2. Information und deren Darstellung

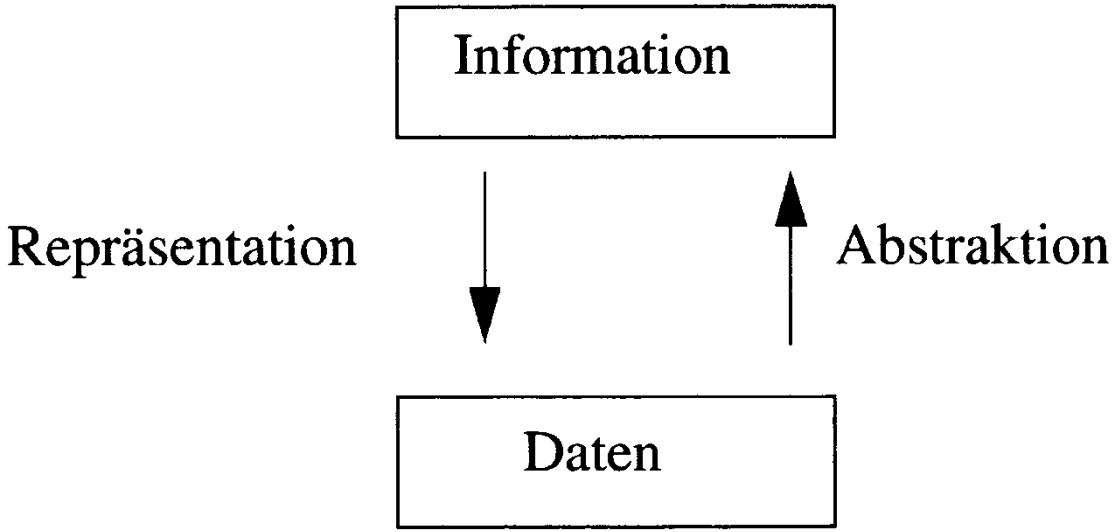
- **Was ist Information?**
  - Informatik: Die Wissenschaft von der systematischen Verarbeitung von ***Informationen***.
  - Begriff Information ist von zentraler Bedeutung für die Informatik
- **Versuch einer Definition:**
  - Broy, „Informatik - Eine grundlegende Einführung“, Band 1, Springer 1998:
    - ***Information*** nennen wir den abstrakten Gehalt („Bedeutungsinhalt“, „Semantik“) eines Dokumentes, einer Aussage, Beschreibung, Anweisung, Nachricht oder Mitteilung.
    - Die äußere Form der Darstellung nennen wir ***Repräsentation*** (konkrete Form der Nachricht) oder ***Daten***.

Information und Daten

- **Dieselbe Information kann auf verschiedene Weisen repräsentiert (dargestellt) werden**
  - Begriffe
    - deutsch, englisch, Schreibschrift, Druckschrift, ...
  - Zahlen
    - Dezimalzahlen, Römische Zahlen, ...
- **Dieselben Daten können verschiedene Informationen darstellen**
  - IC
    - Intercity
    - *Integrated Circuit* (Integrierte Schaltung)
  - PCB
    - Polychlorierte Biphenyle
    - *Printed Circuit Board* (gedruckte Schaltung, "Motherboard")

Information und Daten (2)

- **Daten**
  - Repräsentation der Informationen
- **Information**
  - Abstrakter Inhalt der Daten



Bits

- **Informationen werden durch zwei sich gegenseitig ausschließende Zustände repräsentiert**

Ein Bit (= **Bi**nary Digi**t**) ist entweder "0" oder "1"

Früher auch: „O“ und „L“ (L: Loch in Lochkarte)

- **technisch leicht und eindeutig zu unterscheiden**

Bits (2)

Daten: 0, 1

Information	
0	1
nein	ja
falsch	wahr
weiß	schwarz
gerade	ungerade
nicht best.	bestanden
...	...

Techn. Realisierung	
0	1
ungeladen	geladen
0 Volt	5 Volt
unmagnetisiert	magnetisiert
kein Licht	Licht
kein Loch	Loch
...	...

Bitfolgen

- **Bitfolgen (Bitstrings)**
  - mit einem Bit können nur $2^1 = 2$ Werte dargestellt werden
  - mit n Bits können 2^n verschiedene Werte dargestellt werden
  - müssen k ($= 2^n$) Werte dargestellt werden, benötigt man $n = \log_2(k)$ Bits
    - $\log_2(x)$: Zweierlogarithmus, häufig auch als ld(x) geschrieben (logarithmus dualis)
  - im allgemeinen benötigen wir $n = \lceil \log_2(k) \rceil$ Bits, falls k keine Zweierpotenz ist ($\lceil ... \rceil$ bedeutet: aufgerundet auf die nächste ganze Zahl).
- **Beispiele**
  - 2 Bits: 00 01 10 11
  - 3 Bits: 000 001 010 011 100 101 110 111

Bitfolgen (2)

- **Beachte:**
  - 6 Bits genügen, um alle
    - 26 Großbuchstaben
    - 26 Kleinbuchstaben
    - 10 Ziffern
    - 1 Leerzeichen

    darzustellen
  - nur noch ein weiteres Sonderzeichen möglich

Hexziffern

- **16 verschiedene Bitfolgen mit vier Bits**
- **jede bekommt einen Namen**
  - Ziffern '0' ... '9'
  - Zeichen 'A' ... 'F'

0000 = 0	0100 = 4	1000 = 8	1100 = C
0001 = 1	0101 = 5	1001 = 9	1101 = D
0010 = 2	0110 = 6	1010 = A	1110 = E
0011 = 3	0111 = 7	1011 = B	1111 = F

Hexziffern (2)

- **Lange Bitfolgen**
  - lange Folge von Nullen und Einsen sind für den Menschen sehr unübersichtlich

    00111101010001010010101001101100
  - Aufteilen in 4er-Gruppen (hinten angefangen)

    0011 1101 0100 0101 0010 1010 0110 1100
  - Zuordnen des Namens zu jeder 4er-Gruppe

    3 D 4 5 2 A 6 C
  - kompakte **Hexdarstellung**

    $3D452A6C_{16}$
  - daneben gibt es noch die **Oktaldarstellung** (analog, Ziffern 0…7):

    00111101010001010010101001101100

    00 111 101 010 001 010 010 101 001 101 100

    07521225154_8

Oktett

- **Rechner liest oder schreibt Daten immer in Gruppen von mehreren Bits**
  - es wäre zu langsam, Bits immer einzeln zu behandeln
  - derzeit sind üblich: 8, 16, 32, 64 Bits
  - Rechner werden entsprechend genannt: 8-Bit-Rechner, 16-Bit-Rechner, etc.
  - praktisch immer Vielfache von 8 Bit
- **Oktett**
  - **Tupel** (geordnete Menge) von 8 Bit
  - darstellbar mit zwei Hex-Ziffern (00, ..., FF)
  - kann $256 = 2^8 = 16^2$ verschiedene Werte annehmen

Bytes

- **Byte**
  - ursprünglich: *bite* (engl. für Happen), wegen Verwechslungsgefahr mit Bit später in Byte umbenannt
- **genaue Bedeutung hängt vom Anwendungsgebiet ab**
  - Maßeinheit für Datenmenge von 8 Bit (Einheitenzeichen "B")
  - geordnete Menge (Tupel) von 8 Bit (also ein Oktett)
  - adressierbare Speichereinheit, in der ein Zeichen aus einem vorgegebenen Zeichensatz gespeichert werden kann:
    - Telex: 1 Zeichen = 5 Bits = 1 Byte
    - ASCII: 1 Zeichen = 7 Bits = 1 Byte
    - PC: 1 Zeichen = 8 Bits = 1 Byte
    - UNIVAC 1100 1 Zeichen = 9 Bits = 1 Byte
  - Datentyp in einer Programmiersprache
    - Anzahl Bits kann von Programmiersprache und Plattform abhängen
  - In der Praxis werden die Begriffe Byte und Oktett allerdings fast immer synonym verwendet.

Präfixe für Maßeinheiten

- **Zweierpotenzen**
  - sind sinnvoller bei binären Größen
  - (un)glücklicherweise liegt 2^{10} nahe bei 1000

$$2^{10} = 1024$$

  - daher steht in der Informatik **Kilo** auch schon mal für 1024
  - entsprechend steht **Mega** für

$$2^{20} = 2^{10} * 2^{10} = 1024 * 1024 = 1048576$$

Präfixe für Maßeinheiten (2)

- SI-System (Système International à Unites)
- IEC (International Electrotechnical Commission)

1998 von der IEC eingeführt

SI-Präfixe				Binärpräfixe	
Name (Symbol)	**SI-konforme Bedeutung**	**häufig gemeinte Bedeutung**	**% Unterschied**	**Name (Symbol)**	**Bedeutung**
Kilobyte (kB)	10^3 Byte	2^{10} Byte	2,4 %	Kibibyte (KiB)[1)]	2^{10} Byte
Megabyte (MB)	10^6 Byte	2^{20} Byte	4,9 %	Mebibyte (MiB)	2^{20} Byte
Gigabyte (GB)	10^9 Byte	2^{30} Byte	7,4 %	Gibibyte (GiB)	2^{30} Byte
Terabyte (TB)	10^{12} Byte	2^{40} Byte	10,0 %	Tebibyte (TiB)	2^{40} Byte
Petabyte (PB)	10^{15} Byte	2^{50} Byte	12,6 %	Pebibyte (PiB)	2^{50} Byte
Exabyte (EB)	10^{18} Byte	2^{60} Byte	15,3 %	Exbibyte (EiB)	2^{60} Byte
Zettabyte (ZB)	10^{21} Byte	2^{70} Byte	18,1 %	Zebibyte (ZiB)	2^{70} Byte
Yottabyte (YB)	10^{24} Byte	2^{80} Byte	20,9 %	Yobibyte (YiB)	2^{80} Byte

[1)] wird häufig auch mit KB abgekürzt.

Längen- und Zeiteinheiten

- **Zeiteinheiten**
  - hierfür werden auch in der Informatik Zehnerpotenzen benutzt
  - Beispiel 4 GHz Prozessor
    - Taktfrequenz: $4 \cdot 10^9$ Hz (Schwingungen pro Sekunde)
    - Schwingungsdauer (Kehrwert der Taktfrequenz):

      $0{,}25 \cdot 10^{-9}$ s = 0,25 ns = 250 ps

m	milli	10^{-3}
µ	mikro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

Längen- und Zeiteinheiten (2)

- **Längeneinheiten**
  - neben **metrischen** Einheiten werden auch **amerikanische** Einheiten benutzt

    1" = 1 in = 1 inch = 1 Zoll = 2,54 cm

  - Teile eines Zolls werden in Brüchen angegeben
    - z.B. 3 1/2", 5 1/4" (Diskettengrößen)
    - 1/10" (Abstand der Anschlüsse gängiger IC's)
  - 12 inch sind 1 foot (1ft = 30,48 cm)
  - 3 foot sind ein yard (1 yd = 91,44 cm)

Bytes, Worte, ...

