GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Vorlesung 4: Rechnerarithmetik

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Literatur:
Slomka [1]
Kap. 2.1–2.4

Licensed under CC BY-SA 4.0
unless noted otherwise

Nochmal: Stellenwertsysteme

Zerlegung einer Zahl in ***Exponentialdarstellung***:

12305 = 1 * 10000 + 2 * 1000 + 3 * 100 + 0 * 10 + 5 * 1

oder auch

"Ziffer mit Wert 2"

"an Stelle 3" (von rechts, wir fangen bei 0 an)

$12305 = 1 * 10^4 + 2 * \mathbf{10}^3 + 3 * 10^2 + 0 * 10^1 + 5 * 10^0$

"Basis 10"

…funktioniert auch mit anderen Basen, hier Basis **2**:

$\mathbf{10110_{(2)} = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0}$

$= 16 + 4 + 2 = 22$

Andere nützliche Basen: Oktal

Idee: Zusammenfassen mehrerer Binärziffern (Bits)

***Oktal**: Basis 8*

$12305_{(8)} = 1 * \mathbf{8^4} + 2 * \mathbf{8^3} + 3 * \mathbf{8^2} + 0 * \mathbf{8^1} + 5 * \mathbf{8^0}$

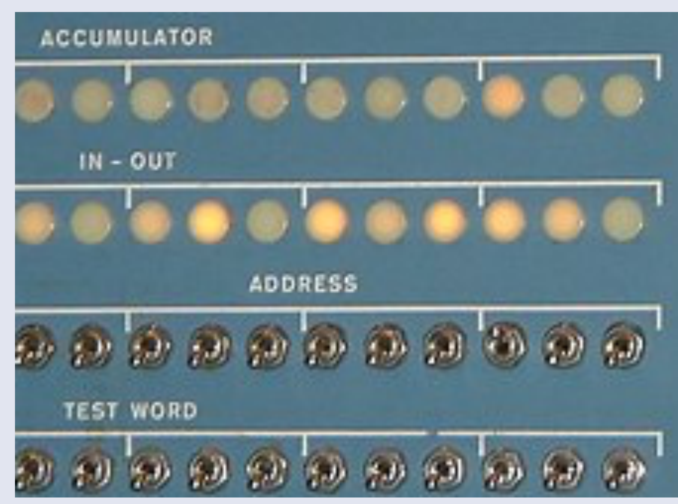
$= 1 * \mathbf{4096} + 2 * \mathbf{512} + 3 * \mathbf{64} + 0 * \mathbf{8} + 5 * \mathbf{1}$

$= 5317_{(10)}$

Warum Basis 8?
Eine Ziffer stellt genau 3 Bits dar!

$12305_{(8)}$

$= 001\ 010\ 011\ 000\ 101_{(2)}$



PDP-1 control board
(CC BY 2.0 DEED by fjarlq / Matt)

Hexadezimal – Basis 16

Gruppen von 3 Bits sind heute unpraktisch…
…Daten werden in Vielfachen von Bytes (8 Bit) gespeichert

***Hexadezimal (auch "hex", "sedezimal")**:*
*Basis 16 = 2^4 also **vier Bits pro Ziffer***

Aber wir haben doch nur 10 Ziffern 0…9?
Die Hexadezimaldarstellung nutzt Buchstaben A…F:
A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

$$
\begin{aligned}
A31B_{(16)} &= 10 * 16^3 + 3 * 16^2 + 1 * 16^1 + 11 * 16^0 \\
&= 10 * 4096 + 3 * 256 + 0 * 16 + 11 * 1 \\
&= 41755_{(10)}
\end{aligned}
$$

Schreibweise auch
0xA31B (z.B. in C)
$A31B (in Assembler)

Allgemein: *b-adische* Darstellung

Theorem:

Für jede Basis $b \in \mathbb{N} \setminus \{1\}$ und jede natürliche Zahl $n \in \mathbb{N}$ lässt sich n eindeutig darstellen als:

$$n = n_0 \cdot b^0 + n_1 \cdot b^1 + \cdots + n_\ell \cdot b^\ell = \sum_{i=0}^{\ell} n_i \cdot b^i$$

mit

- $l \in \mathbb{N}_0$
- $n_0, n_1, \ldots, n_\ell \in \{0,1,...,b-1\}$
- $n_\ell > 0$

Darstellung ganzer Zahlen

Bisher haben wir **positive** Zahlen als Binärzahlen dargestellt:

$$\mathbf{10110_{(2)} = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0}$$

Wie können wir auch **negative Zahlen** darstellen?

Annahme:
feste Anzahl Bits pro Wort: **Länge** ℓ

Binär	Pos. Zahl
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Darstellung ganzer Zahlen: Vorzeichenbit

Universität Bamberg

Wie können wir auch **negative Zahlen** darstellen?

