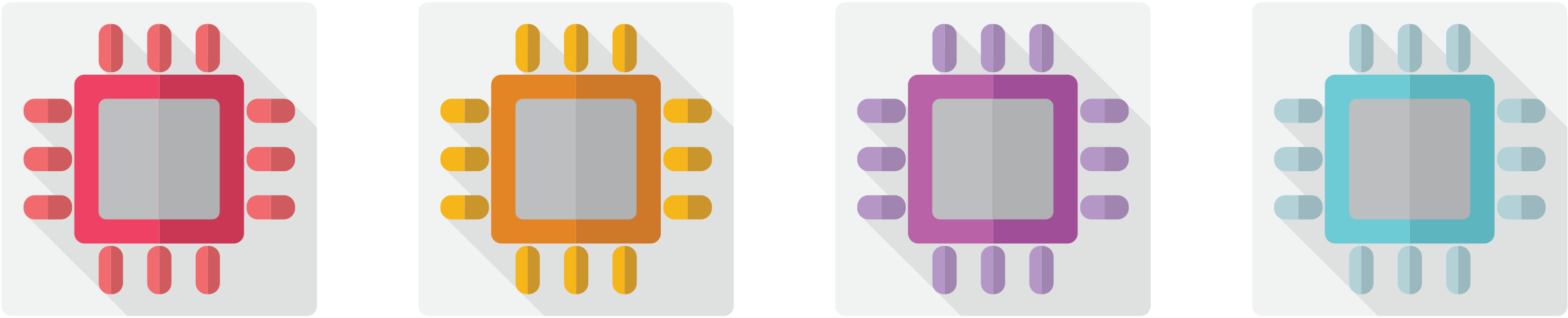
Karlsruher Institut für Technologie

Technische Informatik – SS 2024 – Digitaltechnik und Entwurfsverfahren (TI 1)

Uwe D. Hanebeck, Roman Lehmann

KIT-Fakultät für Informatik, Institut für Technische Informatik (ITEC), Forschungsgruppe für Rechnerarchitektur und Parallelverarbeitung (CAPP)



Übung 1

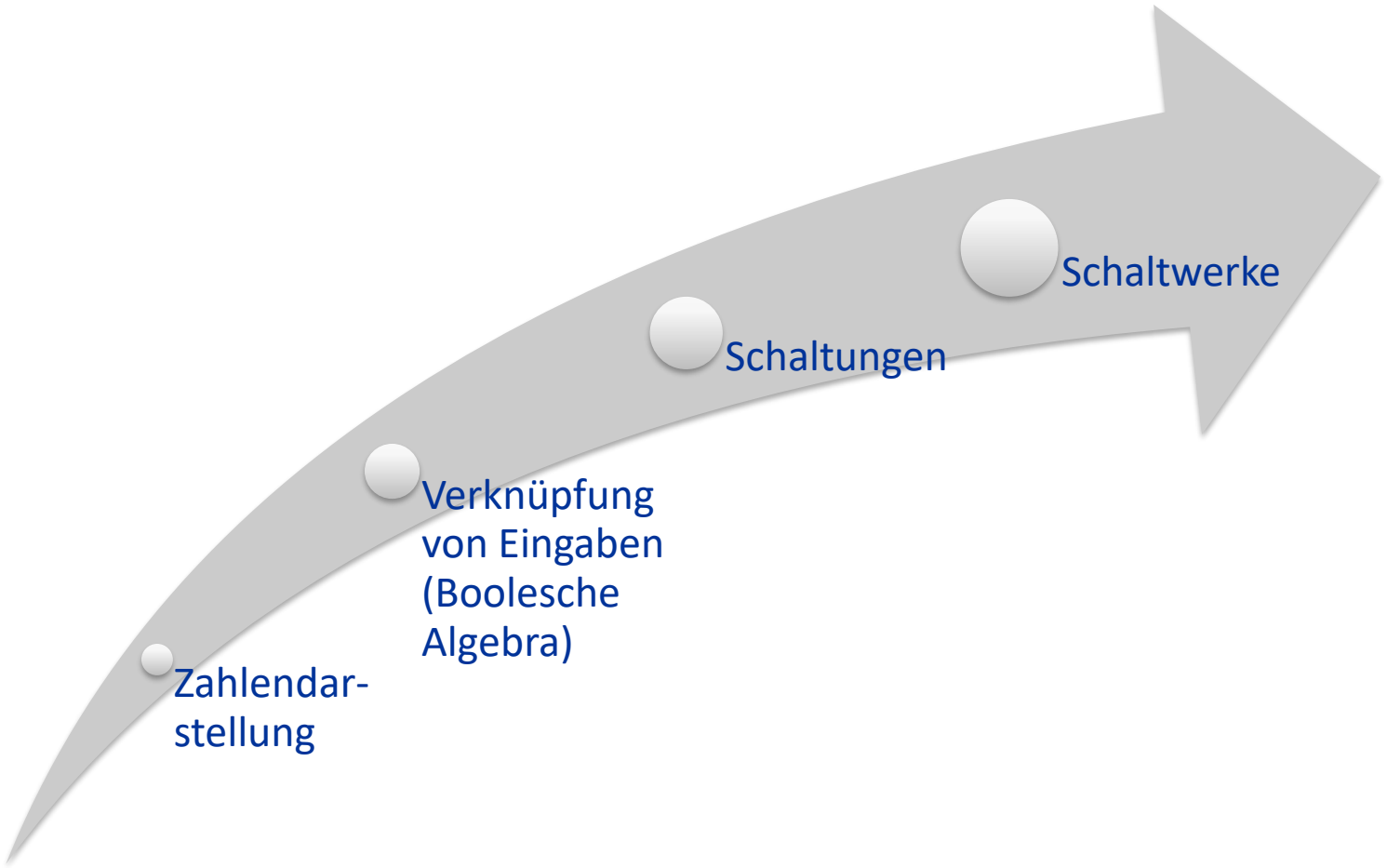
Organisatorisches

- 1. Übungsblatt ausgegeben
- Abgabe bis 06. Mai um 13.15h im Ilias in den Tutorengruppen
- Tutorienbetrieb gestartet

- 1. Mai (Mittwoch) ist Feiertag: Tutorien fallen aus
  - Betroffenen Studierende können in ein beliebiges anderes Tutorium gehen.

Aufbau der Vorlesung

- Bottom-Up Prinzip:



Inhalt 1. Übung

- Zahlendarstellungen
  - Umwandeln zwischen verschiedenen Basen (Euklid/Horner)
  - Rechnen im Zweierkomplement
  - Gleitkommazahlen
  - Addition und Multiplikation von GK-Zahlen
- Fehlerkorrektur
  - Hammingcode

Was ist 11111?