- **Für Gruppen von Bytes gibt es die Begriffe**
  - Wort, Doppelwort, Quadwort
- **Verwendung nicht einheitlich**
  - Wort stellt die "natürliche" Anzahl von Bytes dar, die in einem Schritt verarbeitet werden

# Bytes	16-Bit-Rechner	32-Bit-Rechner
1	Halbwort, Byte	Byte
2	Wort	Halbwort
4	Doppelwort	Wort
8	Quadwort	Doppelwort

Zahlen

- Bits sind nur Bits (keine inhärente Bedeutung)
  - Konventionen definieren die Beziehung zwischen Bits (Daten) und Zahlen (Informationen)
- Binärzahlen (n Bits, Basis 2)
  - Beispiel mit $n = 4$ Bits:

    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 ...
  - dezimal: $0...2^n-1$
- tatsächlich ist es noch komplizierter
  - Zahlen haben nur endlich viele Bits (führt zu Überläufen)
  - es gibt negative Zahlen
  - es gibt rationale und reelle Zahlen

Stellenwertkodierung

- **Idee**
  - nicht-negative ganze Zahlen mit Stellenwertkodierung darstellen
    - Ziffern
    - Ziffern an verschiedenen Stellen haben verschiedene Wertigkeiten
  - Beispiel: Dezimalsystem

    $4711_{10} = 4 \bullet 10^3 + 7 \bullet 10^2 + 1 \bullet 10^1 + 1 \bullet 10^0$
  - analog: Binärsystem

    $11010_2 = 1 \bullet 2^4 + 1 \bullet 2^3 + 0 \bullet 2^2 + 1 \bullet 2^1 + 0 \bullet 2^0 = 26_{10}$
  - allgemein g-adische Darstellung

$$w = \sum_{i=0}^{n-1} b_i g^i, \quad \text{mit} \quad 0 \le b_i < g$$

  - Darstellung mit Ziffern b_i ist eindeutig

Binärsystem: vorzeichenlose Zahlen

- **"vorzeichenlose Zahlen" ist ein anderer Begriff für positive (genauer: nicht negative) Zahlen**
  - haben kein Vorzeichen, weil sie keins brauchen, da das Vorzeichen immer "+" ist

natürliche Zahlen mit 0

$$w : \{0,1\}^n \rightarrow \mathrm{N}_0$$

$$b_{n-1}\ldots b_1 b_0 \mapsto w\left(b_{n-1}\ldots b_1 b_0\right) = \sum_{i=0}^{n-1} b_i 2^i$$

- **2^n Zahlen mit n Bits darstellbar: 0, ..., 2^n-1**
- **jede Zahl hat eine eindeutige Kodierung in n Bits**

Addition von Binärzahlen

- **Zahlen können in jedem Zahlensystem wie gewohnt addiert werden**

	Dezimal	Binär	Oktal	Hexadezimal
	2752	010010	2752	27CA
+	4261	100111	4261	AF93
=	7013	111001	7233	D75D

  - der Computer verwendet meist feste Wortbreiten
  - ein Übertrag (*carry*) über die höchstwertige Stelle hinaus wird häufig ignoriert
    - Die Rechnung stimmt nur „ modulo 2^n “.

Vorzeichenbehaftete ganze Zahlen

- **Höchstwertiges Bit kodiert Vorzeichen**

$$w : \{0,1\}^n \rightarrow \mathbf{Z}$$

ganze Zahlen

$$(b_{n-1}b_{n-2}\ldots b_0) \mapsto w(b_{n-1}b_{n-2}\ldots b_0) = f\left(b_{n-1}, \sum_{i=0}^{n-2} b_i 2^i\right)$$

Vorzeichenbit

gesonderte Behandlung des Vorzeichens

Betrag und Vorzeichen

- **Zahlenwert ergibt sich aus**

$$w(b_{n-1}b_{n-2}\ldots b_0) = \begin{cases} +\sum_{i=0}^{n-2} b_i 2^i & \text{, falls } b_{n-1} = 0 \\ -\sum_{i=0}^{n-2} b_i 2^i & \text{, falls } b_{n-1} = 1 \end{cases}$$

- **Beispiele für *n*=6**

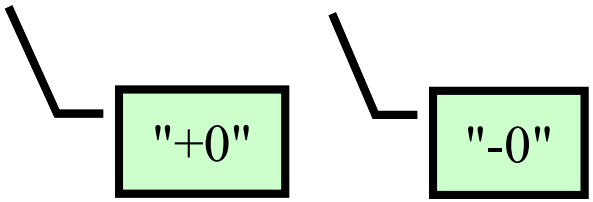
$w(011010) = +\ 1 \bullet 2^4 + 1 \bullet 2^3 + 0 \bullet 2^2 + 1 \bullet 2^1 + 0 \bullet 2^0 = +26_{10}$

$w(111010) = -(1 \bullet 2^4 + 1 \bullet 2^3 + 0 \bullet 2^2 + 1 \bullet 2^1 + 0 \bullet 2^0) = -26_{10}$

Betrag und Vorzeichen (2)

- **Zahlenbereich ist symmetrisch**
  - Wertebereich: $-(2^{n-1}-1)$, ..., $+2^{n-1}-1$
  - insgesamt 2^n-1 Zahlen
- **Darstellung der 0 ist redundant**

$$w(0000...0) = w(1000...0) = 0_{10}$$



  - Vergleichsoperationen schwer zu implementieren

Addition bei Betrag und Vorzeichen

- **Addition mit negativen Zahlen**
  - komplizierter als normale binäre Addition

0011_2		3_{10}
$+1001_2$		$+ -1_{10}$
1100_2	$\neq$	2_{10}

  - technisch zwar realisierbar
    - Fallunterscheidungen notwendig
  - die folgende Zweierkomplementdarstellung vermeidet aber alle genannten Nachteile

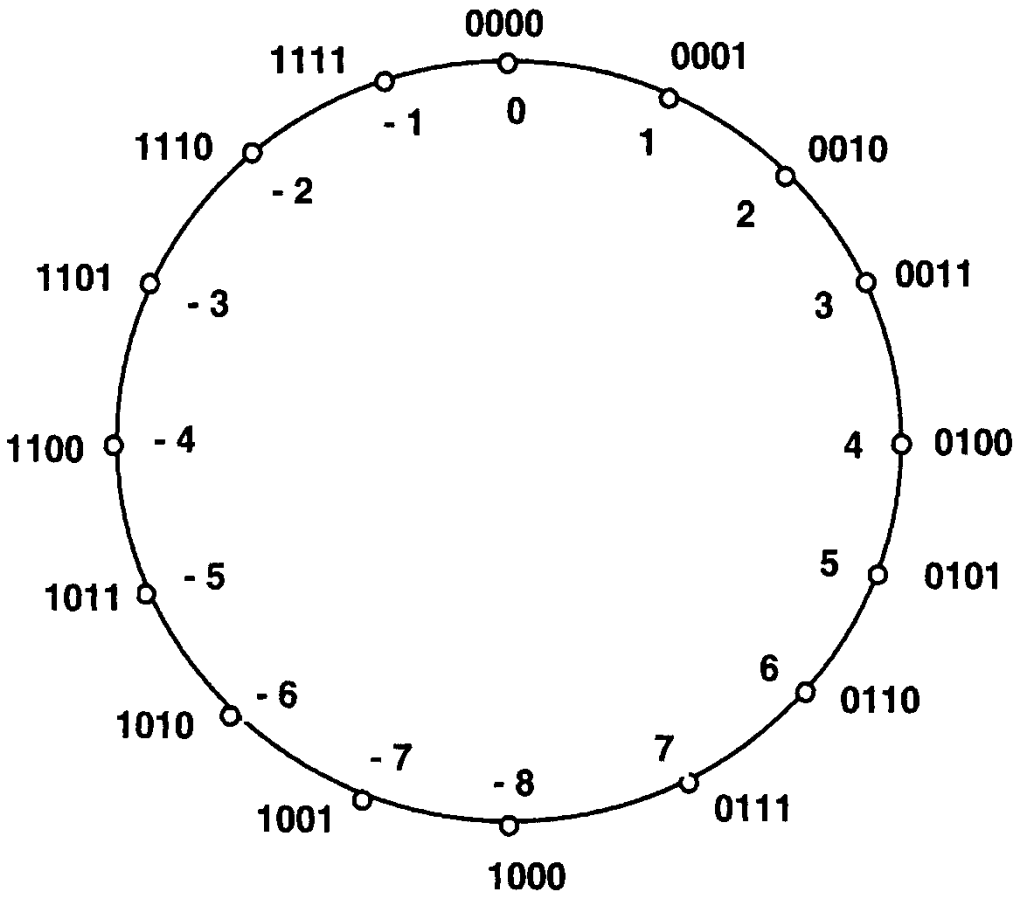
Zweierkomplementdarstellung

- **Wert in Zweierkomplementdarstellung**

$$w(b_{n-1}b_{n-2}\ldots b_0) = -b_{n-1}2^{n-1} + \underbrace{\sum_{i=0}^{n-2} b_i 2^i}_{0,\ldots,2^{n-1}-1}$$

  - für b_{n-1}=0 erhält man die positiven Zahlen wie gehabt
  - für b_{n-1}=1 wird 2^{n-1} vom Wert subtrahiert
  - Zahlenbereich: -2^{n-1}, ..., $2^{n-1}-1$
  - Zahlenbereich **nicht symmetrisch**
  - Darstellung der 0 **nicht redundant**

Zweierkomplementdarst. im Zahlenkreis



Zweierkomplement, negative Werte

- **Zweierkomplement**
  - Alternative Bestimmung des Wertes einer negativen Zahl

  1. Invertiere alle Bits
  2. Bestimme den Wert der nun positiven Zahl
  3. Addiere eine 1
  4. Das Ergebnis ist der Betrag der ursprünglich negativen Zahl

  (Beweis siehe unten)

- **Beispiel: 1011001_2**
  1. 0100110_2
  2. $2^5+2^2+2^1 = 32+4+2 = 38$
  3. $38+1 = 39$
  4. Ergebnis: -39
- **Laut Formel**

  1011001_2

  $= -2^6+2^4+2^3+2^0$

  $= -64+16+8+1$

  $= -64+25$

  $= -39$

Negation in Zweierkomplementdarstellung

Invertierung aller Bits, also Übergang von b_i nach $\overline{b_i}$, entspricht Subtraktion der vorzeichenlosen Zahl von $111111...1_2$, denn z.B.