Vorzeichenbetragsmethode:
erstes Bit: Vorzeichen, restliche Bits: Betrag

Kleinste darstellbare Zahl: $-(2^{\ell-1} - 1)$
Größte darstellbare Zahl: $2^{\ell-1} - 1$

Eigenschaften:
+ symmetrisch
+ Vorzeichenwechsel einfach
– 0 nicht eindeutig
– Vergleich von Zahlen schwierig

Binär	Posit. Zahl	VZ-Betrag
000	0	+0
001	1	+1
010	2	+2
011	3	+3
100	4	-0
101	5	-1
110	6	-2
111	7	-3

Bias-Darstellung ("Exzess"):

"Verschieben" des Wertebereichs mit fester Verschiebung b (**Bias**) –
Zahl z wird als $z + b$ dargestellt

Übliche Werte für Bias:
$b = 2^{\ell-1}$ oder $b = 2^{\ell-1} - 1$

Kleinste darstellbare Zahl: $-b$
Größte darstellbare Zahl: $2^{\ell} - b - 1$

Eigenschaften:

+ 0 eindeutig, alle Codes ausgenutzt
+ Vergleich von Zahlen einfach
+ bei üblichem Bias: 1. Bit = Vorzeichen
– nicht symmetrisch
– Vorzeichenwechsel schwierig

Binär	Posit. Zahl	Bias
000	0	-4
001	1	-3
010	2	-2
011	3	-1
100	4	+0
101	5	+1
110	6	+2
111	7	+3

Ganze Zahlen: Einerkomplement

Einerkomplement:
nicht-negative Zahlen nutzen reguläre Binärdarstellung
negative Zahlen sind **Komplement** der Binärdarstellung

Kleinste darstellbare Zahl: $-(2^{\ell-1}-1)$
Größte darstellbare Zahl: $2^{\ell-1}-1$

Eigenschaften:
+ symmetrisch
+ erstes Bit wie Vorzeichenbit
– 0 nicht eindeutig
– "Verschwendung" eines Codes (-0)
– Vergleich von Zahlen schwierig

Binär	Posit. Zahl	1er-Kompl.
000	0	+0
001	1	+1
010	2	+2
011	3	+3
100	4	-3
101	5	-2
110	6	-1
111	7	-0

Ganze Zahlen: Zweierkomplement

Universität Bamberg

Zweierkomplement:
nicht-negative Zahlen nutzen reguläre Binärdarstellung
negative Zahlen sind (Komplement der Binärdarstellung) **+ 1**

Kleinste darstellbare Zahl: $-(2^{\ell-1})$
Größte darstellbare Zahl: $2^{\ell-1} - 1$

Binär	Posit. Zahl	2er-Kompl.
000	0	+0
001	1	+1
010	2	+2
011	3	+3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Eigenschaften:

+ *0 eindeutig repräsentiert* *
+ erstes Bit wie Vorzeichenbit
+ alle Codes ausgenutzt
– Vergleich von Zahlen schwierig

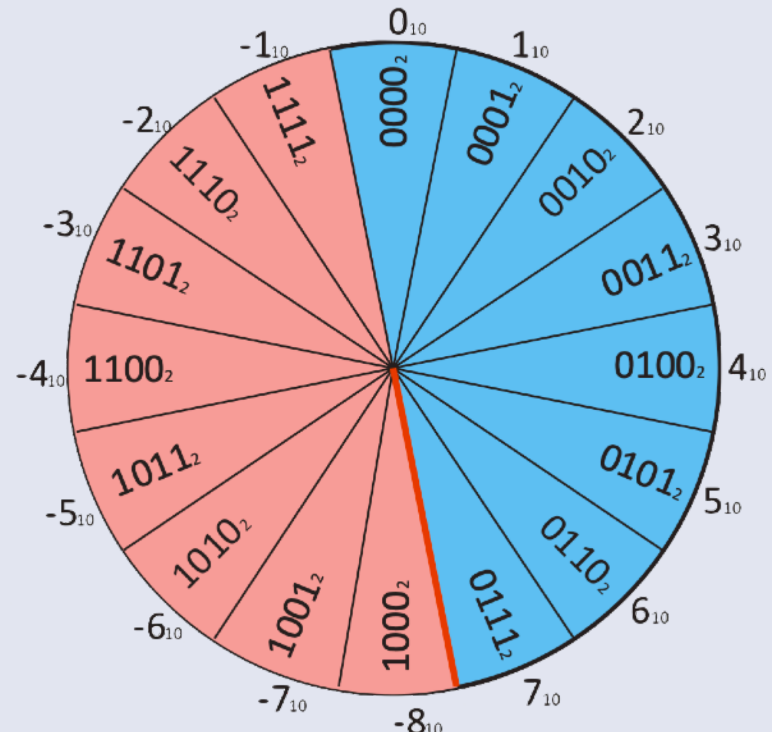
*** hier stand ursprünglich "symmetrisch", was falsch ist…**

Ganze Zahlen: Zweierkomplement

Zweierkomplement:

Kleinste darstellbare Zahl: $-(2^{\ell-1})$

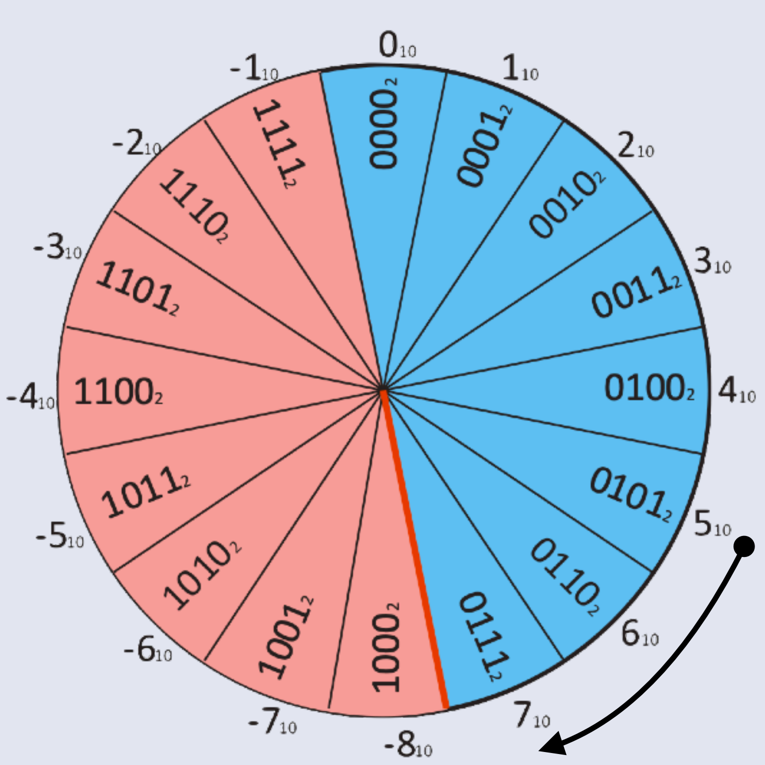
Größte darstellbare Zahl: $2^{\ell-1} - 1$



Binär	Posit. Zahl	2er-Kompl.
000	0	+0
001	1	+1
010	2	+2
011	3	+3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Rechnen mit Zweierkomplement

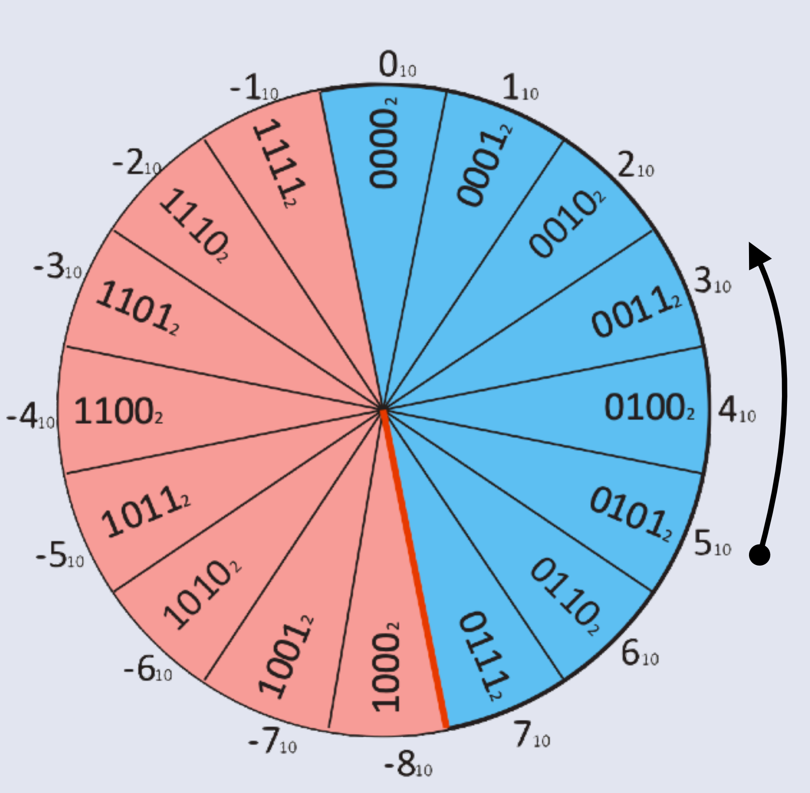
Keine Fallunterscheidung bei Addition und Subtraktion



5	0	1	0	1
+2	0	0	1	0
7	0	1	1	1

Rechnen mit Zweierkomplement

Keine Fallunterscheidung bei Addition und Subtraktion

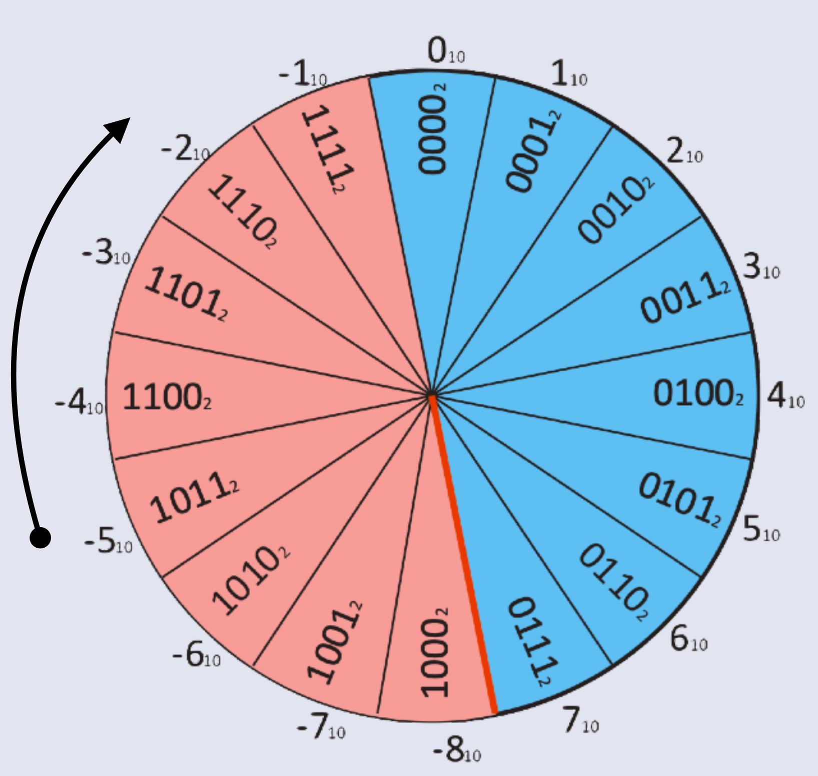


5		0	1	0	1
+(–2)		1	1	1	0
3	**1**	0	0	1	1

Übertragsbit ignorieren!