- $11111_2 = 31_{10}$
- $11111_{VZ} = -15_{10}$
- $11111_{EK} = 0_{10}$
- $11111_{ZK} = -1_{10}$
- $11111_{10} = 11.111_{10}$
- $11111_{16} = 69.905_{10}$

Euklidischer Algorithmus

Horner Schema

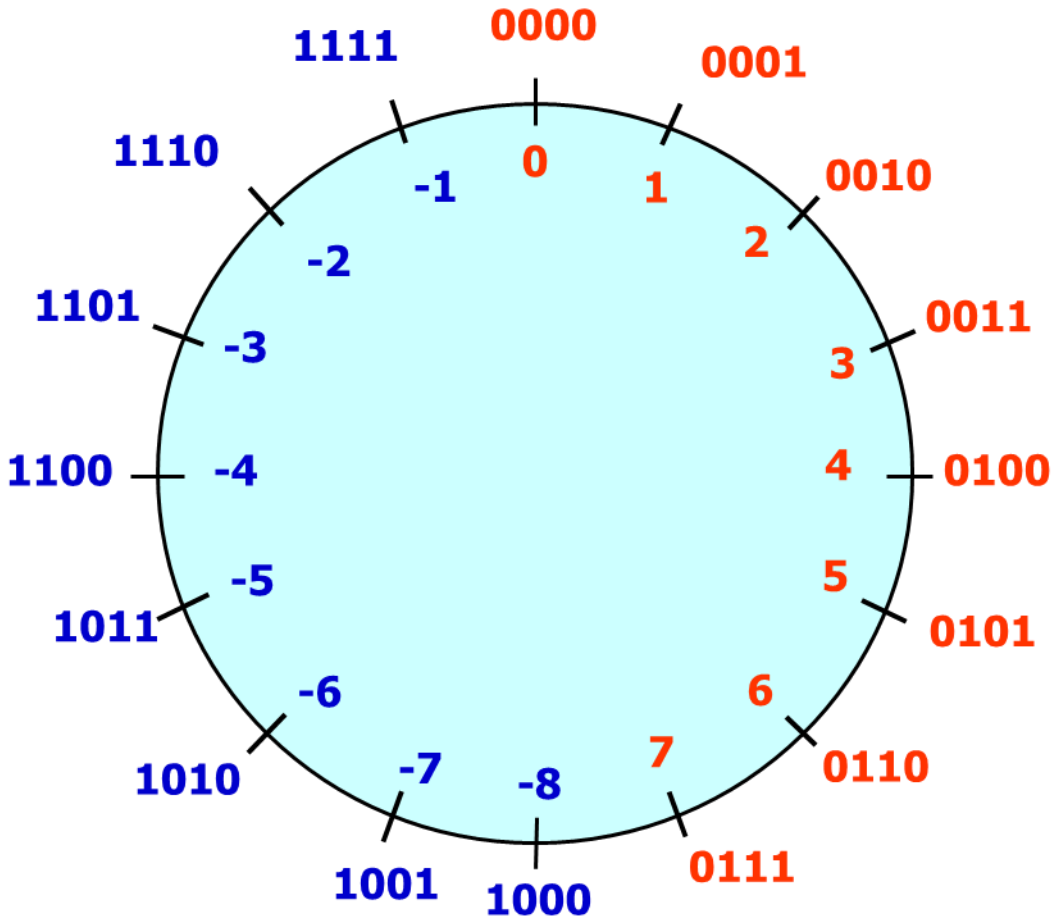
Zahlen in Zweierkomplement (ZK)

- Sei $\boldsymbol{z = b_{n-1} \ldots b_0}$ eine n-stellige Binärzahl
- Wortlänge: $\boldsymbol{n\ bit \rightarrow N = 2^n}$ verschiedene Zahlen darstellbar
- Das Zweierkomplement einer n-stelligen **negativen(!)** Zahl $\boldsymbol{z}$ ist $\boldsymbol{2^n - |z|}$
  - Darstellung einer positiven Zahl **z:** $\boldsymbol{+z = z}$
  - Darstellung einer negativen Zahl **-z:** $\boldsymbol{-z = N - z}$
  - Bespiel: $n = 4 \rightarrow N = 2^4 = 16$, Darstellung von -5?

Zahlen in Zweierkomplement (ZK)

- Es sei $n = 4$
- Gesucht: Darstellung von -14 in ZK

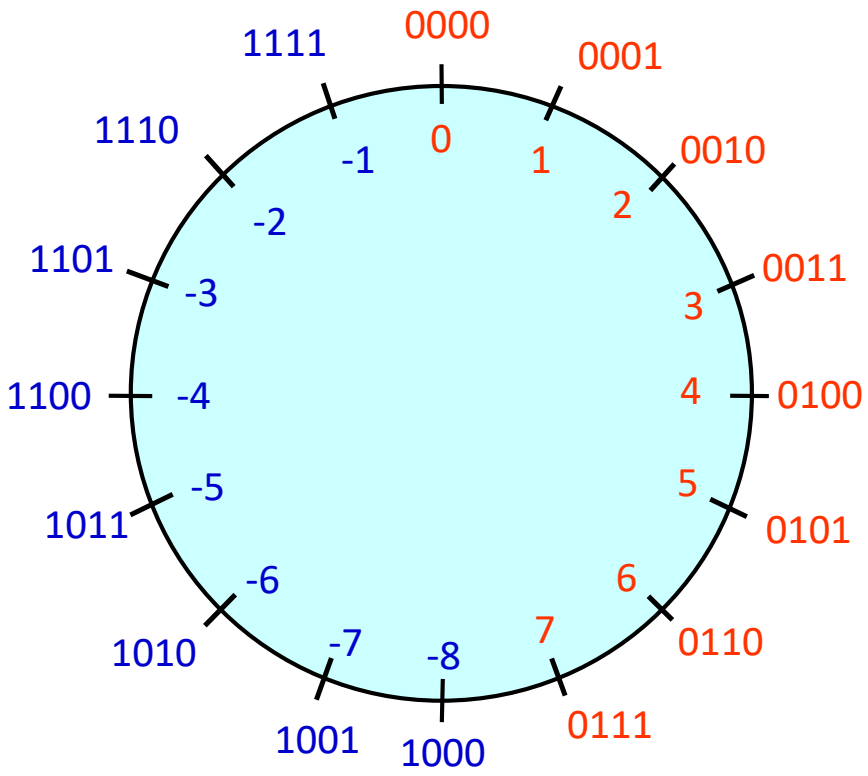
Zweierkomplement-Darstellung



Addition von ZK-Zahlen

```
  1100   - 4
+ 0010   + 2
-------------
  1110   - 2
```