$$011001...1_2 + 100110...0_2 = 111111...1_2 = 2^n\text{-}1$$

Allgemein:

$$\sum_{i=0}^{n-1} \overline{b_i} 2^i = 2^n - 1 - \sum_{i=0}^{n-1} b_i 2^i$$

Also gilt für $b_{n-1} = 1$

$$\sum_{i=0}^{n-1} \overline{b_i} 2^i + 1 = 2^n - \sum_{i=0}^{n-1} b_i 2^i$$

$$= 2^n - b_{n-1} 2^{n-1} - \sum_{i=0}^{n-2} b_i 2^i = 2^n - 2^{n-1} - \sum_{i=0}^{n-2} b_i 2^i$$

$$= 2^{n-1}(2-1) - \sum_{i=0}^{n-2} b_i 2^i = -\left(-2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \right)$$

Addition in Zweierkomplementdarstellung

- **Addition funktioniert wie normale Binäraddition vorzeichenloser Zahlen**
  - Übertrag (*carry*) aus der höchstwertigen Stelle **muss** dabei **ignoriert** werden
  - Beispiele

0011_2		3_{10}	1100_2		-4_{10}
$+1001_2$	+	-7_{10}	$+1101_2$	+	-3_{10}
1100_2		-4_{10}	$\not{1}1001_2$		-7_{10}

- **Beweis: selbst durch Fallunterscheidung**

Zweierkomplementdarstellung

- **Zweierkomplementdarstellung**
  - ist die bevorzugte Methode zur Darstellung ganzer Zahlen in modernen Computern
- **Achtung**
  - nicht für jede Zahl existiert auch deren Zweierkomplement
  - Beispiel für n=6

$$100000_2 \qquad -32_{10}$$
$$011111_2 \qquad 31_{10}$$
$$100000_2 \neq 32_{10}$$

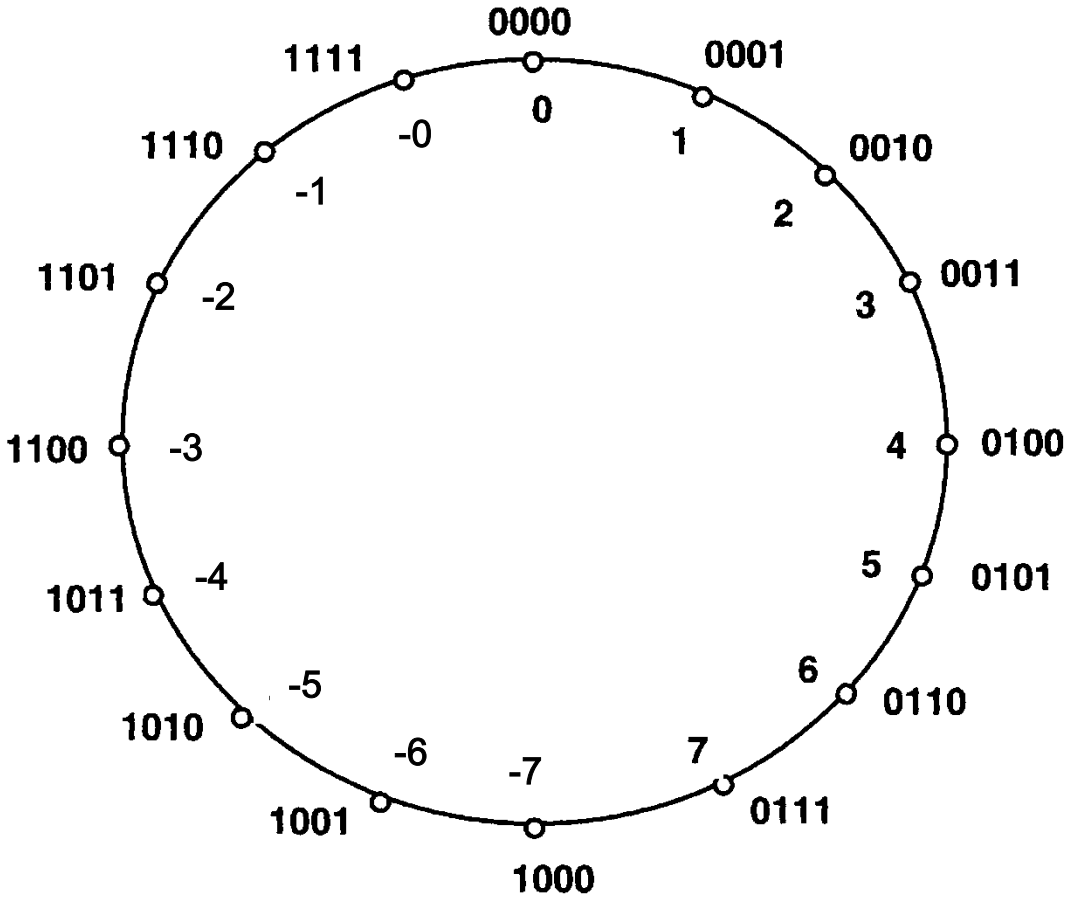
Einerkomplementdarstellung

- **Wert in Einerkomplementdarstellung**

$$w(b_{n-1}b_{n-2}\ldots b_0) = -b_{n-1}\left(2^{n-1}-1\right) + \underbrace{\sum_{i=0}^{n-2} b_i 2^i}_{0,\ldots,2^{n-1}-1}$$

  - für b_{n-1}=0 erhält man die positiven Zahlen wie gehabt
  - für b_{n-1}=1 wird 2^{n-1}-1 vom Wert subtrahiert
  - Zahlenbereich: -(2^{n-1}-1), ..., 2^{n-1}-1
  - Zahlenbereich **symmetrisch**
  - Darstellung der 0 **redundant**
  - Negation einer Zahl: Invertierung aller Bits (s.u.)

Einerkomplementdarstellung im Zahlenkreis



Einerkomplement, negative Werte

- **Einerkomplement**
  - Alternative Bestimmung des Wertes einer negativen Zahl

  1. Invertiere alle Bits
  2. Bestimme den Wert der nun positiven Zahl
  3. Das Ergebnis ist der Betrag der ursprünglich negativen Zahl

  (Beweis siehe unten)

- **Beispiel: 1011001_2**
  1. 0100110_2
  2. $2^5+2^2+2^1 = 32+4+2 = 38$
  3. Ergebnis: -38
- **Laut Formel**

  1011001_2

  $= -(2^6-1)+2^4+2^3+2^0$

  $= -63+16+8+1$

  $= -63+25$

  $= -38$

Negation in Einerkomplementdarstellung

Es gilt für $b_{n-1} = 1$:

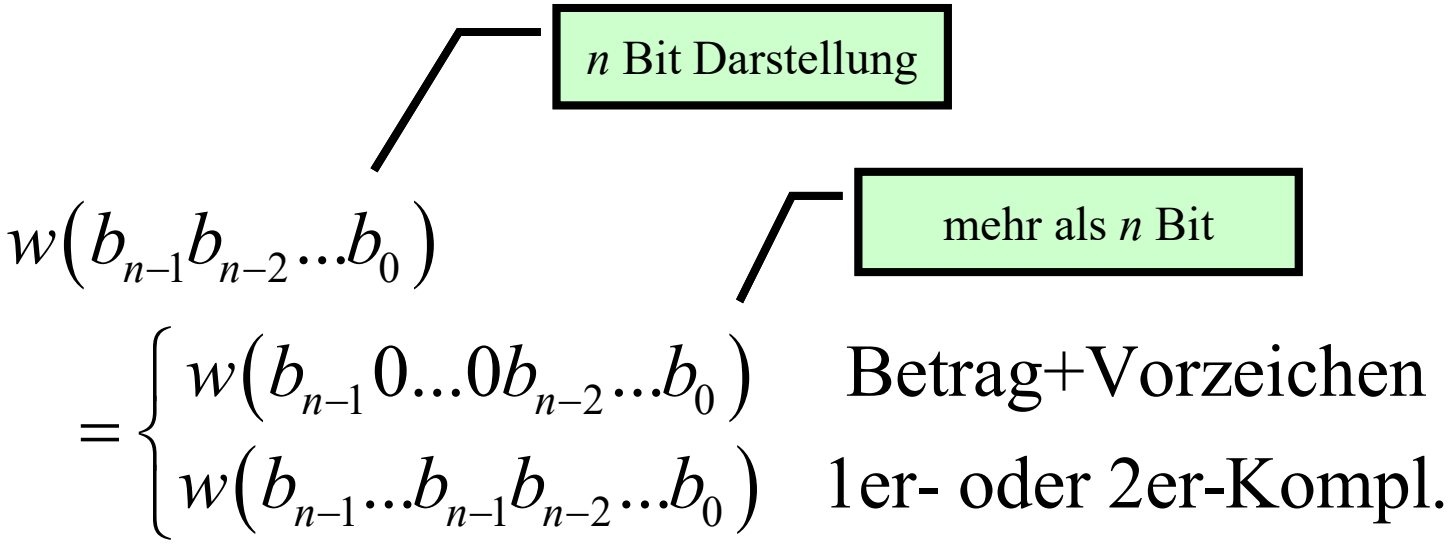
$$\sum_{i=0}^{n-1} \overline{b_i} 2^i = 2^n - 1 - \sum_{i=0}^{n-1} b_i 2^i$$

$$= 2^n - 1 - b_{n-1} 2^{n-1} - \sum_{i=0}^{n-2} b_i 2^i = 2^n - 2^{n-1} - 1 - \sum_{i=0}^{n-2} b_i 2^i$$

$$= 2^{n-1}(2-1) - 1 - \sum_{i=0}^{n-2} b_i 2^i = -\left(-\left(2^{n-1} - 1\right) + \sum_{i=0}^{n-2} b_i 2^i \right)$$

Einbettung in längere Zahldarstellung

- Zahlen müssen sich an den Wortlängen des verwendeten Computers orientieren
- Problem: Einbettung einer Zahl in eine Darstellung mit größerer Wortlänge



Beweis zur Einbettung

- **Beweis hier nur für Zweierkomplementdarstellung bei Erweiterung um ein einziges Bit**
  - zu zeigen: $w\left(b_{n-1}b_{n-1}b_{n-2}\ldots b_0\right) = w\left(b_{n-1}b_{n-2}\ldots b_0\right)$
  - 1. Fall: $b_{n-1} = 0$
    - positive Zahl, nichts zu zeigen
  - 2. Fall: $b_{n-1} = 1$

$$w\left(11b_{n-2}\ldots b_0\right)$$

$$= -1\cdot 2^n + 1\cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i = 2^{n-1}\left(-2+1\right) + \sum_{i=0}^{n-2} b_i 2^i$$

$$= -1\cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i = w\left(1b_{n-2}\ldots b_0\right)$$

