# Zahlendarstellungen und Rechnerarithmetik*

1. Darstellung positiver ganzer Zahlen
2. Darstellung negativer ganzer Zahlen
3. Brüche und Festkommazahlen
4. Gleitkommazahlen
5. binäre Addition
6. binäre Subtraktion
7. binäre Multiplikation
8. binäre Division

*Die Folien wurden unter Verwendung der Folien von A. Strey,Ulm, SS 03 erstellt

# Wo liegt das Problem bei der Rechnerarithmetik?

→ **feste Länge der Zahlenrepräsentation**

- **begrenzte Genauigkeit**
- **keine Abgeschlossenheit der Operationen Addition und Multiplikation**

# Darstellung positiver ganzer Zahlen

- Darstellung positiver ganzer Zahlen in **positionaler Notation** (auch als **Stellenwertsystem** bezeichnet)
- **positive *n*-stellige ganze Dezimalzahl *x***:

$$x = (x_{n-1}\, x_{n-2}\, \ldots\, x_2\, x_1\, x_0)_{10}$$
$$= x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \ldots + x_1 \cdot 10^1 + x_0 \cdot 10^0 = \sum_{i=0}^{n-1} x_i \cdot 10^i$$
$$\text{mit } x_i \in \{0,1,2,3,4,5,6,7,8,9\}$$

Beispiel: $259_{10} = 2 \cdot 10^2 + 5 \cdot 10^1 + 9 \cdot 10^0$

- **positive *n*-stellige ganze Binär- oder Dualzahl *y***:

$$y = (y_{n-1}\, y_{n-2}\, \ldots\, y_2\, y_1\, y_0)_2$$
$$= y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \ldots + y_2 \cdot 2^2 + y_1 \cdot 2^1 + y_0 \cdot 2^0 = \sum_{i=0}^{n-1} y_i \cdot 2^i$$
$$\text{mit } y_i \in \{0,1\}$$

Beispiel: $11101_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 29_{10}$

# Darstellung positiver ganzer Zahlen

- allgemein: ***b*-adisches Zahlensystem**

  Jede natürliche Zahl $z$ mit $0 \le z \le b^n - 1$ ist **eindeutig** als $n$-stellige Zahl zur **Basis *b*** darstellbar:

$$z = (z_{n-1}\, z_{n-2} \ldots z_2\, z_1\, z_0)_b$$

$$= z_{n-1} \cdot b^{n-1} + z_{n-2} \cdot b^{n-2} + \ldots + z_2 \cdot b^2 + z_1 \cdot b^1 + z_0 \cdot b^0 = \sum_{i=0}^{n-1} z_i \cdot b^i$$

  mit **Ziffer** $z_i \in \{0,1,2, \ldots, b-1\}$

- Typische Werte für Basis *b*:

  $b = 2$: Dualzahl

  $b = 8$: Oktalzahl $z_i \in \{0,1,2, \ldots ,7\}$

  $b = 10$: Dezimalzahl

  $b = 16$: Hexadezimalzahl mit $z_i \in \{0,1,2, \ldots ,9,A,B,C,D,E,F\}$

Beispiel: $FE01_{16} = 15 \cdot 16^3 + 14 \cdot 16^2 + 1 \cdot 16^0 = 65040_{10}$

# Darstellung positiver ganzer Zahlen

- Algorithmus zur Umwandlung einer ganzen Zahl *z* aus dem Dezimalsystem in eine Zahl *x* zur Basis *b*:

```
i=0
wiederhole, bis z=0:
    berechne z=z/b          (ganzzahlige Division mit Rest)
    notiere Rest r_i
    i=i+1
```

- Reste stellen das gesuchte Ergebnis dar: $x = (r_{i-1} \ldots r_1 \; r_0)_b$
- Beispiel: Umwandlung von $z = 29_{10}$ in eine Binärzahl $x$

```
29/2 = 14,   Rest r0 = 1
14/2 = 7,    Rest r1 = 0
7/2 = 3,     Rest r2 = 1
3/2 = 1,     Rest r3 = 1
1/2 = 0,     Rest r4 = 1
```