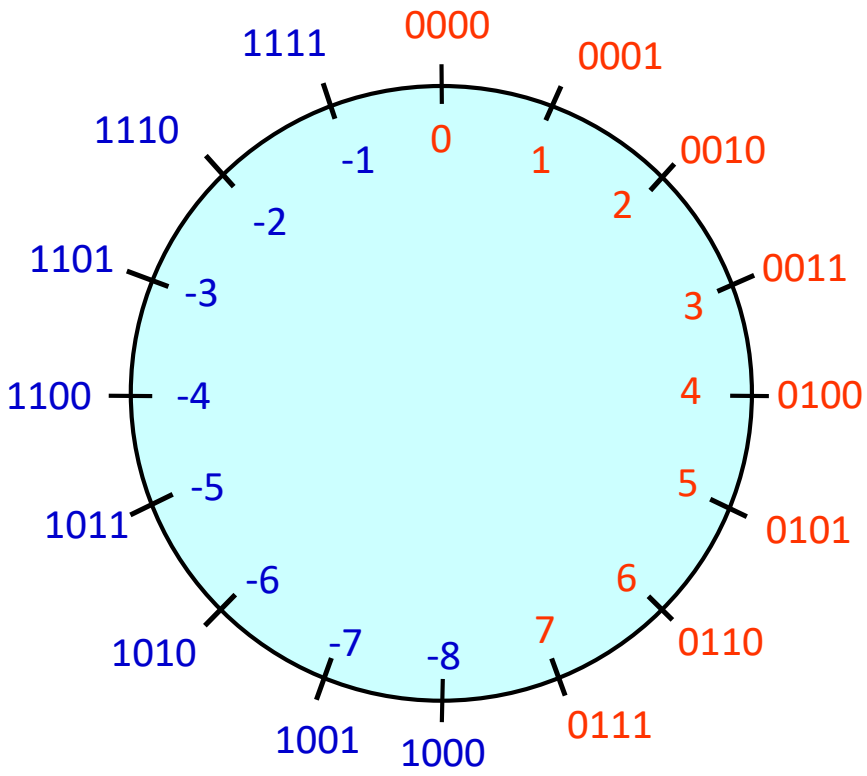
```
  1110   - 2
+ 0100   + 4
-------------
 10010   + 2
```



Subtraktion von ZK-Zahlen (als Addition)

```
  0001   + 1
+ 1100   - 4
---------------
  1101   - 3


  1111   - 1
+ 1100   - 4
---------------
 11011   - 5
```



Addition von ZK-Zahlen (1)

- Bei der Addition lassen sich 3 Fälle unterscheiden:
- **Fall 1: Beide Summanden sind positiv**
  - Die Vorzeichenbits beider Zahlen sind 0
  - Das Ergebnis muss positiv sein
  - Das Ergebnis ist nur dann korrekt, wenn sein Vorzeichenbit gleich 0 ist, ansonsten wurde der Zahlenbereich überschritten.
  - Die beiden vordersten Überträge müssen den gleichen Wert haben.
- Man kann sich die Situation anhand des Zahlenkreises klarmachen.

Beispiel 1

```
   5          0 1 0 1
+  2       +  0 0 1 0
-------     [0][0]0 0
=  7        ---------
              0 1 1 1
```

Überträge gleich
Ergebnis korrekt

```
   5          0 1 0 1
+  6       +  0 1 1 0
-------     [0][1]0 0
= 11        ---------
              1 0 1 1
```

Überträge ungleich
Überlauf
Ergebnis falsch

Addition von ZK-Zahlen (2)

- **Fall 2: Beide Summanden sind negativ**
  - Die Vorzeichenbits beider Zahlen haben den Wert 1.
  - Das Ergebnis muss negativ sein.
  - Das Ergebnis ist nur dann korrekt, wenn das Vorzeichenbit des Ergebnisses gleich 1 ist.
  - Die beiden vordersten Überträge müssen den gleichen Wert haben

Beispiel 2

$$\begin{array}{r} -5 \\ +\ (-2) \\ \hline = -7 \end{array} \qquad \begin{array}{r} 1011 \\ +\ 1110 \\ \hline \boxed{1}\boxed{1}10 \\ \hline 1001 \end{array}$$

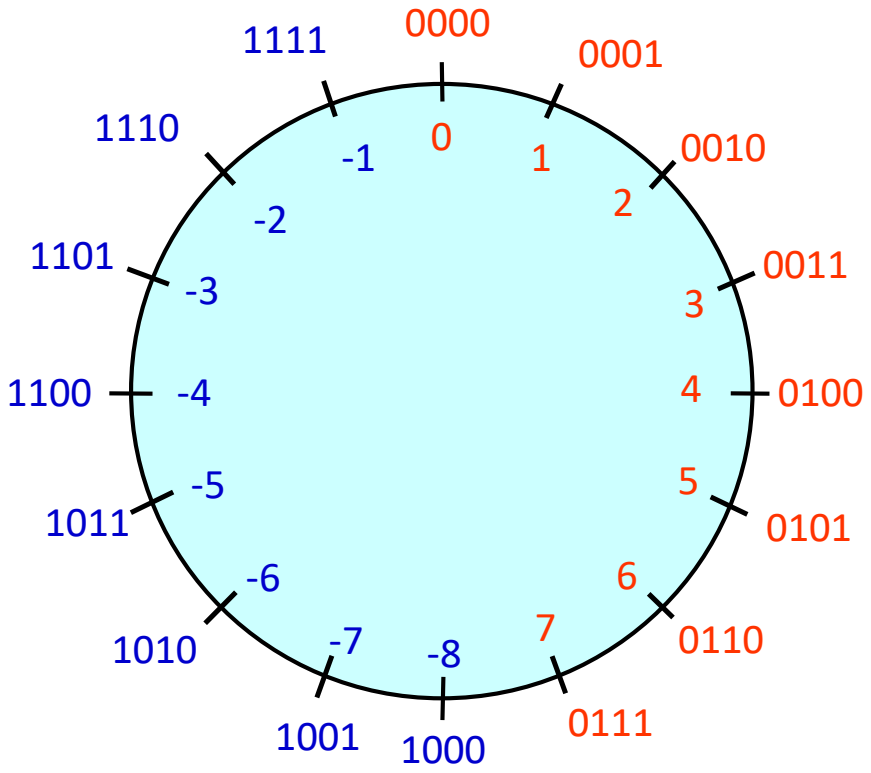
Überträge gleich
Ergebnis korrekt

$$\begin{array}{r} -5 \\ +\ (-6) \\ \hline = -11 \end{array} \qquad \begin{array}{r} 1011 \\ 1010 \\ \hline \boxed{1}\boxed{0}10 \\ \hline 0101 \end{array}$$

Überträge ungleich
Überlauf
Ergebnis falsch

Addition von ZK-Zahlen (3)

- Fall 3 Summanden haben unterschiedliche Vorzeichen:
  - Das Vorzeichen hängt davon ab, ob Subtrahend oder Minuend betragsmäßig größer ist.
  - Das Ergebnis ist auf jeden Fall korrekt.
  - Die beiden vordersten Überträge haben immer den gleichen Wert.