Mögliche Darstellungen

Vorz. u. Betrag	Einer-Komplement	Zweier-Komplement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- **Welches ist die beste Darstellung? Wieso?**
  - Fragestellungen
    - Anzahl der Darstellungen für die Null
    - Einfachheit der Rechenoperationen
  - meist wird die Zweierkomplementdarstellung für ganze Zahlen verwendet

Big-Endian, Little-Endian

- **Speicherreihenfolge (Übertragungsreihenfolge)**
  - Little-Endian
    - niederwertigstes Byte wird zuerst gespeichert (übertragen)

    - ursprünglich eingeführt, weil bei der Addition von z.B. 4 Byte langen Zahlen mit dem niederwertigsten Byte begonnen werden muss
    - Intel Prozessoren benutzen diese Reihenfolge bis heute
  - Big-Endian
    - höchstwertiges Byte wird zuerst gespeichert (übertragen)

    - sieht natürlicher aus, da die Reihenfolge der uns vertrauten Notation von Dualzahlen entspricht

Fixpunktdarstellung

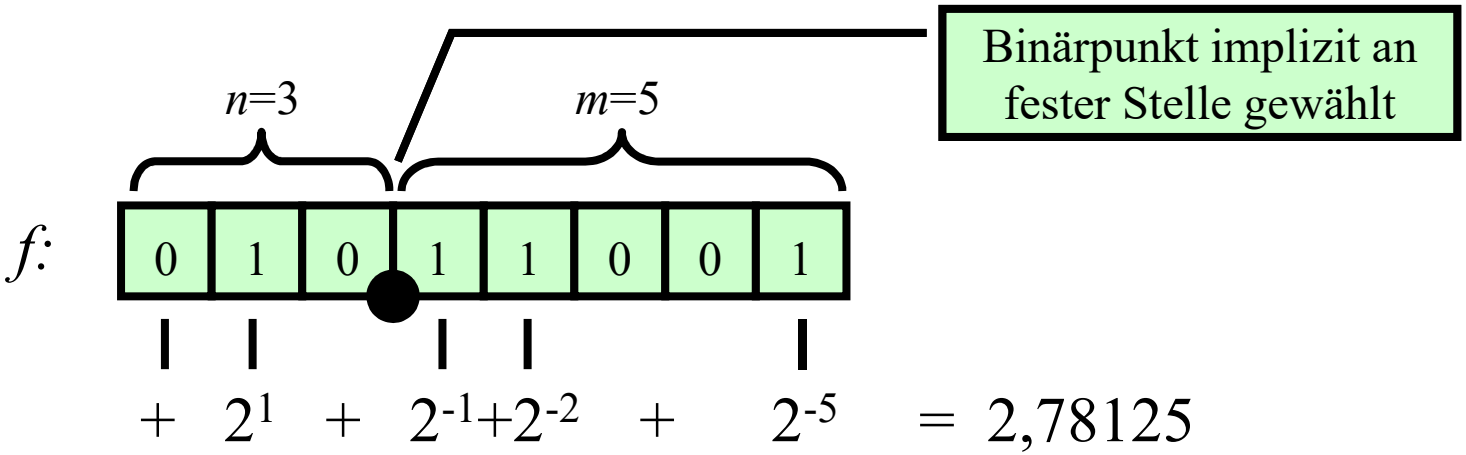
- **Dezimaldarstellung**
  - Dezimalkomma (Informatik: Dezimalpunkt!) steht rechts von der Stelle mit Wertigkeit 10^0
  - nachfolgende Stellen haben Wertigkeit 10^{-1}, 10^{-2}, etc., z.B. 3.14159
- **Binärdarstellung**
  - analog, Wertigkeiten rechts des ***gedachten*** "Binärpunktes": 2^{-1}, 2^{-2}, etc.
  - dadurch Möglichkeit, viele rationale Zahlen darzustellen

$$w\left(b_{n-1}b_{n-2}\ldots b_0 b_{-1}b_{-2}\ldots b_{-m}\right) = f\left(b_{n-1}, \sum_{i=-m}^{n-2} b_i 2^i\right)$$

gedachter Binärpunkt

  - meist wird Zweierkomplementdarstellung verwendet
    - Betrag und Vorzeichen, Einerkomplement wären auch möglich

Fixpunktdarstellung (2)



- *m*: Anzahl Nachkommastellen legt Genauigkeit fest
  - $\Delta f = 2^{-m}$, hier (m=5): $\Delta f = 2^{-5} = 1/32 = 0{,}03125$
- *n*: Anzahl Vorkommastellen legt möglichen Wertebereich fest
  - $-2^{n-1} \leq f < 2^{n-1}$, hier ($n$=3): $-4 \leq f < 4$ (bzw. $-4 \leq f \leq 3{,}96875$ bei m=5)

Fixpunktdarstellung (3)

- **Spezialfall: n=1**
  - Zahlen aus dem Bereich [-1, 1[(einschließlich -1, ausschließlich +1)
  - Abgeschlossenheit bzgl. Multiplikation, d.h. das Produkt zweier Zahlen fällt wieder in den darstellbaren Bereich
    - Ausnahme: (-1)*(-1) = +1
    - verzichtet man auf die -1, gibt es kein Problem
  - das darstellbare Zahlenintervall wird gleichmäßig durch die darstellbaren Zahlen abgedeckt
- **Verwendung**
  - Digitale Signalprozessoren (DSPs) und programmierbare Logikbausteine (Field Programmable Gate Arrays, FPGAs) arbeiten häufig mit diesem Zahlenformat
  - Mikroprozessoren (CPUs) und Grafikprozessoren (GPUs) arbeiten eher mit Gleitpunktzahlen (s.u.)

Fixpunktdarstellung (4)

- **Verschieben des Binärpunktes**
  - ein Bit nach rechts: Multiplikation mit 2
  - ein Bit nach links: Division durch 2
  - Verschiebung um k Stellen nach rechts: $*2^k$
  - Verschiebung um k Stellen nach links: $*2^{-k}$
  - Beispiel:

$$1100.1_2 = 12.5_{10}$$
$$11.001_2 = 3.125_{10} = 12.5_{10}/4_{10}$$
$$11001_2 = 25_{10} = 12.5_{10} * 2_{10} = 3.125_{10} * 8_{10}$$

Fixpunktdarstellung (5)

- **Umrechnung ins Fixpunktformat**
  - m Nachkommastellen
  - Dezimalzahl zunächst mit 2^m multiplizieren und auf ganze Zahl runden
  - dann ganze Zahl ins Binärsystem umrechnen
  - Ergebnis ist um den Faktor 2^m zu groß, daher muss anschließend wieder durch 2^m dividiert werden, d.h. der Binärpunkt muss um m Stellen nach links geschoben werden
- **Beispiel: 0.1_{10} in Fixpunktdarstellung mit n=3 und m=10**
  - skalieren: $0.1*2^{10} = 102.4$
  - runden: $102.4 \rightarrow 102$
  - umrechnen: $102_{10} = 64 + 32 + 4 + 2 = 1100110_2$
  - Binärpunkt: 000.0001100110
  - wegen Rundung ist das nicht exakt, sondern etwa: 0.099609…

Gleitpunktdarstellung

- **Darstellungsgenauigkeit**
  - bisher Fixpunktdarstellung
    - auch einfache Dezimalzahlen, z.B. 0.1, können in Fixpunktdarstellung mit endlich vielen Bits nicht exakt dargestellt werden
    - repräsentierter Wert ist ein gerundeter Wert
    - sehr große Zahlen , z.B. 2.99792×10^8
      - können nicht dargestellt werden, da die Wertigkeit des höchstwertigen Bits festgelegt ist → Überlauf (*overflow*)
    - sehr kleine Zahlen , z.B. 0.000000001
      - können nicht dargestellt werden, da Wertigkeit des niederwertigsten Bits festgelegt ist → Unterlauf (*underflow*)
- **wünschenswert**
  - großes Intervall des Zahlenstrahls darstellbar
  - möglichst konstante relative Genauigkeit über den gesamten Wertebereich

Gleitpunktdarstellung (2)

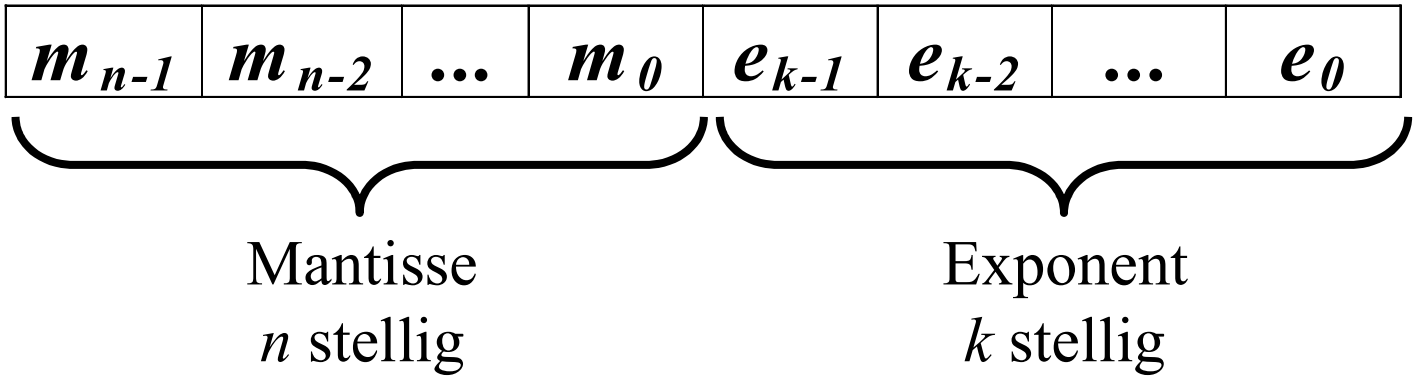
- **Lösung**
  - Gleitpunktdarstellung (*floating point representation*)
    - Hinzufügen eines Exponenten zur Zahldarstellung (Exponentialschreibweise)
    - Beispiel: $0.4711*10^4$
  - Darstellung mithilfe von
    - Vorzeichen
    - Mantisse (nur positiv, also eigentlich Betrag der Mantisse)
    - Exponent (kann negative Werte annehmen)

$$g = (-1)^{\text{Vorzeichen}} \times \text{Mantisse} \times 2^{\text{Exponent}}$$