Rechnen mit Zweierkomplement

Keine Fallunterscheidung bei Addition und Subtraktion



-5		1	0	1	1
+3		0	0	1	1
–2	**0**	1	1	1	0

Übertragsbit ignorieren!

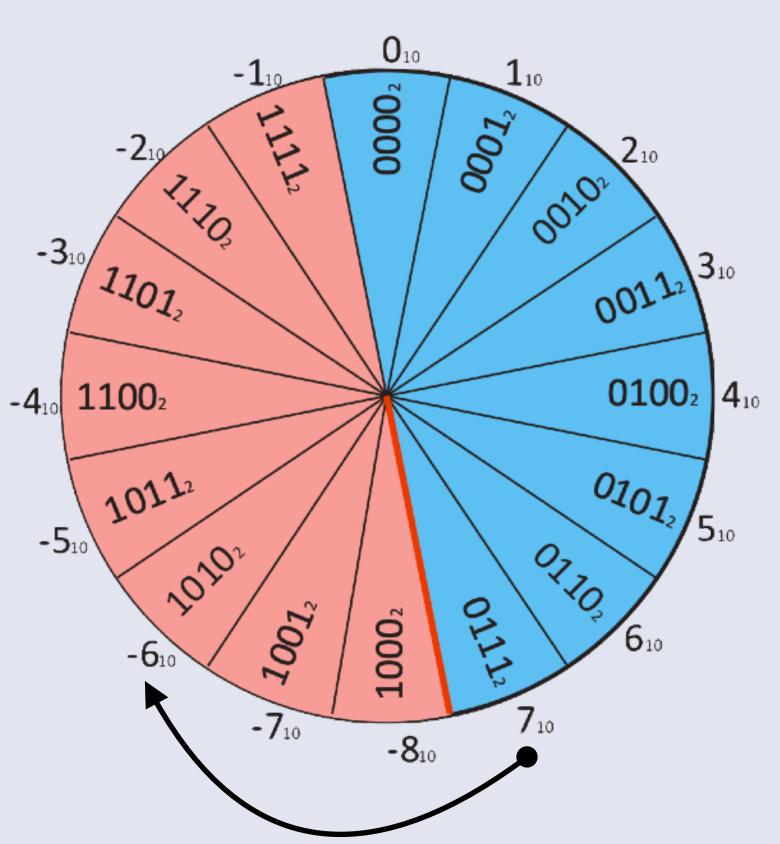
Überblick Ganzzahldarstellungen

Feste Länge $\ell = 5, \quad 2^\ell = 32, \quad (2^{\ell-1} = 16, \quad 2^{\ell-1} - 1 = 15)$

z	VZ-Betrag	Bias b=16	Bias b=15	1er-Kompl.	2er-Kompl.
1	00001	10001	10000	00001	00001
–1	10001	01111	01110	11110	11111
0	00000 10000	10000	01111	00000 11111	00000
15	01111	11111	11110	01111	01111
–15	11111	00001	00000	10000	10001
16	-	-	11111	-	-
–16	-	00000	-	-	10000

Zweierkomplement: Überlauf

Achtung – was passiert hier? **Überlauf (*overflow*)!**



7		0	1	1	1
+3		0	0	1	1
-6	**0**	**1**	**0**	**1**	**0**

Übertragsbit ignorieren!

Größte darstellbare Zahl:

$$2^{\ell-1} - 1 = 2^3 - 1 = 7$$

Unterschied zwischen Mathematik und Computerarithmetik!

Wertebereich

Computer verarbeiten nur eine ***endliche Anzahl von Bits*** gleichzeitig

- Erste Mikroprozessoren (Intel 4004, 4040): 4 Bit
- 70er/80er-Jahre Mikroprozessoren und kleine Mikrocontroller (z.B. AVR auf Arduino): 8 Bit
- 90er-Jahre PCs und Unix-Systeme: 32 Bit
- Heute in PCs, Handys, Servern: 64 Bit

Wir bezeichnen Mengen zusammengehöriger Bits als…
- 4 Bit: Nybble
- 8 Bit: **Byte**
- 32 oder 64 Bit: **Wort** (nicht standardisiert)

Wertebereich von C-Variablen

In C: wie viele Bits benötigt eine Variable vom Typ
(unsigned) char, short, int, long, long long?

⇒ Abhängig von Prozessor, Compiler / Betriebssystem!

CPU	Comp. +OS	char	short	int	long	long long
AVR	**gcc –**	8	16	16	32	32
x86-32	**gcc Linux**	8	16	32	32	64
x86-64	**gcc Linux**	8	16	32	64	64
x86-64	**VC Win**	8	16	32	**32**	64

- sizeof() in C gibt Größe von Datentyp in Bytes
- stdint.h stellt Datentypen mit definierter Größe bereit

Potenzprobleme?

Universität Bamberg

Warum hat eine SSD mit "250 GB" nur
250 * 10^9 = 250.000.000.000 Bytes und nicht
250 * 2^{30} = 268.435.456.000 Bytes?

Hauptspeicher (RAM) nutzt Zweierpotenzen!