Beispiel 3

```
   5          0 1 0 1
+ (- 6)       1 0 1 0
-------      [0][0]0 0
= - 1        ---------
             1 1 1 1
```

Überträge gleich
Ergebnis korrekt

```
  - 5         1 0 1 1
+   6         0 1 1 0
-------      [1][1]1 0
=   1        ---------
             0 0 0 1
```

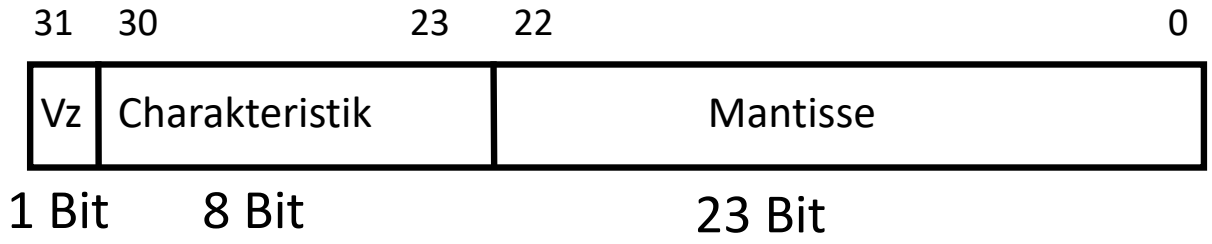
Überträge gleich
Ergebnis korrekt

Überlauferkennung

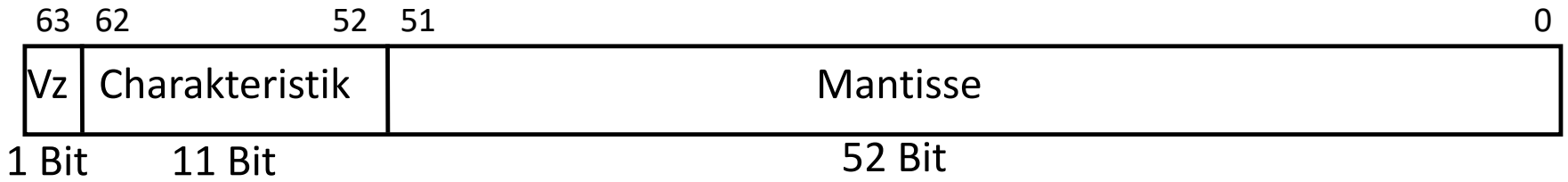
- **Allgemeine Überlauferkennung bei binärer Addition**
  - **Korrekte Addition:** Beide Überträge sind gleich.
  - **Überlauf:** Beide Überträge sind ungleich.
- Realisierung z. B. durch ein Antivalenzgatter.

IEEE-P 754-Floating-Point-Standard

32-Bit Maschinenformate des IEEE-Standards:



64-Bit Maschinenformate des IEEE-Standards:



Zusammenfassung des 32 (64)-Bit-IEEE-Formats

	Charakteristik	Zahlenwert
	0 (0)	$(-1)^{Vz} \cdot 0,Mantisse \cdot 2^{-126}$ (-1022)
Normalisierte Zahlen	1 (1)	$(-1)^{Vz}$ 1,Mantisse $\cdot 2^{-126}$ (-1022)
	...	$(-1)^{Vz}$ 1,Mantisse $\cdot 2^{Charakteristik-127}$ (-1023)
	254 (2046)	$(-1)^{Vz}$ 1,Mantisse $\cdot 2^{127}$ (1023)
	255 (2047)	Mantisse = 0: $(-1)^{Vz} \infty$ overflow
	255 (2047)	Mantisse $\neq 0$: NaN (not a number)

IEEE-P 754 Format (32 Bit)

$$Char = 1 = Exp + 127 \quad \Rightarrow \quad Exp = Char - 127 = -126 = Exp_{min}$$

$0000\ 0000\ 1000\ 0000 \cdots 0000 = (1{,}0 \cdots 00)_2 \cdot 2^{-126}$ = ***minreal***

$0000\ 0000\ 1000\ 0000 \cdots 0001 = (1{,}0 \cdots 01)_2 \cdot 2^{-126} = (1+2^{-23}) \cdot 2^{-126}$

$0000\ 0000\ 1000\ 0000 \cdots 0010 = (1{,}0 \cdots 10)_2 \cdot 2^{-126} = (1+2^{-22}) \cdot 2^{-126}$

$0000\ 0000\ 1000\ 0000 \cdots 0011 = (1{,}0 \cdots 11)_2 \cdot 2^{-126} = (1+2^{-22}+2^{-23}) \cdot 2^{-126}$

IEEE-P 754 Format (32 Bit)

$$\textit{Char} = 2 \quad \Rightarrow \quad \textit{Exp} = \textit{Char} - 127 = -125$$

$0000\ 0001\ 0000\ 0000\ \cdots\ 0000 = (1{,}0\ \cdots 00)_2 \cdot 2^{-125}$

$0000\ 0001\ 0000\ 0000\ \cdots\ 0001 = (1{,}0\ \cdots 01)_2 \cdot 2^{-125} = (1 + 2^{-23}) \cdot 2^{-125}$

IEEE-P 754 Format (32 Bit)

$$\textit{Char} = 127 \quad \Rightarrow \quad \textit{Exp} = \textit{Char} - 127 = 0$$

$$0011\ 1111\ 1000\ 0000\ \cdots\ 0000 = (1{,}0\ \cdots 00)_2 \cdot 2^0 = 1$$
$$0011\ 1111\ 1000\ 0000\ \cdots\ 0001 = (1{,}0\ \cdots 01)_2 \cdot 2^0 = 1 + 2^{-23}$$