  - mehr Bits für Mantisse erhöht Genauigkeit
  - mehr Bits für Exponent erhöht darstellbaren Zahlenbereich
  - relative Genauigkeit weitgehend konstant über den gesamten darstellbaren Zahlenbereich

Gleitpunktdarstellung (3)

- **Gleitpunktdarstellung, z.B. (wirklich nur ein Beispiel!)**



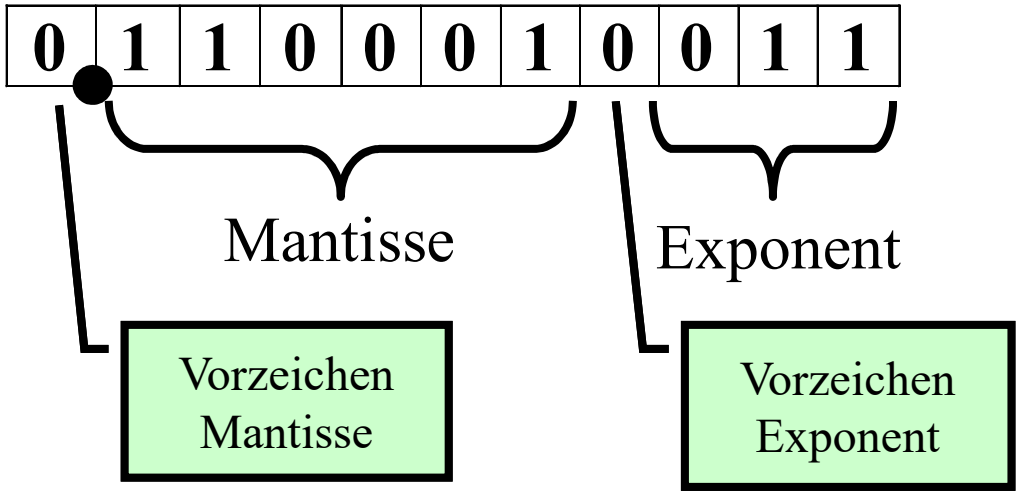
- **Berechnung des Wertes**

$$w(m_{n-1}\ldots m_0 e_{k-1}\ldots e_0) = w(m_{n-1}\ldots m_0) \cdot 2^{w(e_{k-1}\ldots e_0)}$$

Welche Zahldarstellungen nehmen wir hier?

Beispiel Gleitpunktdarstellung

- z.B.
  - Mantisse : 7-Bit-Fixpunkt in Zweierkomplement mit Punkt nach der ersten Stelle
  - Exponent: 4-Bit-Zweikomplement
- 6.125_{10} würde dann dargestellt werden als



- **Probe**

$$(0.110001)_2 \cdot 2^3 = (110.001)_2 = 2^2 + 2^1 + 2^{-3} = 4 + 2 + 0.125 = 6.125$$

Normalisierte Gleitpunktdarstellung

- **Darstellung bisher nicht eindeutig**

$$1.101100 \cdot 2^5 = 0.110110 \cdot 2^6 = 0.011011 \cdot 2^7$$

Eine Gleitpunktzahl zur Basis 2 heißt normalisiert, falls für die Mantisse *m* gilt:

$$1 \leq |w(m)| < 2$$

  - wir werden Betrag und Vorzeichen als Darstellung der Mantisse benutzen
  - bei normalisierten Mantissen ist das höchstwertige Bit des Betrages = 1
  - höchstwertiges Bit ist damit redundant und kann weggelassen werden
    - diese Darstellung der Mantisse nennt man auch "Signifikand"
    - (Vorsicht: in der Literatur wird manchmal auch die Mantisse selbst, also mit der nicht dargestellten 1, als Signifikand bezeichnet)

Interpretation der Darstellung

- **Welche Gleitpunktzahl ist das:**

0	1	1	0	0	0	1	0	0	1	1

?

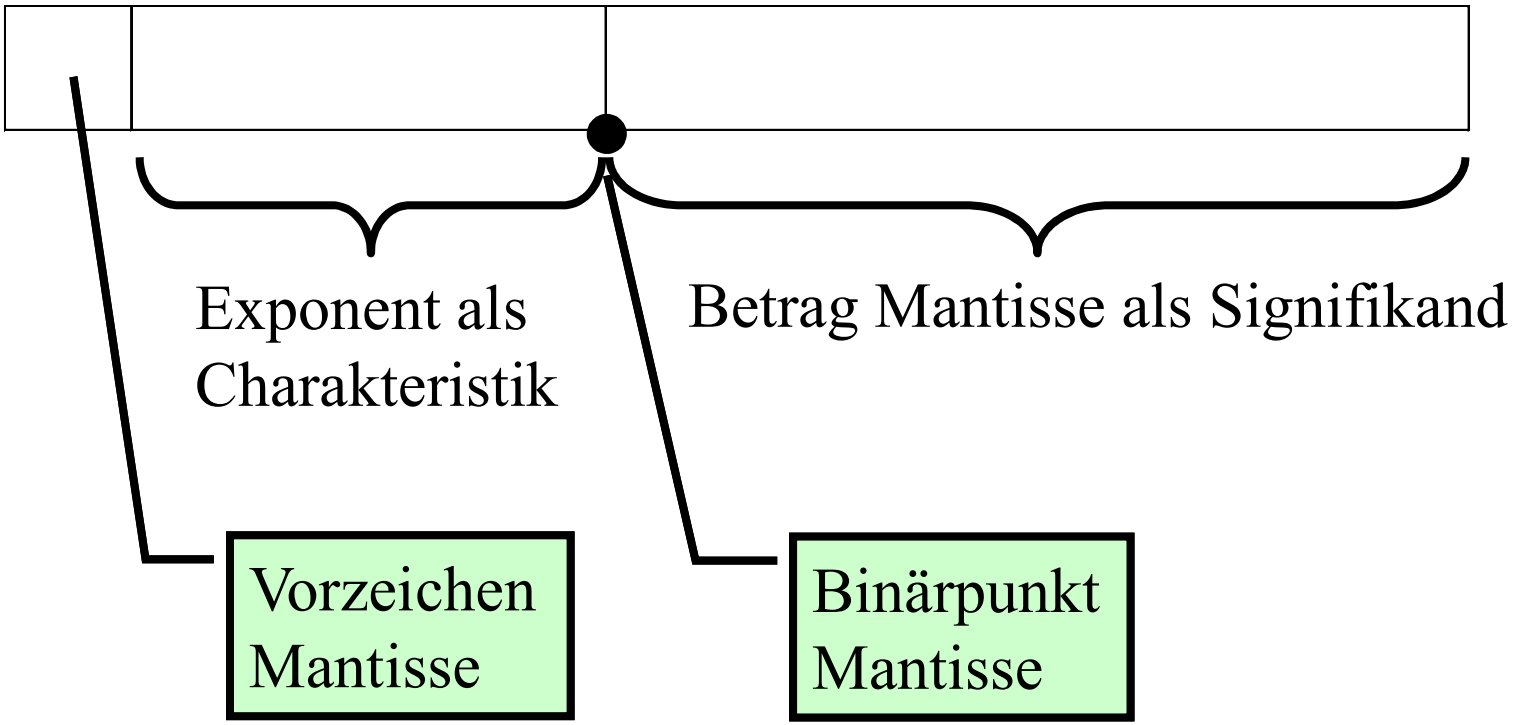
  - Ist das überhaupt eine Gleitpunktzahl?
  - Länge von Mantisse und Exponent?
  - Zuerst Mantisse oder zuerst Exponent?
  - Zahldarstellung für Mantisse?
  - Zahldarstellung für Exponent?
  - Normalisiert oder nicht?

- **Notwendigkeit der Standardisierung**

Gleitpunktdarstellung "IEEE 754"

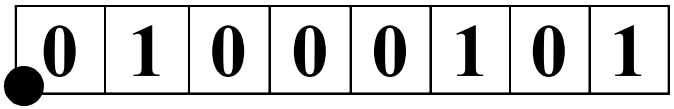
- **Standardisierung**
  - sehr sinnvoll, insbesondere bei der Datenkommunikation von Rechner zu Rechner
- **Institute of Electrical and Electronics Engineers (IEEE)**
  - sprich: I triple E ("ai trippel i")
  - begann 1979 mit der Erarbeitung eines Standards für Gleitpunktzahlen
  - veröffentlichte das Ergebnis 1985 als Standard "IEEE 754"
  - wird seitdem in fast allen Computern benutzt

Gleitpunktdarstellung "IEEE 754" (2)



Gleitpunktdarstellung "IEEE 754" (3)

- **Mantisse**
  - Darstellung als Betrag und Vorzeichen
  - Darstellung als Signifikand, d.h.
    - normalisiert
    - führende 1 wird weggelassen
    - Binärpunkt hinter führender 1 (vor dargestellten Bits)



$= 1.01000101_2$

Gleitpunktdarstellung "IEEE 754" (4)

- **Exponent**
  - vorzeichenlose ganze Zahl mit *bias* (*excess* [*bias*] *code*) (das ist eine weitere Möglichkeit, negative Zahlen darzustellen!)

> engl. *bias*: Hang, Neigung, Vorliebe, Vorurteil
> bezeichnet oft "Verschiebung um additive Konstante"

  - *bias* muss subtrahiert werden, um den wahren Exponenten zu erhalten
  - die Werte 0...0 und 1...1 sind reserviert (s.u.)
  - eine solche Darstellung eines Exponenten wird auch als **Charakteristik** bezeichnet

Vergleich der Ganzzahldarstellungen

Vorz. u. Betrag	Einer-Komplement	Zweier-Komplement	Excess 3 Code
000 = +0	000 = +0	000 = +0	000 = -3
001 = +1	001 = +1	001 = +1	001 = -2
010 = +2	010 = +2	010 = +2	010 = -1
011 = +3	011 = +3	011 = +3	011 = 0
100 = -0	100 = -3	100 = -4	100 = +1
101 = -1	101 = -2	101 = -3	101 = +2
110 = -2	110 = -1	110 = -2	110 = +3
111 = -3	111 = -0	111 = -1	111 = +4

benutzt für Mantissen bei Gleitpunktzahlen

benutzt für Exponenten bei Gleitpunktzahlen

Gleitpunktdarstellung "IEEE 754" (5)

- **Beispiel (mit 8 Bit Charakteristik und *bias* 127)**
  - wahrer Exponent: 12_{10}
  - Darstellung: $12_{10} + 127_{10} = 139_{10}$