Heute ist die Basiseinheit für Speicher das Byte
Mehr als ein Byte: SI-Einheiten vs. Binärprefixe

Dezimalpräfixe gemäß SI		
Kilobyte	kB	1 000 = 10^3
Megabyte	MB	1 000 000 = 10^6
Gigabyte	GB	1 000 000 000 = 10^9
Terabyte	TB	1 000 000 000 000 = 10^{12}
Petabyte	PB	1 000 000 000 000 000 = 10^{15}
Exabyte	EB	1 000 000 000 000 000 000 = 10^{18}
Zettabyte	ZB	1 000 000 000 000 000 000 000 = 10^{21}
Yottabyte	YB	1 000 000 000 000 000 000 000 000 = 10^{24}
Ronnabyte	RB	1 000 000 000 000 000 000 000 000 000 = 10^{27}

Dezimalpräfixe gemäß SI		
Kibibyte	kiB	1 024 = 2^{10}
Mebibyte	MiB	1 048 576 = 2^{20}
Gibibyte	GiB	1 073 741 824 = 2^{30}
Tebibyte	TiB	1 099 511 627 776 = 2^{40}
Pebibyte	PiB	1 125 899 906 842 624 = 2^{50}
Exbibyte	EiB	1 152 921 504 606 846 976 = 2^{60}
Zebibyte	ZiB	1 180 591 620 717 411 303 424 = 2^{70}
Yobibyte	YiB	1 208 925 819 614 629 174 706 176 = 2^{80}
Robibyte	RiB	1 237 940 039 285 380 274 899 124 224 = 2^{90}

Was kommt nach dem Komma?

Binärdarstellung von Ganzzahlen:

$\mathbf{01010110_{(2)} = 1 * 2^6 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0}$

Wie können wir auch **Nachkommastellen** darstellen?

Idee: Verschieben der Wertigkeit von Bits

Beispiel: 3 Bits Nachkommastellen:

$\mathbf{00010{,}110_{(2)} = 1 * 2^1 + 0 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3}}$

$\mathbf{= 1 * 2 + 0 * 1 + 1 * {}^1/_2 + 1 * {}^1/_4 + 0 * {}^1/_8 = 2^3/_4}$

"Binärkomma"

Die Welt nach dem Komma…

Bisher haben wir **Ganzzahlen** betrachtet
Festkommazahlen: Verschieben des "Binärkommas"

$1010[,0000\ldots]_{(2)} = 10$
⇒ ***Binäres Komma*** *3 Stellen nach links verschieben:*

$1{,}0100000_{(2)} = 1{,}25$

Nachkommastellen:
$2^{-1} = {}^{1}/_{2} = 0{,}5 = 0{,}1_{(2)}$
$2^{-2} = {}^{1}/_{4} = 0{,}25 = 0{,}01_{(2)}$
$2^{-3} = {}^{1}/_{8} = 0{,}125 = 0{,}001_{(2)}$
$2^{-4} = {}^{1}/_{16} = 0{,}0625 = 0{,}0001_{(2)}$
$2^{-5} = {}^{1}/_{32} = 0{,}03125 = 0{,}00001_{(2)}$
$2^{-6} = {}^{1}/_{64} = 0{,}015625 = 0{,}000001_{(2)}$
$2^{-7} = {}^{1}/_{128} = 0{,}0078125 = 0{,}0000001_{(2)}$
$2^{-8} = {}^{1}/_{256} = 0{,}00390625 = 0{,}00000001_{(2)}$

… und so weiter …

Genaue Darstellung von Zahlen

Problem ist von Dezimalzahlen bekannt:

$^1/_3 = 0{,}3333333333333333333333333333333333333333333$

Bei Binärzahlen sind auch "glatte" (dezimale) Nachkommastellen nicht immer mit einer endlichen Anzahl von Ziffern darstellbar!

Beispiel: 0,1 als Binärzahl mit 8 Bits nach dem Komma:

$$0{,}1 \cong {}^1/_{16} + {}^1/_{32} + {}^1/_{256} = 0{,}0625 + 0{,}003125 + 0{,}00390625$$
$$= 0{,}09765625 = 0{,}00011001_{(2)}$$

0,1 ist mit 8 Nachkommabits nicht präzise darstellbar!

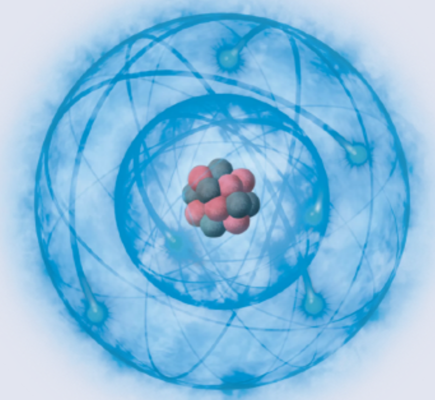
Bessere Nachkommadarstellung?

Problem von Festkommazahlen:
Eingeschränkter Wertebereich
(z.B. 32 Bit-Zahlen: 0…~4,2 Milliarden)

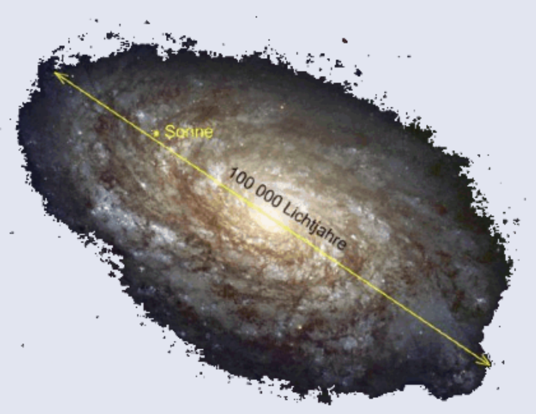
Frage: Wie können wir

- sehr kleine *und*
- sehr große

Zahlen darstellen,
ohne viele Nullen zu speichern?