Ergebnis: $x = (r_4 \; r_3 \; r_2 \; r_1 \; r_0)_2 = (1 \; 1 \; 1 \; 0 \; 1)_2 = 11101_2$

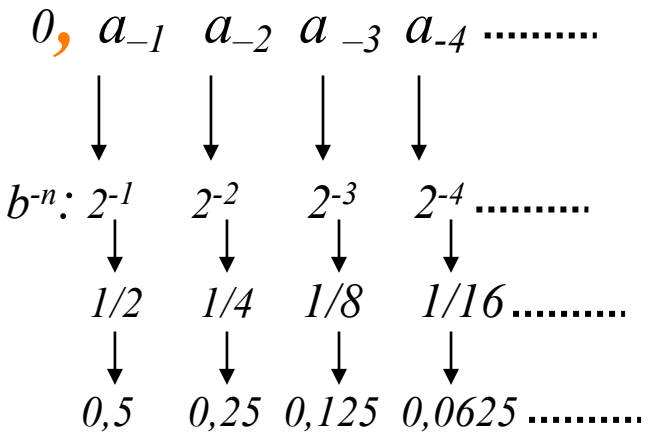
# Darstellung von Brüchen

$$a = a_{n-1}\ a_{n-2} \ldots\ldots a_2\ a_1\ a_0\ ,\ a_{-1}\ a_{-2} \ldots\ldots a_{-m+1}\ a_{-m}$$

$$a = a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \ldots + a_1 \cdot b^1 + a_0 \cdot b^0 + a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + \ldots + a_{-m} \cdot b^{-m}$$

Beispiel: $51{,}25 = 5 \cdot 10^1 + 1 \cdot 10^0 + 2 \cdot 10^{-1} + 5 \cdot 10^{-2}$

Stellenwertigkeit im Binärsystem:




Beispiel Binär → Dezimal:

$1011{,}1101 = 11 + 1/2 + 1/4 + 1/16 = 11\ 13/16$

$= 11 + 0{,}5 + 0{,}25 + 0{,}0625 = 11{,}8125$

# Darstellung von Brüchen

Umwandlung Dezimal → Binär

- Bruchteil wird getrennt konvertiert
- Der Bruchteils wird mit 2 multipliziert.
  Falls das Ergebnis > *1* ist, wird eine *1* notiert und man fährt mit dem Restbruch fort.
  Das Verfahren bricht ab, wenn eine *0* erreicht wird oder eine ausreichende Genauigkeit.

Beispiel 1: $a = 0{,}75$

$0{,}75 \cdot 2 = 1{,}5 \quad : \; 1 + \; (\text{Rest} = 0{,}5)$

$0{,}5 \quad \cdot 2 = 1{,}0 \quad : \; 1 + \; (\text{Rest} = 0)$

$0{,}75_{10} = 0{,}11_2$

Beispiel 2: $a = 0{,}125$

$0{,}125 \cdot 2 = 0{,}25 \quad : \quad 0 \; + (\text{Rest} = 0{,}25)$

$0{,}25 \quad \cdot 2 = 0{,}5 \quad : \; 0 + (\text{Rest} = 0{,}5)$

$0{,}5 \quad \cdot 2 = 1{,}0 \quad : \quad 1 + \; (\text{Rest} = 0)$

$0{,}125_{10} = 0{,}001_2$

# Darstellung von Brüchen

Beispiel: Konversionsfehler $0{,}1_{10} \rightarrow X_2$

$$
\begin{aligned}
0{,}1 \cdot 2 &= 0{,}2 \rightarrow 0 \\
0{,}2 \cdot 2 &= 0{,}4 \rightarrow 0 \\
0{,}4 \cdot 2 &= 0{,}8 \rightarrow 0 \\
0{,}8 \cdot 2 &= 1{,}6 \rightarrow 1 \\
0{,}6 \cdot 2 &= 1{,}2 \rightarrow 1 \\
0{,}2 \cdot 2 &= 0{,}4 \rightarrow 0 \\
0{,}4 \cdot 2 &= 0{,}8 \rightarrow 0 \\
0{,}8 \cdot 2 &= 1{,}6 \rightarrow 1 \\
0{,}6 \cdot 2 &= 1{,}2 \rightarrow 1 \\
0{,}2 \cdot 2 &= 0{,}4 \rightarrow 0 \\
&\vdots
\end{aligned}
$$

$$0{,}1_{10} = 0{,}0001100110011\ldots\ldots_2$$

→ keine exakte Repräsentation

# Darstellung von Brüchen

Weitere Umwandlungen:

Binär → Octal : $0{,}1011011 = 0,\ 101\ 101\ 1(00) = 0{,}554_8$

Binär → Hexadezimal: $0{,}1011011 = 0,\ 1011\ 011(0) = 0{,}B6_{16}$

Octal → Binär: $0{,}7421_8 = 0,\ 111\ 100\ 010\ 001$

Hex → Binär: $0{,}F11_{16} = 0,\ 1111\ 0001\ 0001$

# Festkommazahlen

ZIEL: Vermeidung von spezieller Behandlung von Brüchen

- die feste Kommaposition k kennt nur der Anwender, der Rechner arbeitet mit skalierten ganzen Binärzahlen $z' = z \cdot 2k$
- Beispiel: ein 8-Bit Register enthält die Binärzahl $z' = 01101110$; es gelte $k = 3 \Rightarrow z = 01101{,}110_2 = 2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-2} = 13{,}75_{10}$
- alle betrachteten Rechenwerke sind somit auch für Operationen auf Festkommazahlen geeignet!

# Darstellung positiver und negativer Zahlen

- Sollen **positive** und **negative** Werte $z$ als **Zahlen zur Basis $b$** in $n$ Stellen kodiert werden, so stellt die höchstwertige Ziffer $z_{n-1}$ das Vorzeichen dar (positiv bei $z_{n-1} = 0$, **negativ** bei $z_{n-1} = b-1$)
- für positive Zahl gilt stets: $z = (\mathbf{0}\ z_{n-2}\ z_{n-3}\ \ldots\ z_1\ z_0)_b$
- zur Kodierung negativer Zahlen gibt es drei Möglichkeiten (jeweils mit $z_i = b-1-z_i$):
-

  A) **Vorzeichen und Betrag**: $-z := (b-1\ z_{n-2}\ z_{n-3}\ \ldots\ z_1\ z_0)_b$

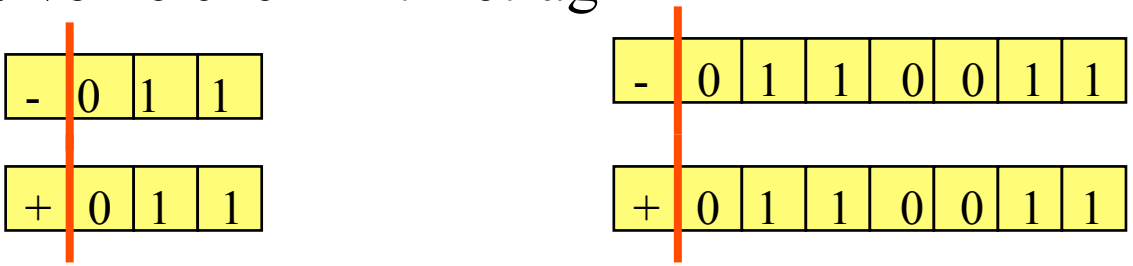
  B) **(b – 1)-Komplement**: $-z := (b^n-1) - z = (b-1\ z_{n-2}\ z_{n-3}\ \ldots\ z_1\ z_0)_b$

  C) **b-Komplement**: $-z := b^n - z = (b-1\ z_{n-2}\ z_{n-3}\ \ldots\ z_1\ z_0)_b + 1$

# Darstellung positiver und negativer Zahlen

Darstellung von Vorzeichen und Betrag:

1. Vorzeichen mit Betrag




2. Komplementäre Arithmetik
   hier ist die Eigenschaft positiv oder negativ in der Zahlendarstellung enthalten.

   Vorteil: keine gesonderten arithmetischen Einheiten für Addition und Subtraktion.