±	1	0	0	0	1	0	1	1	Signifikand

  - wahrer Exponent: -2_{10}
  - Darstellung: $-2_{10} + 127_{10} = 125_{10}$

±	0	1	1	1	1	1	0	1	Signifikand

  - Charakteristik und Signifikand können zusammen wie vorzeichenlose Binärzahlen miteinander verglichen werden (<, =, >)
    - Stellenwert Kodierung: je weiter links, desto größer der Wert
    - das gilt genauso für den Exponenten (deshalb die Excess-Kodierung)
- **Zusammenfassung**

$$g = (-1)^{\text{Vorzeichen}} \times (1+\text{Signifikand}) \times 2^{\text{Charakteristik} - \text{Bias}}$$

Gleitpunktdarstellung "IEEE 754" (6)

- **Charakteristik 0...00**
  - nicht normalisierte Mantisse
  - führendes (weggelassenes) Bit ist jetzt 0
  - der Wert des Exponenten entspricht dem Wert des kleinsten darstellbaren Exponenten für normalisierte Mantissen (Charakteristik: 0…01)
  - dadurch sind noch kleinere Zahlen darstellbar
  - ist der Signifikand auch 0...0: Darstellung der Zahl 0 ("+0" oder "-0")
    - entsteht auch bei Rechenoperationen bei Unterlauf (*underflow*)
- **Charakteristik 1...11**
  - Signifikand 0..0
    - unendlich ("+∞" oder "-∞")
    - entsteht z.B. bei x/0 für $x \neq 0$
  - Signifikand != 0..0
    - NaN (*Not a Number*) undefiniertes Resultat
    - entsteht z.B. bei 0/0 oder 0*∞

Gleitpunktdarstellung "IEEE 754" (7)

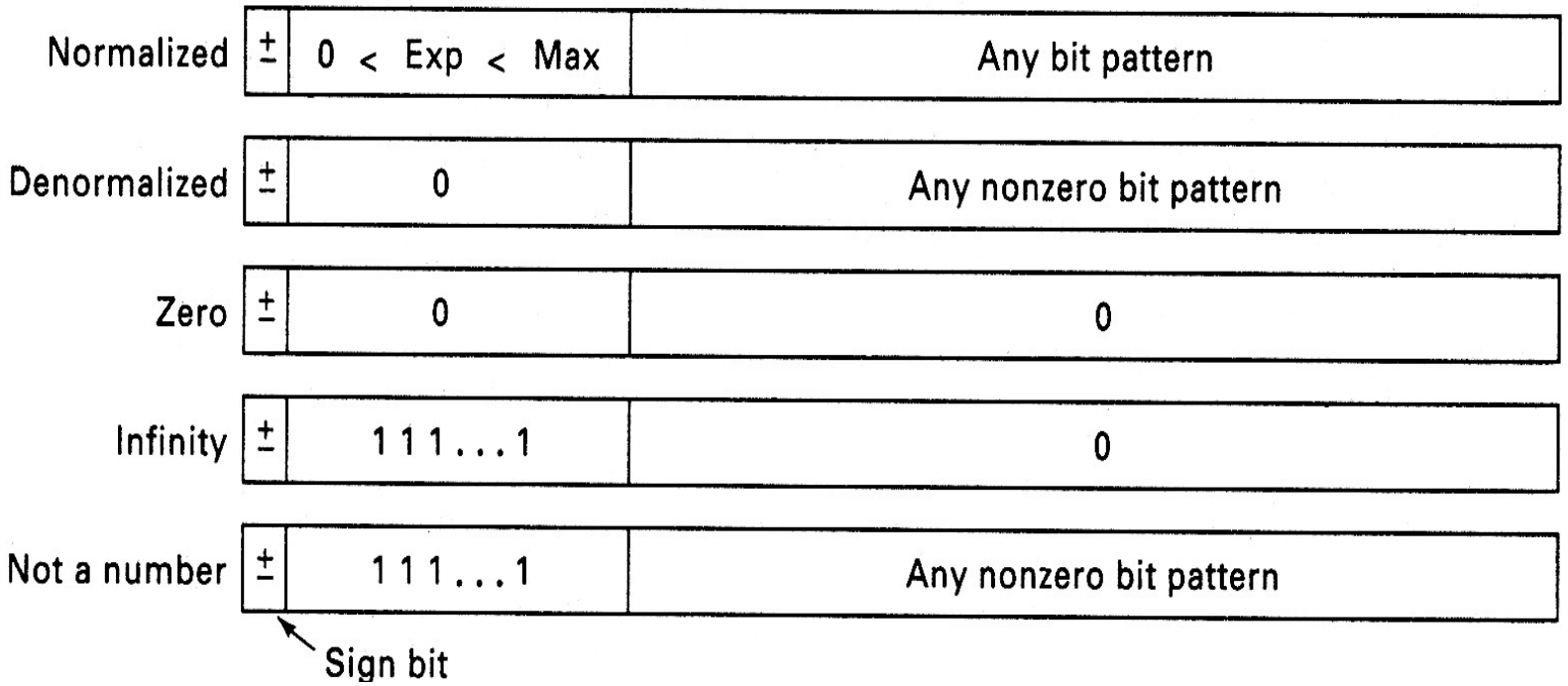
- **drei Formate standardisiert**
  - einfache Genauigkeit (*single precision*): 32 Bit
    - Genauigkeit: ca. 7 Dezimalstellen
  - doppelte Genauigkeit (*double precision*): 64 Bit
    - Genauigkeit: ca. 15 Dezimalstellen
  - erweiterte Genauigkeit (*extended precision*): 80 Bit
    - Genauigkeit: ca. 19 Dezimalstellen
    - wird nur ***innerhalb*** von Gleitpunkt-Rechenwerken (*floating point unit*, FPU) zur Reduzierung von Rechenungenauigkeiten (Rundungsfehlern) benutzt

Gleitpunktdarstellung "IEEE 754" (8)

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	−126 to +127	−1022 to +1023
Smallest, normalized	2^{-126}	2^{-1022}
Largest, normalized	approx. 2^{+128}	approx. 2^{+1024}
Decimal range	approx. 10^{-38} to 10^{+38}	approx. 10^{-308} to 10^{+308}
Smallest, denormalized	approx. 10^{-45}	approx. 10^{-324}

Gleitpunktdarstellung "IEEE 754" (9)

- **Zusammenfassung**



Gleitpunkt-Operationen

- **Rechenoperationen sind kompliziert**
  - zusätzlich zu *overflow* können wir auch *underflow* haben
  - viele Sonderfälle
    - positive Zahl dividiert durch 0 ergibt "*infinity*"
    - 0 dividiert durch 0 ergibt "*not a number*"
- **Genauigkeit kann ein großes Problem sein**
  - Daten müssen wegen der Darstellung mit Signifikand immer normalisiert werden
  - nach dem Normalisieren muss gerundet werden
  - vier verschiedene Rundungsarten
  - durch Runden kann die Zahl wieder denormalisiert werden
  - erneutes Normalisieren und Runden notwendig

Gleitpunkt-Operationen (2)

- **Vorsicht beim Programmieren mit Gleitpunktzahlen!**
  - auch "einfache" Dezimalzahlen sind im Binärsystem nicht mehr exakt darstellbar (Rundungsfehler), z.B.

$$0.1_{10} = 0.0001100110011001100110\ldots_2$$

  - bei arithmetischen Operationen entstehen weitere Ungenauigkeiten
  - Vergleich zweier Gleitpunktzahlen in Programmen ist problematisch
    - Beispiel mit Fixpunktzahlen mit 14 Nachkommastellen:

$$0.1*5 = 0.5 \quad ?$$

$$\begin{aligned} 0.1_{10} * 1_{10} &= 0.00011001100110_2 \\ \underline{0.1_{10} * 4_{10}} &= \underline{0.01100110011000_2} \\ 0.1_{10} * 5_{10} &= 0.01111111111110_2 \\ 0.5_{10} &= 0.10000000000000_2 \end{aligned}$$

Man muss beim Programmieren mit Fixpunkt- und Gleitpunktzahlen immer mit Rundungsfehlern rechnen!

Darstellung von Text

- **7 Bit pro Zeichen genügen für einfache Texte ($2^7 = 128$)**
  - 26 Kleinbuchstaben
  - 26 Großbuchstaben
  - 10 Ziffern

  } alphanumerische Zeichen

  - Sonderzeichen wie '&', '!', '"
  - nicht druckbare Steuerzeichen, z.B.
    - CR (*carriage return* = Wagenrücklauf)
    - LF (*line feed* = Zeilenvorschub)
    - TAB (Tabulator)
    - BEL (*bell* = Klingelzeichen)

ASCII-Code

- **Code-Erzeugung**
  - Tabelle mit 128 Zeichen
  - Einträge werden durchnummeriert
  - jedes Zeichen erhält als Code seine Position in der Tabelle als 7 stellige Binärzahl
  - das achte Bit eines Bytes b_7 ist entweder 0 oder wird als Paritäts-Bit (*parity bit*) genutzt (s.u.)

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

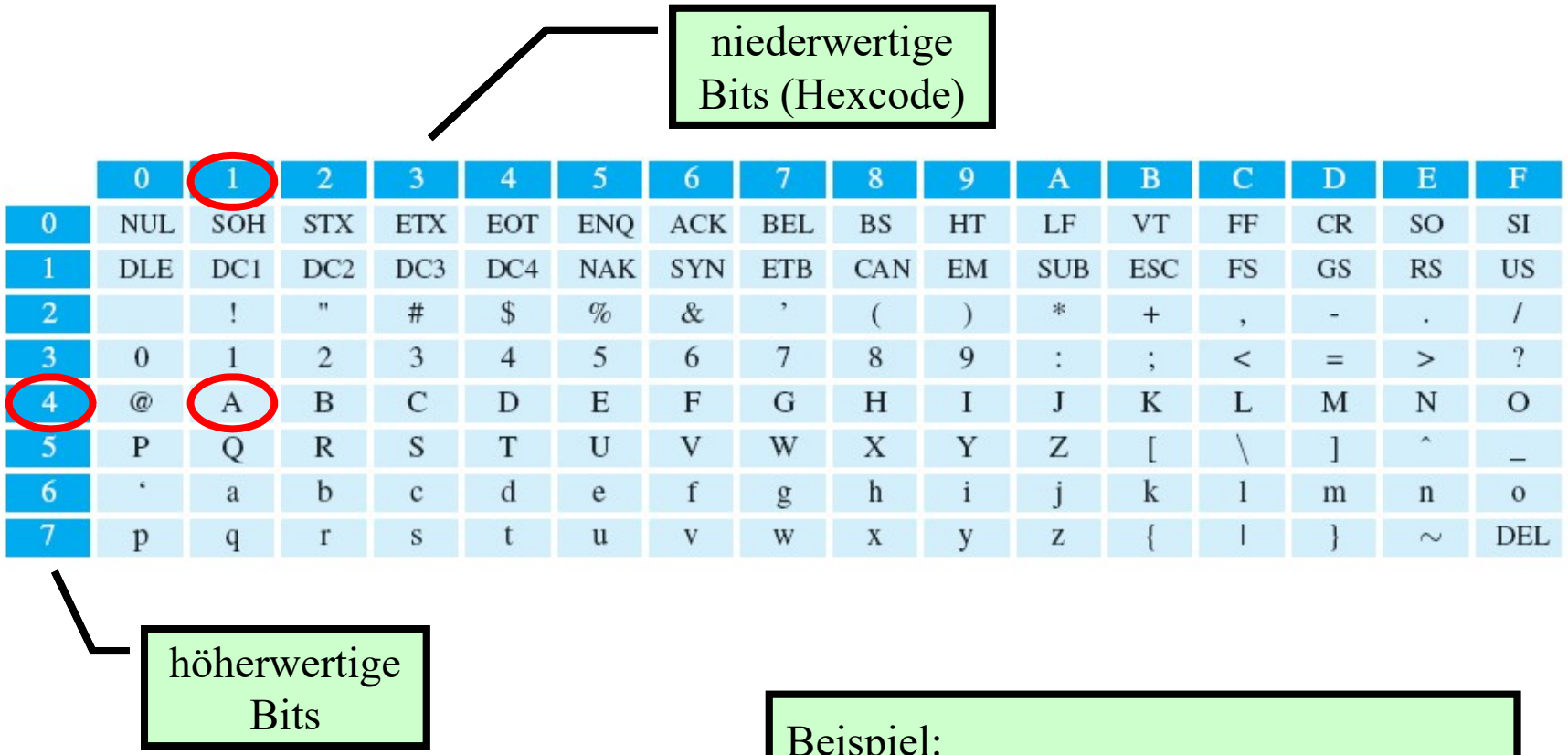
ASCII-Code (2)