Atomradius:
ca. 0,0000000001 m



Durchmesser der Milchstraße:
ca. 946073000000000000000 m

Erweiterung des Wertebereiches

Verwendung der **wissenschaftlichen Notation:**

3 850 000 = 3,850 000, * 10^6 = 3,85 * 10^6

Komma sechs Stellen nach links verschieben

0,000 000 32 = 0,000 000 3,2 * 10^{-7} = 3,2 * 10^{-7}

Komma sieben Stellen nach rechts verschieben

Wichtig: *stets eine Vorkommastelle (> 0) in wiss. Notation*

…das ganze in binär!

Wissenschaftliche Notation in Binärzahlen:
Gleitkommazahlen

$$1536 = 1024 + 512$$
$$= 100\ 0000\ 0000_{(2)} + 10\ 0000\ 0000_{(2)}$$
$$= 110\ 0000\ 0000_{(2)}$$

Binärkomma zehn Stellen nach links verschieben:

$$110\ 0000\ 0000_{(2)} = 1{,}10\ 0000\ 0000_{(2)} * 2^{10} = 1{,}1_{(2)} * 2^{10}$$

…das ganze in binär!

Standard: **IEEE 754-Gleitkommazahlen**

Gleitkommazahl z: z = (-1)$^{\text{Vorzeichen}}$ * Mantisse * 2$^{\text{Exponent}}$

mit $s \in \{0,1\}$, $e \in \mathbb{Z}$, $m \in \mathbb{Q}$ und $1 \leq m < 2$

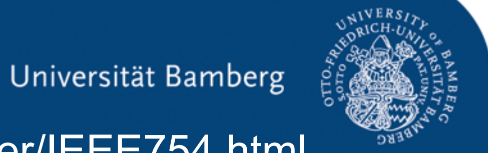
Vorzeichen: positiv (0, da $-1^0=+1$) / negativ (1, da $-1^1 = -1$)

Exponent: z.B. –126 (=$0000\ 0001_{(2)}$)…+127 (=$1111\ 1111_{(2)}$)

Mantisse: Binäre Nachkommastellen der Zahl

Einfache Präzision: 1 Bit VZ, 8 Bit Exponent, 23 Bit Mantisse
Doppelte Präzision: 1 Bit VZ, 11 Bit Exponent, 52 Bit Mantisse

…das ganze in binär!



https://www.h-schmidt.net/FloatConverter/IEEE754.html

Gleitkommazahl $z = (-1)^{\text{Vorzeichen}} * \text{Mantisse} * 2^{\text{Exponent}}$

Beispiel: $1536 = 1024 + 512 = 110\ 0000\ 0000_{(2)}$

Stets nur eine Vorkommastelle > 0 in wiss. Notation:

$110\ 0000\ 0000_{(2)} = 1{,}10\ 0000\ 0000_{(2)} = +\ 1{,}1 * 2^{10}$

Als IEEE 754-Gleitkommazahl mit einfacher Präzision:

	Sign	Exponent	Mantissa
Value:	+1	2^{10}	1.5
Encoded as:	0	137	4194304
Binary:	0	10001001	10000000000000000000000

You entered	1536
Value actually stored in float:	1536
Error due to conversion:	0
Binary Representation	01000100110000000000000000000000

+1 -1

Achtung, Trick!

Stets nur eine Vorkommastelle > 0 in wiss. Notation:

$110\ 0000\ 0000_{(2)} = 1{,}10\ 0000\ 0000_{(2)} = +\ 1{,}1 * 2^{10}$

⇒ Die **Vorkommastelle** bei binären Gleitkommazahlen **ist immer = 1**!

Also müssen wir sie nicht abspeichern: **1 Bit gespart!**

	Sign	Exponent	Mantissa
Value:	+1	2^{10}	1.5
Encoded as:	0	137	4194304
Binary:	☐	☑ ☐ ☐ ☐ ☑ ☐ ☐ ☑	☑ ☐

You entered	1536
Value actually stored in float:	1536
Error due to conversion:	0
Binary Representation	01000100110000000000000000000000

+1

-1

Seltsame Effekte…

Universität Bamberg

0,1 + 0,2 ≠ 0,3?

Die "1"-Vorkommastelle nicht vergessen!

$0{,}1 \cong +\ 1{,}\ 1001\ 1001\ 1001\ 1001\ 1001\ 101_{(2)} * 2^{-4}$
$= 0{,}100000001490116119384765625$

$0{,}2 \cong +\ 1{,}\ 1001\ 1001\ 1001\ 1001\ 1001\ 101_{(2)} * 2^{-3}$
$= 0{,}20000000298023223876953125$

Addieren:

```
 0,00011001,10011001100110011101(2)
+0,0001100,110011001100110011010(2)
 0,0010110011001100110011001....(2)
```

Also nochmal

0,1 + 0,2 ≠ 0,3

```
 0,00001001,10011001100110011001101(2)
+0,0001100,110011001100110011001010(2)
 0,0010110011001100110011001....(2)

= 1,0011001100110011001100111(2)* 2^-2
```

Nur 23 Bit Mantisse!

