + 1$$
$$0{,}36 \cdot 2 = 0{,}72 + 0$$
$$0{,}72 \cdot 2 = 0{,}44 + 1 \longrightarrow 0{,}00111101_B$$

LSB!

  – Abbruch nach 8 Stellen (Näherung mit 0,238…)

- Ziffernbegrenzung kann zu Rundungsfehlern führen

Gleitkommazahlen (*floating point*) (1)

- Näherung für reelle Zahlen: $\boxed{(-1)^s \times 1.f \times 2^e}$
  - s: Vorzeichen (*sign*): 0 → positiv, 1 → negativ
  - $1.f$: Mantisse (Betrag, *significand*) als normalisierte Zahl
    - Zahl wird so lange geschoben, bis sie führende 1 aufweist
    - Binärpunkt wird rechts von dieser 1 festgelegt ($1.0 \leq 1.f < 2.0$)
  - f: nur der Bruch f (*fraction*) wird gespeichert, führende 1 ist implizit (wird von Recheneinheit ergänzt)

Gleitkommazahlen (*floating point*) (2)

- Näherung für reelle Zahlen: $\boxed{(-1)^s \times 1.f \times 2^e}$
  - *e*: vorzeichenbehafteter Exponent, wird als transformierter Exponent *E* gespeichert
  - *E*: $E = e + \text{bias}$
    - bias wird so gewählt, dass 2-Komplement-Zahl e zur vorzeichenlosen Dualzahl *E* wird
    - bias entweder bei einfacher Genauigkeit 127 oder 1023 bei doppelter Genauigkeit (double)

IEEE-754 Standard

- Einfache Genauigkeit (*single precision*, 32 Bit)

1	8	23 Bit
s	*E*	*f*

bias = 127

C/Java: float

  – E=0 / 255 (dienen als Sonderfälle)

- Doppelte Genauigkeit (*double precision*, 64 Bit)

1	11	52 Bit
s	*E*	*f*

bias = 1023

C/Java: double

Beispiel (I)

- $-0{,}75_D$ mit einfacher Genauigkeit
  - $s = 1$
  - $0{,}75_D$ als gebrochene Dualzahl ist $0{,}11_B$
  - Normalisiere: $0{,}11 = 1{,}1 \times 2^{-1}$
    - führende 1 ist implizit → Bruch = 10000....
  - Transformierter Exponent E

$$E = e + \text{bias} = -1 + 127 = 126 = 0111\ 1110_B$$

1	8	23 Bit
1	0111 1110	100 0000 0000 0000 0000 0000

1	8	23 Bit
s	E	f

Beispiel (II)

- $-0{,}75_D$ mit einfacher Genauigkeit

1	8	23 Bit
1	0111 1110	100 0000 0000 0000 0000 0000

- https://www.h-schmidt.net/FloatConverter/IEEE754de.html

Tools & Thoughts

IEEE-754 Konverter für Fließkommazahlen

Translations: en

Diese Webseite dient zur Umrechnung zwischen der Dezimaldarstellung von Zahlen (z.B. "1.02") und den von modernen CPUs verwendeten IEEE 754-Fließkommazahlen. Aus Platzgründen werden nur Zahlen mit einfacher Genauigkeit (32 Bit) dargestellt. Diese Seite soll dem besseren Verständnis von Fließkommazahlen dienen.

IEEE 754 Converter (JavaScript), V0.21

	Sign	Exponent	Mantissa
Value:	-1	2^{-1}	1.5
Encoded as:	1	126	4194304
Binary:	☑	☐ ☑ ☑ ☑ ☑ ☑ ☑ ☐	☑ ☐

You entered	-0.75
Value actually stored in float:	-0.75
Error due to conversion:	0.00
Binary Representation	10111111010000000000000000000000
Hexadecimal Representation	0xbf400000

+1

-1

Beispiel (III)

- $0{,}075_D$ mit einfacher Genauigkeit
  - $s = 0$
  - $0{,}075_D$ als gebrochene Dualzahl ist $0{,}0001\ 0011\ 0011\ 0011\ 0011..._B$