IEEE-P 754 Format (32 Bit)

$$\textit{Char} = 254 \quad \Rightarrow \quad \textit{Exp} = \textit{Char} - 127 = 127 = \textit{Exp}_{max}$$

$$0111\ 1111\ 0000\ 0000\ \cdots\ 0000 = (1{,}0\ \cdots 00)_2 \cdot 2^{127} = 2^{127}$$

$$0111\ 1111\ 0000\ 0000\ \cdots\ 0001 = (1{,}0\ \cdots 01)_2 \cdot 2^{127} = (1 + 2^{-23}) \cdot 2^{127}$$

$$0111\ 1111\ 0111\ 1111\ \cdots\ 1111 = (1{,}11\ \cdots\ 11)_2 \cdot 2^{127}$$

$$= (1 + 2^{-1} + 2^{-2} + \cdots + 2^{-22} + 2^{-23}) \cdot 2^{127}$$

$$= (1 + (1 - 2^{-23})) \cdot 2^{127} = 2^{128} - 2^{104} = \textit{maxreal}$$

IEEE-P 754 Format (32 Bit)

***Char* = 0** $\Rightarrow$ **Spezialfall!** ***Exp* = –126**

- Darstellung von ±0

0000 0000 0000 0000 ⋯ 0000 = + 0

1000 0000 0000 0000 ⋯ 0000 = – 0

Kleinste denormalisierte Zahl

- Darstellung denormalisierter Zahlen

$0000\ 0000\ 0000\ 0000 \cdots 0001 = (0{,}0 \cdots 01)_2 \cdot 2^{-126} = 2^{-23} \cdot 2^{-126} = 2^{-149}$

$0000\ 0000\ 0000\ 0000 \cdots 0010 = (0{,}0 \cdots 10)_2 \cdot 2^{-126} = 2^{-22} \cdot 2^{-126} = 2^{-148}$

$0000\ 0000\ 0111\ 1111 \cdots 1111 = (0{,}1 \cdots 11)_2 \cdot 2^{-126}$

$= (2^{-1} + 2^{-2} + \cdots + 2^{-23}) \cdot 2^{-126}$

$= (1 - 2^{-23}) \cdot 2^{-126}$

Größte denormalisierte Zahl

IEEE-P 754 Format (32 Bit)

Char* = 255** ⇒ **Spezialfall!** ***Exp **nicht benötigt**

- Darstellung von $\pm\infty$

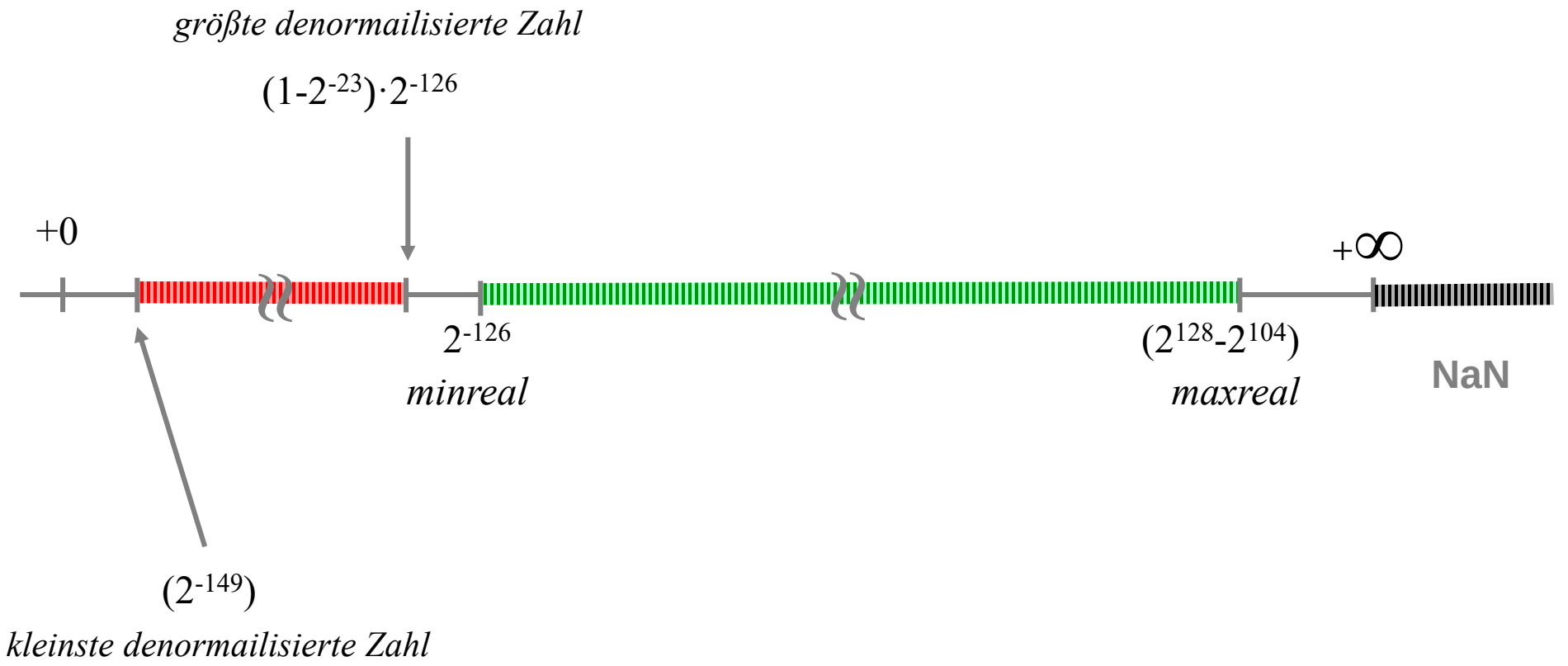
$0111\ 1111\ 1000\ 0000\ \cdots\ 0000 = +\infty$

$1111\ 1111\ 1000\ 0000\ \cdots\ 0000 = -\infty$

- Darstellung von NaN (Not a Number):

$0111\ 1111\ \underbrace{1000\ 0000\ \cdots\ 0001}_{M \neq 0}$ ⇒ **NaN**

Zusammenfassung der 32-Bit Format



Zusammenfassung des 32-Bit Formats

- Normalisierte GK-Zahlen: 254 verschiedene Exponenten
  - ➔ $2 \cdot 254 \cdot 2^{23} = 127 \cdot 2^{25}$ normalisierte Zahlen
- Denormalisierte GK-Zahlen (Char = 0)
  - ➔ $2 \cdot 2^{23} = 2^{24}$ denormalisierte Zahlen (inkl. ± 0)
- NaN (Char = 255)
  - ➔ $2 \cdot 2^{23} = 2^{24}$ NaN (inkl $\pm\infty$)

Beispielproblem

- Beispielhafte Umsetzung mit JS (JavaScript):

```
if (0.1 + 0.2 === 0.3) {
    console.log("0.1 + 0.2 equals 0.3");
} else {
    console.log("0.1 + 0.2 does not equal 0.3");
}
```

- Warum fehlerhaft:

```
console.log(0.1 + 0.2)
console.log(0.3)
```

```
0.30000000000000004
0.3
```

- Deswegen wird in der Praxis oft ein Vergleich über ein sehr kleines Epsilon gemacht.