Kein Platz mehr für diese beiden Bits ⇒ Präzisionsverlust!

…was ist denn jetzt?

Universität Bamberg

0,1 + 0,2 ≠ 0,3

1,00110011001100110011001 11 * 2^{-2}

Nur 23 Bit Mantisse!

Kein Platz mehr für diese beiden Bits ⇒ Präzisionsverlust!

Unser Computer sieht die Summe:

$1{,}00110011001100110011001_{(2)} * 2^{-2}$

$= (1+2^{-3}+2^{-4}+2^{-7}+2^{-8}+2^{-11}+2^{-12}+2^{-15}+2^{-16}+2^{-19}+2^{-20}+2^{-23})*2^{-2}$

= 0.2999999821186065673828125

…was ist denn jetzt?

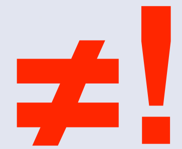
0,1 + 0,2 ≠ 0,3!

Unser Computer sieht die Summe 0,1 + 0,2 wie folgt:

$1,00110011001100110011001_{(2)} * 2^{-2}$

$= (1+2^{-3}+2^{-4}+2^{-7}+2^{-8}+2^{-11}+2^{-12}+2^{-15}+2^{-16}+2^{-19}+2^{-20}+2^{-23})*2^{-2}$

$= 0,2999999821186065673828125$



Unser Computer stellt 0,3 aber dar als:

$1,00110011001100110011010_{(2)} * 2^{-2}$

$= 0,300000011920928955078125$

Spaß mit der Bank…

Zinsen auf Bankguthaben (als es noch welche gab…) sind ein Prozentsatz des Guthabens, z.B. 3% 😂

Beispiel

Guthaben: 1000 €
Zinsen: 1000 € * 3% = 1000 € * 0,03 = 33,333333333… €

Wer erhält die Bruchteile von Cents (0,003333333… €)?

Ein schlauer Bankprogrammierer in den 1970er Jahren ließ alle diese Bruchteile auf sein eigenes Konto überweisen…
und wurde fast zehn Jahre lang nicht erwischt!

Erster Flug der Ariane 5-Rakete

Erster Flug der Ariane 5-Rakete am 4. Juni 1996

Abweichung von Flugbahn nach 40 Sekunden in 3,7 km Höhe und Explosion der Rakete als Folge eines kompletten Verlusts von Positions- und Fluglagedaten

Ursache: Spezifikations- und Entwurfsfehler in der Software des Trägheitskontrollsystems

Ariane 5-Rakete

Fehler, der im Trägheitskontrollsystem auftrat:

Ausführung der Umwandung eines Datenwerts
… von 64 Bit Gleitkommazahl
… zu einer 16 Bit (vorzeichenbehafteten) Ganzzahl

Der Wert der Gleitkommazahl war größer als der maximale Wert, der in einer 16 Bit Ganzzahl gespeichert werden kann (32767)!

Spezielle Werte in IEEE754

Gleitkommazahl z **= (-1)$^{\text{Vorzeichen}}$ * Mantisse * 2$^{\text{Exponent}}$**

Aber:
Die Vorkommastelle der Mantisse bei Gleitkommazahlen ist immer = 1!

Wie können wir die Zahl 0 darstellen?

Besondere Darstellung:
Exponent = 0, Mantisse = 0
Vorzeichen = 0 oder 1 (+0 oder -0)

Zeichendarstellung

Computer können beliebige Daten verarbeiten…
wenn sie als Bits dargestellt werden

⇒ Wie werden Texte repräsentiert?

Naiver Ansatz:
Codierung 'A' = 0, 'B' = 1, … und dann Binärcodierung

Probleme:
- Welche Zeichen sollen codiert werden?
- Wie kann man Daten mit anderen austauschen?

Lösung: *Standards*

ASCII-Standard

Universität Bamberg

	000	001	010	011	100	101	110	111
0000...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
0001...	BS	HT	LF	VT	FF	CR	SO	SI
0010...	DLE	DC1	XON	DC3	XOF	NAK	SYN	ETB
0011...	CAN	EM	SUB	ESC	FS	GS	RS	US
0100...		!	"	#	$	%	&	'
0101...	(	)	*	+	,	-	.	/
0110...	0	1	2	3	4	5	6	7
0111...	8	9	:	;	<	=	>	?
1000...	@	A	B	C	D	E	F	G
1001...	H	I	J	K	L	M	N	O
1010...	P	Q	R	S	T	U	V	W
1011...	X	Y	Z	[	\	]	^	_
1100...	`	a	b	c	d	e	f	g
1101...	h	i	j	k	l	m	n	o
1110...	p	q	r	s	t	u	v	w
1111...	x	y	z	{	|	}	~	DEL

ASCII-Standard

American Standard Code for Information Interchange

- verabschiedet 1963 von der American Standards Organization (ASO)
- 7 Bit-Code: Zeichen mit Werten 0…127 codiert
- für die USA gedacht – keine Umlaute, Accents usw.
- codiert Zeichen und Steuercodes

ISO-8859-1

Problem: viele wichtige Zeichen fehlen!