# Komplementäre Zahlendarstellung

Def.:Gegeben sei eine n-stellige Zahl *N* zur Basis b. Dann ist ihr b-Komplement gegeben durch: $\mathbf{b^n - N}$

**Beispiel: Das 10er (b-) Komplement**:
Gegeben die 4-stellige Zahl **1234**
Das 10er Kompl. ist $\mathbf{10^4 = 10000 - 1234 = 8766}$ $\mathbf{= 9999 - 1234 + 1}$

Def.:Gegeben sei eine n-stellige Zahl *N* zur Basis b. Dann ist ihr (b-1) - Komplement gegeben durch: $\mathbf{b^n - 1 - N}$

**Beispiel: Das 9er (b-1) - Komplement:**
Das (b-1)-Komplement kann **für jede Ziffer einzeln** gebildet werden, indem man die Differenz zur Ziffer 9 bildet.

```
  9999
- 1234
------
  8765
```

Das 10-Komplement kann gebildet werden, indem man das ziffernweise Komplement zu 9 bildet und "1" addiert!

# Komplementäre Zahlendarstellung

Beispiel:

$A = X - Y$

$A = 7432 - 1234$
$= 6198$

Das Komplement von Y wurde berechnet als $10^4 - Y$

$10000 - 1234 = 8766$

Man erhält:

$A = X + (10^4 - Y)$
$= 10^4 + (X - Y)$

$A = 7432 + 8766$
$= 16198$

Man erhält also das gewünschte Resultat, allerdings mit einer unerwünschten Ziffer in der höchstwertigen Position.

# Komplementäre Zahlendarstellung

- für Binärzahlen (d.h. für eine Basis $b = 2$) ergibt sich:

Das b-Komplement wird **Zweierkomplement** genannt;

Das 2er-Komplement wird erzeugt durch:

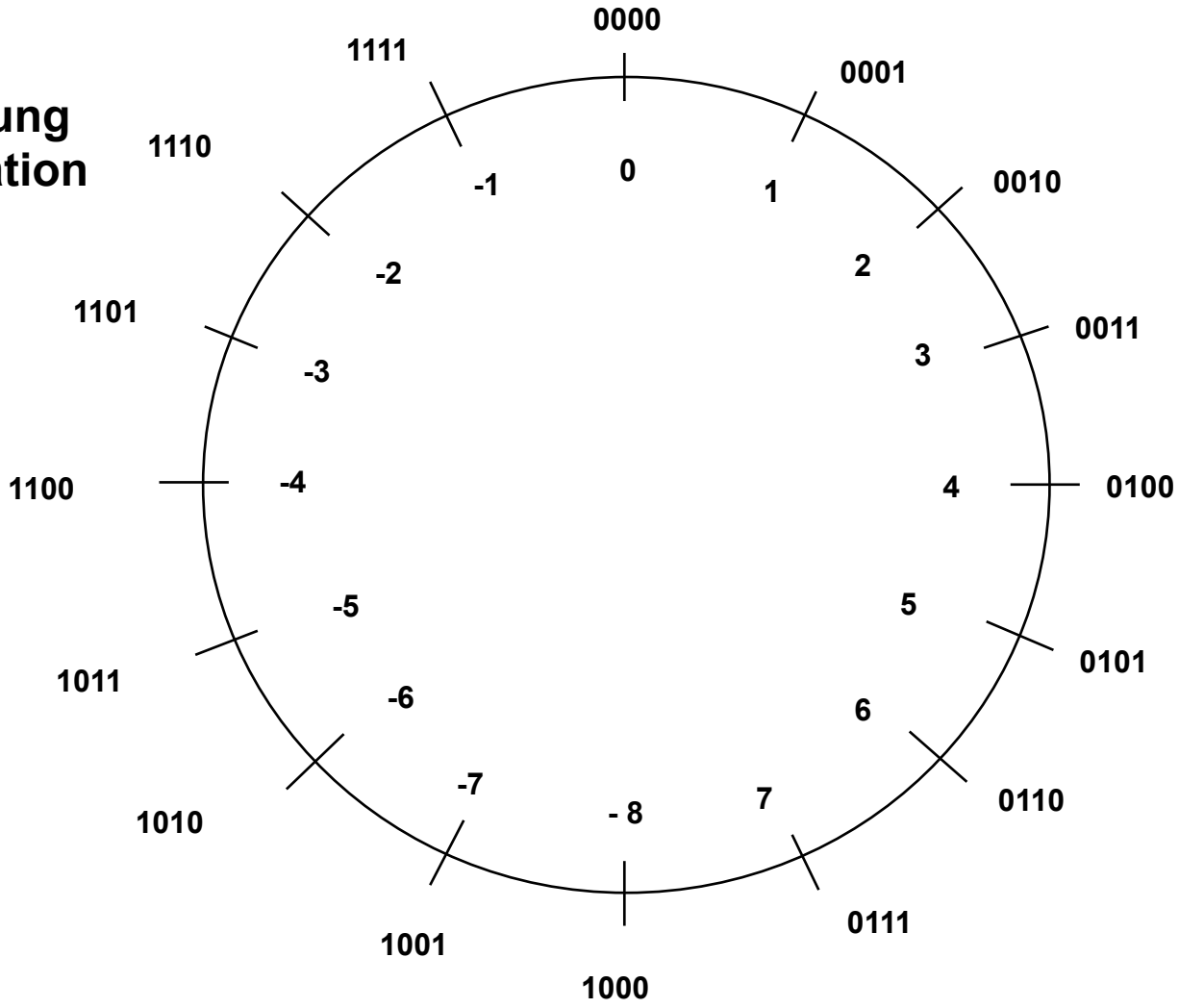
1. bitweises Komplementieren aller Stellen einer Binärzahl,
2. Addition einer „1“. (Konversionskorrektur)