Literatur

- IEEE Computer Society:

  IEEE Standard for Binary Floating-Point Arithmetic

  ANSI/IEEE Standard 754-1985, SIGPLAN Notices, Vol. 22, No. 2, pp 9-25, 1978

- D- Goldberg:

  What every computer scientist should know about floating point arithmetic

  ACM Computing Suveys, Vol. 13, No. 1, pp. 5-48,1991

Aufgabe 1

- Die Zahl π mit 7 Dezimalstellen hinter dem Komma im 32-Bit-IEEE-Format.

$$\pi \approx 3{,}1415926_{10} = 11{,}00100100001111110110102$$

$$= 1{,}00100100001111110110102 \cdot 2^1$$

$$= (-1)^0 \cdot 2^1 \cdot 1{,}00100100001111110110102$$

$$= (-1)^0 \cdot 2^{(128-127)} \cdot 1{,}00100100001111110110102$$

$$VZ = 0$$

$$Exp = 1 \rightarrow Char = 127 + 1 = 128 = 1000\ 00002$$

$$\pi = 0100\ 0000\ 0100\ 1001\ 0000\ 1111\ 1101\ 1010$$

$$= \quad 4 \quad 0 \quad 4 \quad 9 \quad 0 \quad F \quad D \quad A_{16}$$

Aufgabe 2

- Gegeben sei eine Gleitkomma-Zahl im 32-Bit-IEEE-Format.

  Z = 01000001111000000000000000000000

  Wie lautet die Zahl in Dezimal-Darstellung?

  0100 0001 1110 0000 0000 0000 0000 0000

  VZ = 0

  Char = 128 + 2 + 1 = 131 ➔ 131-127 = 4_{10}

  Mantisse = 110 0000 0000 0000 0000 0000$_2$

  $Z = (-1)^0 \cdot 2^4 \cdot (1{,}110\ 000\ \ldots\ 000_2)$

  $= + (1 + 0{,}5 + 0{,}25) \cdot 2^4$

  $= + 1{,}75 \cdot 16 = 28_{10}$

Rechenregeln für Gleitkommazahlen

- Es seien $x = m_x \bullet 2^{e_x}$ und $y = m_y \bullet 2^{e_y}$ zwei Gleitkommazahlen
  - Addition: $x + y = \left(m_x \bullet 2^{e_x - e_y} + m_y\right) \bullet 2^{e_y}$ falls $e_x \leq e_y$
  - Subtraktion: $x - y = \left(m_x \bullet 2^{e_x - e_y} - m_y\right) \bullet 2^{e_y}$ falls $e_x \leq e_y$
    $x - y = \left(m_x - m_y \bullet 2^{e_y - e_x}\right) \bullet 2^{e_x}$ falls $e_y \leq e_x$
  - Multiplikation: $x \bullet y = \left(m_x \bullet m_y\right) \bullet 2^{e_x + e_y}$
  - Division: $x \div y = \left(m_x \div m_y\right) \bullet 2^{e_x - e_y}$

Aufgabe 3

- Addiere die beiden GK-Zahlen (im IEEE-Standard)

  x = 0100 0011 1110 0000 … 0000

  y = 0100 0010 0101 0000 … 0000

Aufgabe 4

- Multipliziere die beiden GK-Zahlen (im IEEE-Standard)

$x = 3FE0\ 0000_{16} = 0011\ 1111\ 1110\ 0000 \ldots 0000$

$y = 3E40\ 0000_{16} = 0011\ 1110\ 0100\ 0000 \ldots 0000$

Aufgabe 5

- Gegeben seien zwei 32-Bit Gleitkommazahlen x und y im IEEE Standard

  x = 0 0000 0001 000 0000 0000 0000 0000 0001

  y = 0 0000 0001 000 0000 0000 0000 0000 0000

- Gesucht? 1/(x-y)

Lösung

- Implementierung im Rechner:
  - 1. Vorschlag: z:= 1/(x-y)
  - 2. Vorschlag: if (x != y) then z:= 1/(x-y)

x = 0 0000 0001 000 0000 0000 0000 0000 0001

y = 0 0000 0001 000 0000 0000 0000 0000 0000

Probleme bei Gleitkommaarithmetik

- **Auslöschung:** Bei der Subtraktion fast gleich großer Zahlen wird das Ergebnis aufgrund von Rundungsfehler falsch.
- **Absorption:** Die Addition bzw. Subtraktion einer betragsmäßig viel kleineren Zahl ändert die größere Zahl nicht.
- **Unterlauf:** Ist ein (Zwischen-)Ergebnis betragsmäßig keiner als minreal, wird es mit 0 approximiert. Dadurch kann das Endergebnis der Rechnung erheblich von dem erwarteten Ergebnis abweichen.
- **Ungültigkeit des Distributivgesetzes und Assoziativgesetzes**

Hammingcode

- Vom Datenwort zum Codewort
- 1- und 2-Bit Fehler

1-Bit-Fehlerkorrektur

- Notwendiger Aufwand für die sog. ***Hammingcodes***:

$$2^k \geq m + k + 1$$

k: Zahl der zusätzlichen Prüfbits

m: Zahl der Datenbits

Mindestaufwand: Ein-Bit Fehlerkorrekturcodes

Zahl der Datenbits m	Zahl der Prüfbits k	Bits insgesamt: $m + k$
1	2	$m + 2$
2 bis 4	3	$m + 3$
5 bis 11	4	$m + 4$
12 bis 26	5	$m + 5$
27 bis 57	6	$m + 6$
...	...	...