Lösung: "längere Codierung": Code erweitern (z.B. 8 Bit)

Anmerkung: Erweitern ist besser als ersetzen.
- bekannteste Erweiterung hier ist ISO-8859-1
- International Organization for Standardization (gegr. 1947)

ISO-8859-1 = ISO Latin 1
- 8 Bit-Code
- enthält viele Sonderzeichen für westeuropäischeSprachen
  - z.B. Umlaute (ä, ö, ü, …)
- enthält nicht alle gewünschten Zeichen (z.B. japanische)

Unicode

Standard zur Codierung *aller möglicher* Zeichen

- aktueller Standard Unicode 15.1 (12. September 2023)
- verwaltet vom Unicode-Konsortium (http://www.unicode.org)
- unterstützt verschiedene Codierungsformate
  - Unicode Transformation Format: keine konstante Länge! UTF-8, UTF-16, UTF-32 mit 8, 16, 32 Bits
  - längere Formate erweitern kürzere Formate
  - vereinbart auch weitere Informationen (Codepoints)
    - z.B. Schreibrichtung, Kombination von Zeichen

- UTF-8 von Ken Thompson and Rob Pike entworfen
  - Unix-/C-Erfinder!
- Windows verwendet standardmäßig UTF-16!

UTF-8-Eigenschaften

Eigenschaften

- Codes 0-127 identisch zu ASCII!
- Codes >= 128 benötigen mehr als 1 Byte
  - Bit 1 in den vorherigen Bytes gesetzt!

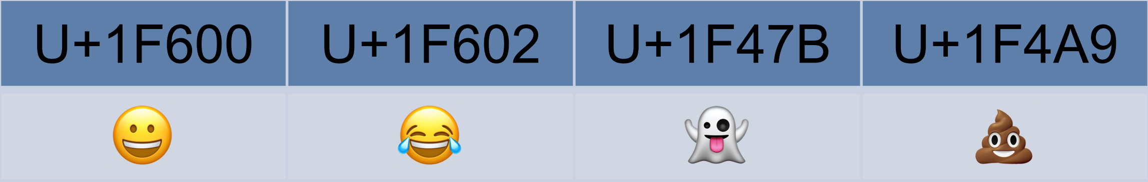
First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Character		Binary code point	Binary UTF-8	Hex UTF-8
$	U+0024	010 0100	00100100	24
£	U+00A3	000 1010 0011	11000010 10100011	C2 A3
И	U+0418	100 0001 1000	11010000 10011000	D0 98
ह	U+0939	0000 1001 0011 1001	11100000 10100100 10111001	E0 A4 B9
€	U+20AC	0010 0000 1010 1100	11100010 10000010 10101100	E2 82 AC

Unicode-Beispiele

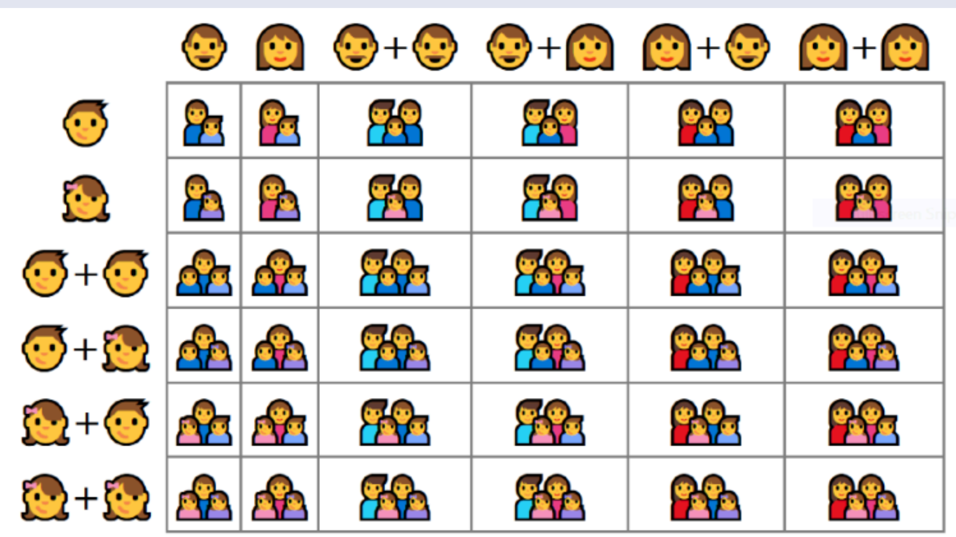
Was sollte Teil der Zeichencodierung sein?

Emojis…



U+1F600	U+1F602	U+1F47B	U+1F4A9
😀	😂	👻	💩

Kombination von Zeichen schafft "neue" Emojis



Sicherheitsprobleme, z.B. kyrillisches "a" (U+0430) in URLs

Problem: Stringlänge

Wie werden **Strings** (Zeichenketten) dargestellt?

Problem: Erkennung des String-Endes bzw. der Stringlänge

C: Nullbyte (ASCII NUL, binär 0) als Zeichen = Ende

Sicherheitsproblem Buffer Overflow ⇒ später…

Pascal: Erstes Byte des Strings enthält Länge

Implementierungsproblem: Länge von Strings <= 255

Referenzen

[1] Frank Slomka, Michael Glaß
Grundlagen der Rechnerarchitektur
Von der Schaltung zum Prozessor

[2] David Harris, Sarah Harris
Digital Design and Computer Architecture, RISC-V Edition

[3] David Goldberg
What Every Programmer Should Know About Floating-Point Arithmetic
ACM Computing Surveys March 1991
https://dl.acm.org/doi/10.1145/103162.103163