Es gilt: $-z := 2^n - z$

Beispiel: $z = 00101101_2 = 45_{10} \Rightarrow -z = \mathbf{11010011}_2 = -45_{10}$

# Komplementäre Zahlendarstellung

**Der Zahlenkreis der 2er-Komplementdarstellung für eine 4-Bit Repräsentation**



# Komplementäre Zahlendarstellung

- darstellbarerer Zahlenbereich für im **Zweierkomplement** kodierte *n*-Bit Zahlen *z*: $-2^{n-1} \leq z \leq 2^{n-1} - 1$

- Beispiel: für $n = 8$ ist
  - kleinste negative darstellbare Zahl: $10000000_2 = -128_{10}$
  - größte negative darstellbare Zahl: $11111111_2 = -1_{10}$
  - größte positive darstellbare Zahl: $01111111_2 = 127_{10}$

  $\Rightarrow$ es können alle Zahlen *z* mit $-128 \leq z \leq 127$ dargestellt werden

- **Nachteile** des Zweierkomplements:
  - darstellbarer Zahlenbereich ist **asymmetrisch** (Zweierkomplement der kleinsten negativen Zahl (-128 im obigen Beispiel) ist nicht darstellbar!)
  - Umwandlung von positiver zu negativer Zahl und umgekehrt erfordert die Invertierung aller Bits sowie ein **Addierwerk** zur Addition von 1