$0{,}075_D \rightarrow$

$$0{,}075 \cdot 2 = 0{,}150 + 0$$
$$0{,}150 \cdot 2 = 0{,}300 + 0$$
$$0{,}300 \cdot 2 = 0{,}600 + 0$$
$$0{,}600 \cdot 2 = 0{,}200 + 1$$
$$0{,}200 \cdot 2 = 0{,}400 + 0$$
$$0{,}400 \cdot 2 = 0{,}800 + 0$$
$$0{,}800 \cdot 2 = 0{,}600 + 1$$
$$0{,}600 \cdot 2 = 0{,}200 + 1$$

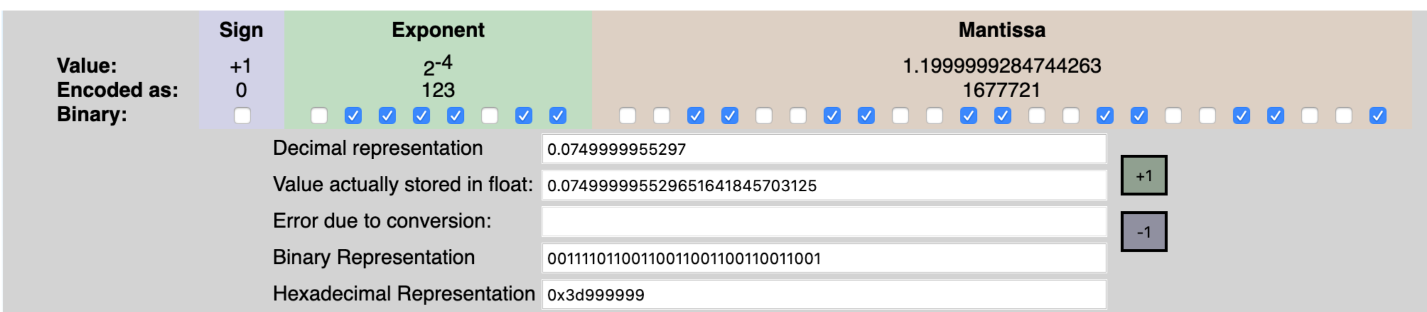
MSB!

Beispiel (III)

- $0{,}075_D$ mit einfacher Genauigkeit
  - $s = 0$
  - $0{,}075_D$ als gebrochene Dualzahl ist $0{,}0001\ 0011\ 0011\ 0011\ 0011\ldots_B$
  - Normalisiere: $0{,}00010011 = 1{,}0011 \times 2^{-4}$
    - führende 1 ist implizit → Bruch = 0011....
  - Transformierter Exponent *E*

$$E = e + \text{bias} = -4 + 127 = 123 = 0111\ 1011_B$$

1	8	23 Bit
0	0111 1011	0011 0011 0011 0011 0011 001



Sonderfälle (1)

- Neben normalisierten Zahlen sind außerhalb des Zahlenraums definiert:
  - ± Null
  - ± Unendlich: z. B. Division durch Null
  - ± Unnormalisierte (*unnormalized*) Zahlen: winzige (*tiny*) Zahlen
  - Nichtzahlen (Not a Number, NaN): Ergebnis ungültiger Operation wie 0/0

Sonderfälle (2)

- Codiert durch den größten und kleinsten Exponentwert E und f:
  - Normal.: $(-1)^s$ x $1.f$ x $2^{E\text{-}127/\text{-}1023}$ $1 \leq E \leq 254/2046$
  - Null: $(-1)^s$ x 0 $E = 0, f = 0$
  - Unendlich: $(-1)^s$ x ∞ $E = 255/2047, f = 0$
  - Unnorm.: $(-1)^s$ x $0.f$ x $2^{-126/-1022}$ $E = 0, f \neq 0$; interpretiert mit $E = 1$
  - Nichtzahl: NaN $E = 255/2047, f \neq 0$

Addition von Gleitkommazahlen

Addition von Gleitkommazahlen (1)