- **Tabelle kann im Prinzip beliebig gewählt werden**
  - Sender und Empfänger von Informationen müssen aber dieselbe Tabelle verwenden

American Standard Code for Information Interchange

  - seit 1968 festgelegt
  - 8-Bit-Code, höchstwertiges Bit ist Parity-Bit (s.u.)
- **Systematik**
  - die 10 Ziffern folgen aufeinander
    - Bits $b_3b_2b_1b_0$ sind Binärdarstellung des Zahlenwertes
  - Klein- und Großbuchstaben sind jeweils alphabetisch sortiert

ASCII-Tabelle

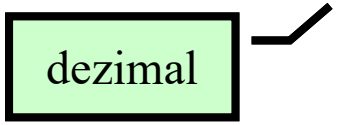


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Beispiel:

Buchstabe 'A'
hat ASCII-Code: 41_{16}

ASCII Steuerzeichen



000	NUL	Nil (Null)
001	SOH	Start of Heading
002	STX	Start of Text
003	ETX	End of Text
004	EOT	End of Transmission
005	ENQ	Enquiry
006	ACK	Acknowledge
007	BEL	Bell
008	BS	Backspace
009	HT	Horizontal Tabulation
010	LF	Line Feed
011	VT	Vertical Tabulation
012	FF	Form Feed
013	CR	Carriage Return
014	SO	Shift-out

ASCII Steuerzeichen (2)

015	SI	Shift-in
016	DLE	Date Link Escape
017-020	DC	Device Control
021	NAK	Negative Acknowledgement
022	SYN	Synchronous Idle
023	ETB	End of Transmission Block
024	CAN	Cancel
025	EM	End of Medium
026	SUB	Substitute Character
027	ESC	Escape
028	FS	File Separator
029	GS	Group Separator
030	RS	Record Separator
031	US	Unit Separator
127	DEL	Delete

Paritäts-Bit

- **Parity-Bit**
  - b_7 wird so gewählt, dass $\sum_{i=0}^{7} b_i$ (die Anzahl der Einsen) entweder immer gerade oder immer ungerade ist:
    - bei ungerader Parität (*odd parity*) ungerade
    - bei gerader Parität (*even parity*) gerade
  - Ein-Bit-Fehler bei der Datenübertragung können so erkannt werden
    - kippt ein einzelnes Bit, ändert sich die Anzahl der Einsen von gerade nach ungerade, bzw. von ungerade nach gerade

Beispiel ASCII Code

- **"Uni" im ASCII-Code darstellen**
- **ASCII-Tabelle (mit ungerader Parität)**

'U':	P101 0101	$\Rightarrow$ P=1	$\Rightarrow$ 1101 0101
'n':	P110 1110	$\Rightarrow$ P=0	$\Rightarrow$ 0110 1110
'i':	P110 1001	$\Rightarrow$ P=1	$\Rightarrow$ 1110 1001

- **Ergebnis:**

11010101 01101110 11101001

ASCII & länderspezifische Zeichen

- **länderspezifische Zeichen fehlen, z.B.**

  Ä, Ö, Ü, ä, ö, ü, ß, §, Å, æ, ¥

- **1. Lösungsansatz**
  - weniger häufig benutzte Zeichen ersetzen, z.B.

    { → ä, | → ö, ...
  - nur geringe Anzahl von Ersetzungsmöglichkeiten
- **2. Lösungsansatz (1986)**
  - Verzichten auf das Parity-Bit
  - 8. Bit für Verdopplung der Größe der Code-Tabelle nutzen

ISO 8859

- **ISO**
  - International Organization for Standardization
    - kommt von griechisch: *isos* ("gleich")
    - normiert alles außer Elektrik und Elektronik (IEC) und Telekommunikation (ITU)
- **128 zusätzliche Codes**
  - genormt in der ISO 8859
    - 15 Teilnormen, die die jeweils länderspezifischen Codes festlegen
- **Vorteil**
  - in vielen Regionen können alle länderspezifischen Zeichen dargestellt werden
- **Nachteile**
  - verschiedene Regionen interpretieren dieselben Codes unterschiedlich
  - Benutzung verschiedener Code-Tabellen in verschiedenen Regionen
  - Datenaustausch schwierig

Unicode

- **jedes erdenkliche Zeichen bekommt eindeutigen Code**
  - 17 Ebenen (planes), (5 bit Code)
    - jede Ebene kodiert bis zu 65536 Zeichen (16 bit Code)
    - Plane 0: Basic Multilingual Plane (BMP)
      - die wichtigsten Zeichen vieler Sprachen sind hier vorhanden
    - Planes 1-16: Supplemental (engl.: *ergänzend*) Planes
      - Plane 1: historische, nicht mehr genutzte Zeichen, musische und mathematische Symbole
      - Plane 2: selten benutzte chinesische Symbole
      - Planes 3-13: unbenutzt
      - Plane 14: Sonderzwecke
      - Planes 15-16: private Nutzung (eingeschränkte Interoperabilität)
  - damit ist ein Unicode-Zeichen eindeutig durch 21 bit festgelegt (5+16)
  - damit sind prinzipiell $17*2^{16} = 1114112$ (über eine Million) Zeichen kodierbar

Unicode, UTF

- **UTF (Unicode Transformation Format)**
  - Methode, um Unicode-Zeichen (21 bit) byteweise (UTF-8), doppelbyteweise (UTF-16) und vierfachbyteweise (UTF-32) zu schreiben

UTF-8:	Unicode-Bitmuster	UTF-8-Codierung
0x000000-0x00007F:	00000 00000000 0xxxxxxx	0xxxxxxx
0x000080-0x0007FF:	00000 00000yyy xxxxxxxx	110yyyxx 10xxxxxx
0x000800-0x00FFFF:	00000 yyyyyyyy xxxxxxxx	1110yyyy 10yyyyxx 10xxxxxx
0x010000-0x10FFFF:	zzzzz yyyyyyyy xxxxxxxx	11110zzz 10zzyyyy 10yyyyxx 10xxxxxx

UTF-16:	Unicode-Bitmuster	UTF-16-Codierung
0x000000-0x00FFFF:	00000 yyyyyyyy xxxxxxxx	yyyyyyyy xxxxxxxx
0x010000-0x10FFFF:	zzzzz yyyyyyyy xxxxxxxx	110110vv vvyyyyyy 110111yy xxxxxxxx

mit vvvv = zzzzz-1

UTF-32:	Unicode-Bitmuster	UTF-32-Codierung
0x000000-0x10FFFF:	zzzzz yyyyyyyy xxxxxxxx	00000000 000zzzzz yyyyyyyy xxxxxxxx

Unicode, UTF-8

- **UTF-8**
  - meistgebrauchte Kodierung für Unicode-Zeichen
  - Unicode-Zeichen werden auf 1 bis 4 Bytes abgebildet
    - höchstwertiges Bit = 0
      - Einbytezeichen: ASCII
    - höchstwertiges Bit = 1
      - Mehrbytezeichen
      - 11xx..x: Start-Byte: Anzahl führender Einsen gibt Gesamtzahl der Bytes an
      - 10xx..x: Folge-Byte
    - auch wenn man mitten im Strom der Zeichen anfängt zu lesen, findet man eindeutig den Beginn des nächsten Zeichens
  - Platz sparend, da häufig benötigte Zeichen nur wenige Bytes verbrauchen (ASCII-Zeichen bleiben bei einem einzigen Byte)
- **daraus folgt: eine ASCII-Datei ist eine gültige UTF-8 Datei**

Unicode, UTF-16, UTF-32

- **UTF-16**
  - ältestes Unicode Format
  - optimal für Zeichen der BMP (basic multilingual plane)
    - werden in 2 Byte kodiert
    - andere Zeichen benötigen 4 Byte
      - um diese Kodierung von 2 Byte BMP Zeichen unterscheiden zu können, sind in der BMP die Bereiche D800-DBFF (*high surrogates*) und DC00–DFFF (*low surrogates*) als Ersatzzeichen (engl. *surrogate*: Ersatz-) extra für diesen Zweck reserviert
      - d.h. 110110 0000000000 – 110111 1111111111 kommen nicht als echte Zeichen in der BMP vor
- **UTF-32**
  - einfachstes Format
  - benötigt am meisten Speicherplatz (4 Bytes für jedes Zeichen)
  - aus der Länge der Datei ist die Anzahl der Zeichen sofort berechenbar

Byte Order Mark, BOM

- **Bytereihenfolge-Markierung**
  - in UTF-16 und UTF-32 können die Zeichen in Big-Endian oder Little-Endian Reihenfolge kodiert werden
  - daher kann ein BOM als erstes Wort verwendet werden, um die Reihenfolge eindeutig festzulegen
  - dazu dient das Unicode Zeichen FEFF (*zero width no-break space*) der BMP, es kann als erstes Zeichen in der Unicode Datei benutzt werden
    - UTF-16 Big-Endian FE FF
    - UTF-16 Little-Endian FF FE
    - UTF-32 Big-Endian 00 00 FE FF
    - UTF-32 Little-Endian FF FE 00 00
  - in UTF-8 ist das eigentlich nicht nötig,
    - UTF-8 EF BB BF (UTF-8 Codierung von FEFF)
  - kann als Unterscheidung zu ISO-Zeichensätzen dienen
    - ISO: ï»¿ (sehr unwahrscheinlich, dass diese drei Zeichen gemeint sind!)