Aufbauprinzipien

- Unterschiedliche Aufbauprinzipien sind möglich

- **Hier:** Einfaches Realisierungsschema
- **Vorteil:** Beliebige Erweiterbarkeit auf größere Datenwortlängen

- Der Ansatz verwendet ***k*** verschiedene Prüfgleichungen für die Quersummen QS_0 bis QS_{k-1}

Ansatz (1)

- Die i-te Quersumme (QS_i) ergänzt alle Positionen im Codewort, die die Potenz 2^i in ihrer Ziffernwertigkeit enthalten (i=0,1,…,k-1) auf gerade Parität

- $k_1 = QS_0$ über alle ungeraden Stellen ***1, 3, 5, 7, …***
- $k_2 = QS_1$ über die Stellen ***2, 3, 6, 7, 10, 11, …***
- $k_3 = QS_2$ über die Stellen ***4, 5, 6, 7, 12, 13, 14, 15, …***
- $k_4 = QS_3$ über die Stellen ***8, 9, 10, 11, 12, 13, 14, 15, …***

0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
…

Ansatz (2)

- Es werden gerade Quersummen verwendet
- Datenwörter werden zum Einfügen der Prüfbits gespreizt, indem die Codewortposition 2^i für die i-te Quersummenprüfstelle k_{i+1} freigehalten wird
- (m_i bezeichnen die Datenbits)
- Damit erhält man folgendes Schema:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Position
	4								3				2		1	0	i
m_{12}	k_5	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1	Stellen

Ansatz (3)

- Beim Lesen eines evtl. fehlerhaften Codeworts kann durch **erneutes Bilden der Quersummen** ermittelt werden, ob und an welcher Position ein Fehler aufgetreten ist.

- Die Quersummen geben dabei, als Binärzahl interpretiert, die fehlerhafte Position im Codewort an

- Liegt kein Fehler vor, dann sind alle Prüfgleichungen erfüllt

Beispiel (1)

- Hammingcode mit 4 Datenbits und 3 Prüfbits, also 7-Bit Codewörtern

Datenwort : 1 1 0 1

	7	6	5	4	3	2	1
→	**1**	**1**	**0**	**–**	**1**	**–**	**–**
	m_4	m_3	m_2	k_3	m_1	k_2	k_1

Bestimmung der Prüfbits:

$$k_1 = m_1 \oplus m_2 \oplus m_4 = 1 \oplus 0 \oplus 1 = \mathbf{0}$$
$$k_2 = m_1 \oplus m_3 \oplus m_4 = 1 \oplus 1 \oplus 1 = \mathbf{1}$$
$$k_3 = m_2 \oplus m_3 \oplus m_4 = 0 \oplus 1 \oplus 1 = \mathbf{0}$$

➔ **Codewort :** **1 1 0 0 1 1 0**

Beispiel (2)

- Bildet man erneut die Prüfbits bei fehlerfreiem Codewort, ergibt sich als Syndrom $k_3k_2k_1$ der Wert **000**.

Codewort: **1 1 0 0 1 1 0**

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 = 0 \oplus 1 \oplus 0 \oplus 1 = \mathbf{0}$$
$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 = 1 \oplus 1 \oplus 1 \oplus 1 = \mathbf{0}$$
$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 = 0 \oplus 0 \oplus 1 \oplus 1 = \mathbf{0}$$

Codewort: **1 1 0 0 0 1 0**

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 = 0 \oplus 0 \oplus 0 \oplus 1 = \mathbf{1}$$
$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 = 1 \oplus 0 \oplus 1 \oplus 1 = \mathbf{1}$$
$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 = 0 \oplus 0 \oplus 1 \oplus 1 = \mathbf{0}$$

$k_3k_2k_1 = 011_2 = 3_{10}$ Fehler an 3. Position

Beispiel (3)

- Ist ein **Prüfbit verfälscht**, z. B. k_3, wird nur die betroffene Prüfgleichung beeinflusst, hier QS_2

	m_4	m_3	m_2	k_3	m_1	k_2	k_1
Codewort:	**1**	**1**	**0**	**1**	**1**	**1**	**0**

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 = 0 \oplus 1 \oplus 0 \oplus 1 = \mathbf{0}$$
$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 = 1 \oplus 1 \oplus 1 \oplus 1 = \mathbf{0}$$
$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 = 1 \oplus 0 \oplus 1 \oplus 1 = \mathbf{1}$$

$\boldsymbol{k_3 k_2 k_1 = 100_2 = 4_{10}}$ Fehler an 4. Position

Zusammenfassung

- Fehlerkorrekturcodes können ein fehlerfreies Codewort von einem fehlerhaften Codewort unterscheiden.
- Bei fehlerhaften Codewörtern sind die entsprechenden Bits im Codewort innerhalb bestimmter Grenzen korrigierbar.

  → Mehr als ein Prüfbit pro Datenwort notwendig

  → Fehlertoleranz durch Informationsredundanz

- Um die Fehlerkorrektur vornehmen zu können, wird in den Prüfbits eindeutig kodiert, ob und ggf. welche Bits im gesamten Codewort (Daten- und Prüfbits) fehlerhaft sind.

Hammingcode: Beispiel 2-Bitfehler

- **Fall 1:** Kein Fehler

m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1	***PB***
0	0	1	1	0	0	1	1	0	0