- Beispiel basiert auf 16-Bit Minifloat Format:

1	5	10 Bit
s	*E*	*f*

bias = 15
Exponentenber.: $-14 \leq e \leq 15$

- $Z = X + Y$ mit
  - $X = 2{,}35_D = 10\mathbf{.}0101\ 1001\ 1001\ 1001\ \ldots_B$
  - $Y = 10{,}17_D = 1010\mathbf{.}0010\ 1011\ 1000\ 0101\ \ldots_B$

Addition von Gleitkommazahlen (2)

- **1. Schritt:**
  - **Normalisieren** und **Anpassung** an 16-Bit-Format:
  - $X = 1\mathbf{.}0010\ 1100\ 11 \cdot 2^1$
  - $Y = \mathbf{1.0100}\ 0101\ 01 \cdot 2^3$

    → Verlust von signifikanten Stellen!

$X = 2{,}35_D =$	$10\mathbf{.}0101\ 1001\ 1001\ 1001\ _B$
$Y = 10{,}17_D =$	$1010\mathbf{.}0010\ 1011\ 1000\ 0101\ _B$

- **2. Schritt:**
  - **Vergleichen der beiden Exponenten** *e***.**
  - Bei Ungleichheit kleineren Exponent an den größeren anpassen
  - $X = 0\mathbf{.}0100\ 1011\ 00\ 11 \cdot 2^3$

    → rot dargestellte Stellen gehen verloren

Addition von Gleitkommazahlen (3)

- **3. Schritt:**
  - **Addieren der Mantissen**:

$$
\begin{array}{r}
0\mathbf{.}0100\ 1011\ 00\ (X) \\
\underline{+\ 1\mathbf{.}0100\ 0101\ 01\ (Y)} \\
1\mathbf{.}1001\ 0000\ 01\ (Z)
\end{array}
$$

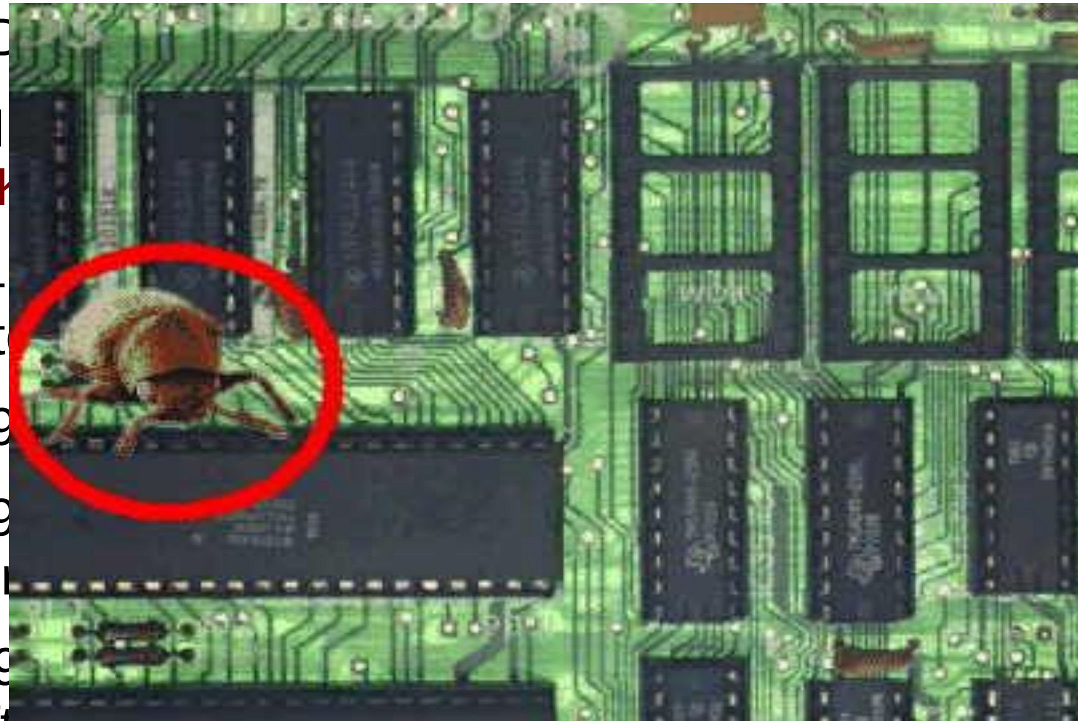
- **Ergebnis**
  - Muss ggf. noch normalisiert werden (hier nicht)
  - $Z = 1\mathbf{.}1001\ 0000\ 01 \cdot 2^3 = 12{,}5078125$ (korrekt wäre: 12,52).