Hamming-Distanz

- **Definition**
  - zwei Codewörter besitzen die Hamming-Distanz *n*, genau dann, wenn sich ihre Bitmuster an *n* Stellen unterscheiden
- **Beispiel**

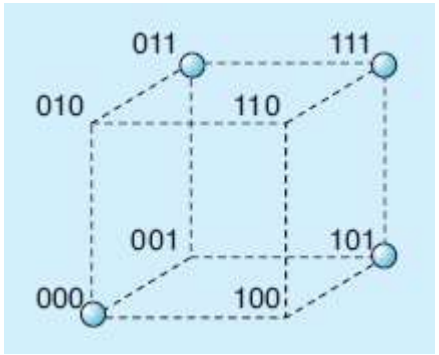
  0 0 1 0 1 0 1 1
  0 1 1 0 1 0 0 0

  - Hamming-Distanz 3
- **Code-Distanz**
  - kleinste Hamming-Distanz, die zwischen zwei Codewörtern auftritt
  - kann größer als 1 sein, wenn nicht alle 2^n möglichen *n*-Bit Codewörter benutzt werden

Fehlererkennende Codes

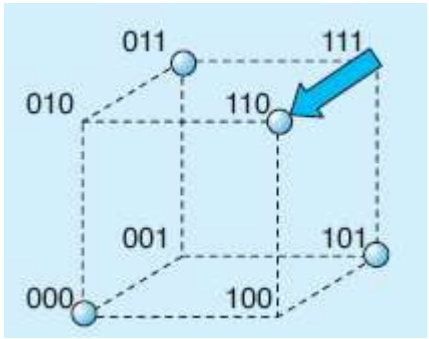
- Wenn man nur einige der 2^n möglichen Codewörter benutzt, können unter Umständen Fehler erkannt werden
- z.B. bei n=3 Codewörter: 000, 011, 101, 111
- 000 und 011 haben Hamming-Distanz 2
- wenn man nur ein Bit kippt (z.B. Fehler bei Datenübertragung), kann aus dem einen Code nicht der andere werden
- bei einer Code-Distanz d können bis zu d-1 Bitfehler erkannt werden

- **Hamming-Würfel**
  - jedes Bit spannt eine Raumdimension auf



Code-Distanz 1

besser



Code-Distanz 2

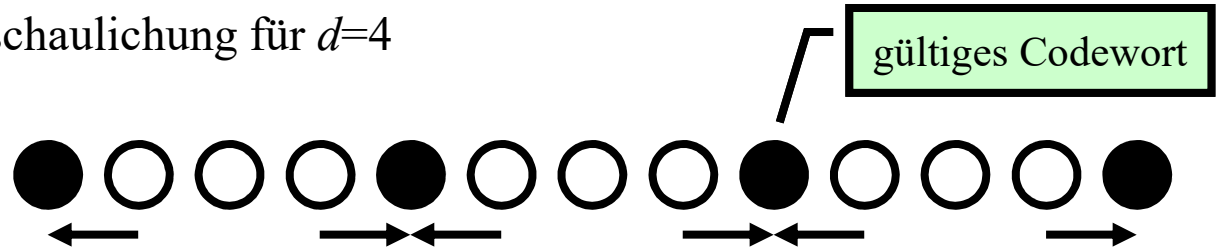
Codes haben gerade Parität

Fehlerkorrigierende Codes

- **Code mit Code-Distanz $d > 2$**
  - Fehlererkennung
    - bis zu d-1 Bitfehler können erkannt werden
  - Fehlerkorrektur
    - bis zu $\lfloor (d-1)/2 \rfloor$ Bitfehler können korrigiert werden
    - dazu wählt man das gültige Codewort, das die kleinste Hamming-Distanz zum fehlerhaften Wort hat

$\lfloor x \rfloor$ ist die größte ganze Zahl, die kleiner oder gleich x ist ("abgerundet auf ganze Zahl")

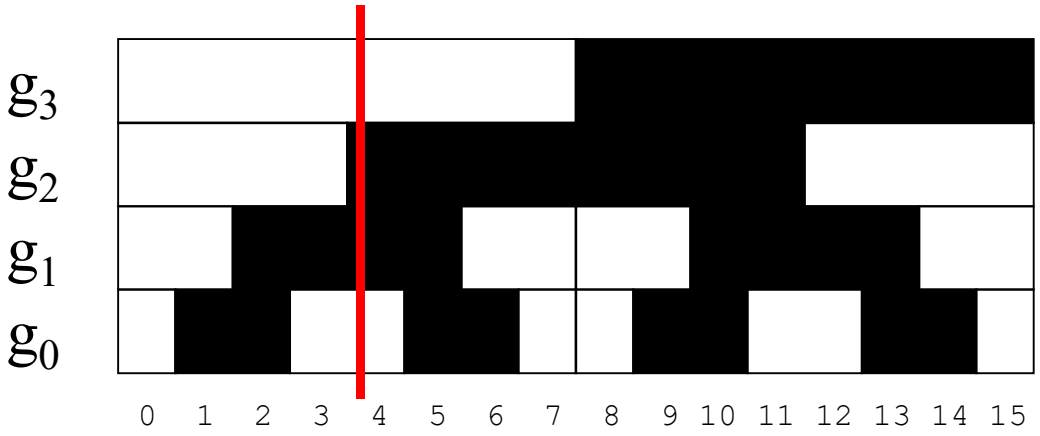
  - Veranschaulichung für d=4



    - 1-Bit Fehler können noch korrigiert werden
    - 2-Bit Fehler können noch erkannt werden
    - Verzichtet man auf die Korrektur, werden sogar 3-Bit Fehler noch sicher erkannt
  - Konstruktion von fehlerkorrigierenden Codes
    - siehe: Codierungs-Theorie

Gray-Code

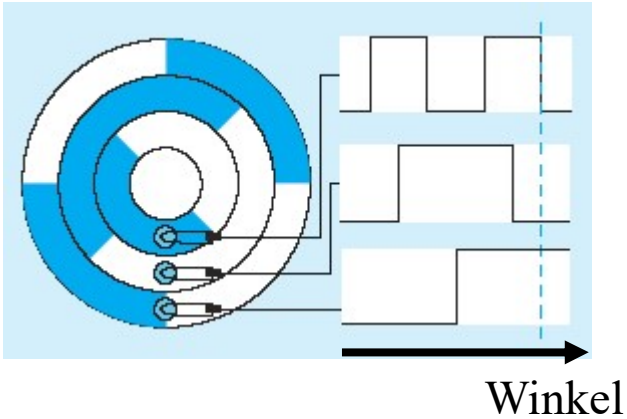
- im Gray-Code ändert sich beim Übergang von einer Zahl zur nächsten nur ein einziges Bit
- Ausnutzung in Meßsystemen, bei denen Längen oder Winkel von Code-Scheiben abgetastet werden
- Vermeidung von Fehlabtastungen am Übergangspunkt
- spielt in KV-Diagrammen eine Rolle (s.u.)



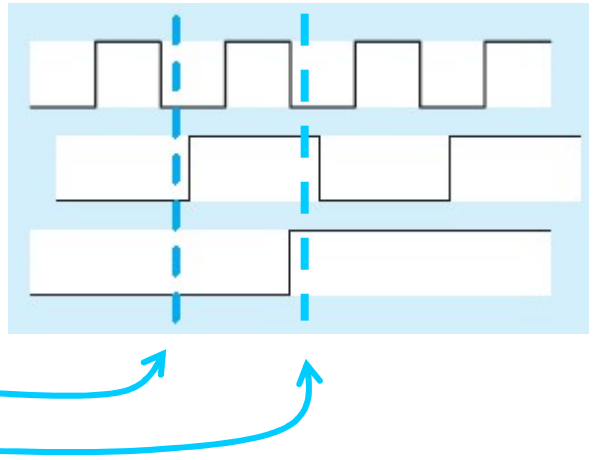
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Gray-Code (2)

- **Beispiel Winkelmessung**
  - Sensoren tasten konzentrische Code-Scheiben ab
  - beim Drehen der Scheibe ändert sich zu jedem Zeitpunkt nur die Ausgabe von höchstens einem Sensor



  - Anders wäre es bei mehrschrittigen Codes (hier normaler Binärcode)
  - man kann nicht vermeiden, dass einzelne Sensoren leicht versetzt arbeiten (hier der mittlere)
  - fatale Fehlmessungen sind daher unvermeidbar
    - zwischen 001 und 010 liegt hier 000
    - zwischen 011 und 100 liegt hier 110



1 aus *n* Code

- **1 aus *n* Codes**
  - von *n* Bits ist nur ein Bit 1, alle sind 0
  - um *n* Codes zu erzeugen braucht man *n* Bits
  - spielt bei Adressdekodierern und Multiplexern eine Rolle (s.u.)

0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

Weitere Codes

- **Codes zur Reduzierung der Datenmenge**
  - häufig verwendete Zeichen erhalten kürzere Codes, z.B.
    - Morsecode
      - Code aus drei Zeichen: Punkt, Strich, Pause
      - E: Punkt Pause
      - M: Strich Strich Pause
      - S: Punkt Punkt Punkt Pause
      - O: Strich Strich Strich Pause
      - Y: Strich Punkt Strich Strich Pause
    - Huffman-Code
      - wird anhand der bekannten Wahrscheinlichkeiten für das Auftreten der Symbole konstruiert

Weitere Codes (2)

- **Codes für bestimmte technische Anforderungen**
  - Codes mit häufigen Signalwechseln
    - Rückgewinnung des Taktsignals

```
schlecht: 000000    111111     000111
gut:      010101    010011     001010
```

  - Codes mit gleich vielen Einsen und Nullen
    - konstante mittlere Spannung oder Strom

```
schlecht: 000000    111111     000001
gut:      010101    000111     100011
```

- **Beispiel: 8b/10b Code**
  - 8-bit Codes werden auf 10-bit Codes abgebildet, so dass
    - nie mehr als 5 aufeinander folgende 1en oder 0en auftreten
    - die Differenz der Anzahl der 1en und 0en für zwei aufeinander folgende Zeichen nicht größer als 2 ist
  - benutzt in: PCI Express, SATA, Gigabit Ethernet, DVI, HDMI, …