Gesamtparität: gerade

$k_4k_3k_2k_1 = 0000$

- **Fall 2:** 1-Bit-Fehler

0 0 1 0 0 0 1 1 0 0

Gesamtparität: ungerade

$k_4k_3k_2k_1 = 0110$

Fehler an 6. Position

- **Fall 3:** 2-Bitfehler

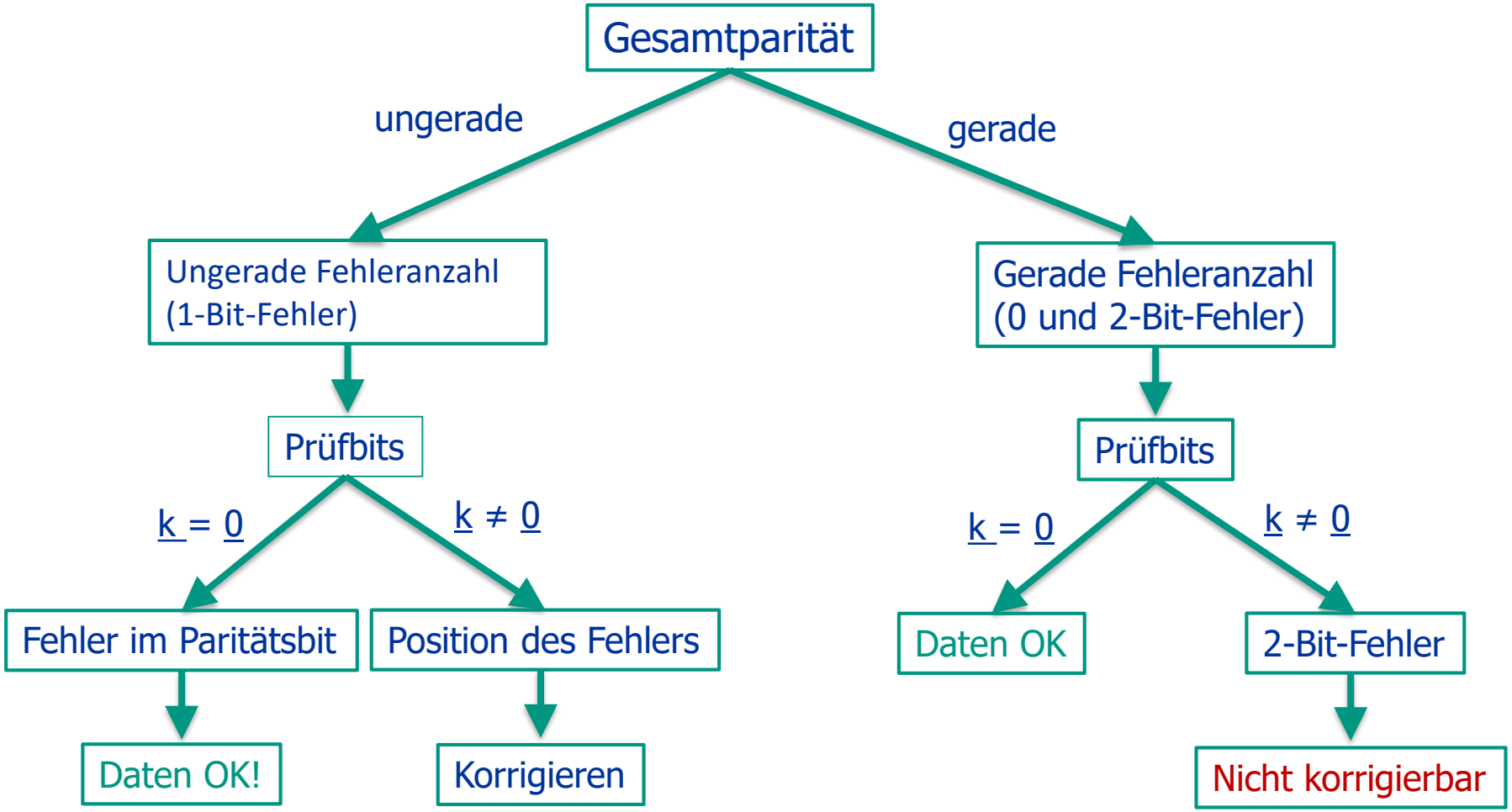
0 0 1 0 0 0 0 1 0 0

Gesamtparität: gerade

$k_4k_3k_2k_1 = 0101$

2-Bit-Fehler

Fallunterscheidungen



Hammingcode: Zusammenfassung

- An Standard-Codewörter lässt sich ein weiteres Quersummenbit anfügen, so dass Zweibitfehler im Codewort erkennbar werden.
- 32-Bit Wortlänge werden durch 7 Prüfbits zur Einzelfehlerkorrektur und Doppelfehlererkennung ergänzt.
- 64 Bit lange Datenwörter erfordern 8 Prüfbits.

Aufgabe 1

- Datenwort: 1 0 1 0 1 0 1 0

- Codewort bilden!

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Position
	4								3				2		1	0	i
m_{12}	k_5	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1	Stellen

Datenwort: 1 0 1 0 1 0 1 0

0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
...

Datenwort ausspreizen:

12	11	10	9	8	7	6	5	4	3	2	1
1	**0**	**1**	**0**	**–**	**1**	**0**	**1**	**–**	**0**	**–**	**–**
m_8	m_7	m_6	m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1

Bestimmung der Prüfbits:

$$k_1 = m_1 \oplus m_2 \oplus m_4 \oplus m_5 \oplus m_7 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = \mathbf{0}$$
$$k_2 = m_1 \oplus m_3 \oplus m_4 \oplus m_6 \oplus m_7 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = \mathbf{0}$$
$$k_3 = m_2 \oplus m_3 \oplus m_4 \oplus m_8 = 1 \oplus 0 \oplus 1 \oplus 1 = \mathbf{1}$$
$$k_4 = m_5 \oplus m_6 \oplus m_7 \oplus m_8 = 0 \oplus 1 \oplus 0 \oplus 1 = \mathbf{0}$$

➔ **Codewort: 1 0 1 0 0 1 0 1 1 0 0 0**

Aufgabe 2

- Der Hauptspeicher eines Rechners mit 8-Bit Datenwortbreite unterstützt eine Einzelbitfehler-Korrektur. Aus dem Speicher erhält man die Codewörter

  Codewort 1: `101001000110`

  Codewort 2: `100000000000`

- Prüfen Sie beide Codewörter auf Fehler, die beim Übertragen oder Speichern entstanden sein könnten und korrigieren Sie diese (falls notwendig). Geben Sie die zugehörigen Datenwörter an.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Position
	4								3				2		1	0	i
m_{12}	k_5	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1	Stellen

0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
...

Hammingcode mit 8 Datenbits und 4 Prüfbits

Codewort 1 : 1 0 1 0 0 1 0 0 0 1 1 0

Bestimmung der Prüfbits:

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 \oplus m_5 \oplus m_7 = 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = \mathbf{0}$$
$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 \oplus m_6 \oplus m_7 = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = \mathbf{0}$$
$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 \oplus m_8 = 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = \mathbf{0}$$
$$k_4 = k_4 \oplus m_5 \oplus m_6 \oplus m_7 \oplus m_8 = 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = \mathbf{0}$$

$k_4 k_3 k_2 k_1 = 0000$ ➔ Codewort ist fehlerfrei

➔ **Datenwort:** 1 0 1 0 1 0 0 1

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Position
	4								3				2		1	0	i
m_{12}	k_5	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1	Stellen

0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
...

	12	11	10	9	8	7	6	5	4	3	2	1
Codewort 2 :	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	m_8	m_7	m_6	m_5	k_4	m_4	m_3	m_2	k_3	m_1	k_2	k_1

Bestimmung der Prüfbits:

$$k_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4 \oplus m_5 \oplus m_7 = 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 = \mathbf{0}$$
$$k_2 = k_2 \oplus m_1 \oplus m_3 \oplus m_4 \oplus m_6 \oplus m_7 = 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 = \mathbf{0}$$
$$k_3 = k_3 \oplus m_2 \oplus m_3 \oplus m_4 \oplus m_8 = 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = \mathbf{1}$$
$$k_4 = k_4 \oplus m_5 \oplus m_6 \oplus m_7 \oplus m_8 = 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = \mathbf{1}$$

$k_4 k_3 k_2 k_1 = 1100 = 12_{10}$ ➔ **Fehler in der 12. Position**

➔ **Datenwort: 0 0 0 0 0 0 0 0**