Pentium Bug

- Fehler im Divisionsalgorithmus für Gleitpunktzahlen
- Juli 1994: Intel entdeckt Fehler. Geschätzte Kosten zum Beheben: einige 100K$
- Sept. 1994: Matheprof Thomas Nicely entdeckt Fehler. Erhält keine offizielle Stellungnahme Intel, veröffentlicht Entdeckung im Internet
- 7. Nov. 1994: EE Times bringt Geschichte auf Titelseite
- 22. Nov. 1994: Pressemitteilung Intel: Pentium könne Fehler an 9. Stelle verursachen, nur wenige Benützer könnten betroffen sein
- 5. Dez. 1994: Intel behauptet, Fehler würde nur einmal in 27000 Jahren auftreten bei typischer Anwendung
- 21. Dez. 1994: Intel gibt zu: jeder Besitzer darf Pentium austauschen. Geschätzte Kosten: **500 M$**!

4195835.0/3145727.0 = 1.333 820 449 136 241 002 (korrekter Wert)
4195835.0/3145727.0 = 1.333 739 068 902 037 589 (fehlerhafter Pentium)

Pentium Bug



- Fehler im D
- Juli 1994: I Beheben: einige 100k
- Sept. 1994 rhält keine offizielle St m Internet
- 7. Nov. 199
- 22. Nov. 19 hler an 9. Stelle veru en sein
- 5. Dez. 199 27000 Jahren auftreten bei typischer Anwendung
- 21. Dez. 1994: Intel gibt zu: jeder Besitzer darf Pentium austauschen. Geschätzte Kosten: **500 M$**!

4195835.0/3145727.0 = 1.333 820 449 136 241 002 (korrekter Wert)
4195835.0/3145727.0 = 1.333 739 068 902 037 589 (fehlerhafter Pentium)

Zusammenfassung (1)

- Computer benutzen Binärzahlen statt Dezimalzahlen
  - $b_{n-1}b_{n-2}...b_1b_0 = b_{n-1}\cdot 2^{n-1} + b_{n-2}\cdot 2^{n-2} + ... + b_1\cdot 2^1 + b_0\cdot 2^0$
- Binärzahlen können leicht in oktale und hexadezimale Zahlen konvertiert werden
- 2. Komplement wird benutzt um vorzeichenbehaftete Zahlen darzustellen
  - $b_{n-1}b_{n-2}...b_1b_0 = - b_{n-1}\cdot 2^{n-1} + b_{n-2}\cdot 2^{n-2} + ... + b_1\cdot 2^1 + b_0\cdot 2^0$
- Computer-Arithmetik ist beschränkt durch limitierte Genauigkeit.
  - *Overflow* (Überlauf)
  - *Underflow* (floating point)

Zusammenfassung (2)

- Bitmuster haben keine inhärente Bedeutung, es gibt jedoch Standards
  - 2-Komplement
  - IEEE 754 Floating Point
- Computer-Befehle bestimmen die “Bedeutung” der Bitmuster.
- Leistung und Genauigkeit sind wichtig. Daraus ergeben sich viele Hardwareanforderungen/Komplexitäten für reale Maschinen / CPUs
  - Z. B. Algorithmen und Implementierungen

- Was kommt als Nächstes?
  - Wir werden einen Prozessor implementieren.
  - Dazu brauchen wir Grundlagen **Digitaltechnik**.