# Komplementäre Zahlendarstellung

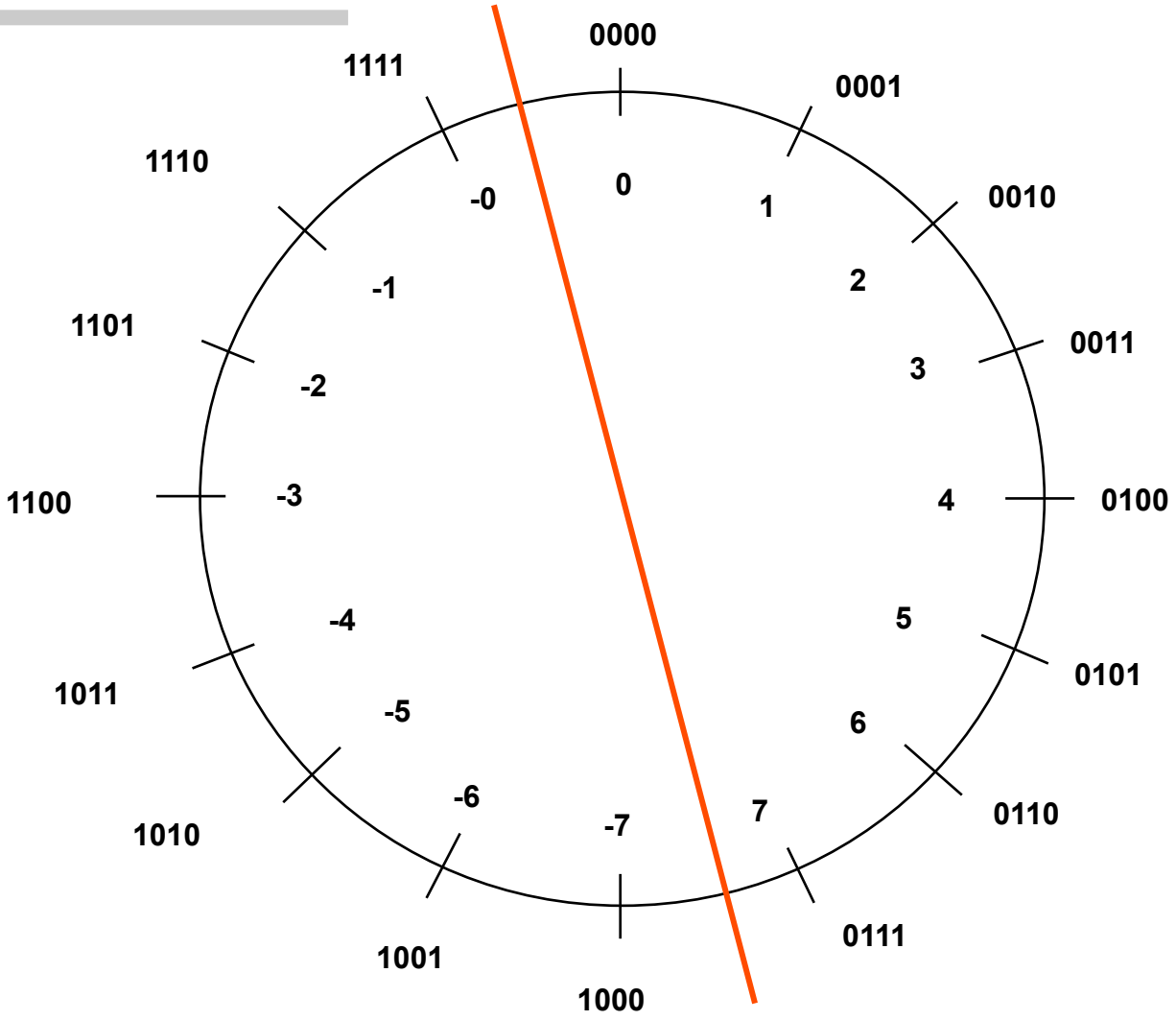
## Das 2-1 Komplement
(oder 1-Komplement)

Das (b–1)-Komplement wird **Einerkomplement** genannt; die Kodierung von $-z$ erhält man, indem man alle Bitstellen einer positiven $n$-Bit Zahl $z$ invertiert.

Es gilt: $-z := (2^n - 1) - z$

Beispiel: $z = 00101101_2 = 45_{10}$

$\Rightarrow -z = \mathbf{11010010}_2 = -45_{10}$



# Komplementäre Zahlendarstellung

- darstellbarer Zahlenbereich für im **Einerkomplement** kodierte $n$-Bit Zahlen $z$: $-2^{n-1} + 1 \leq z \leq 2^{n-1} - 1$

- Beispiel: für $n = 8$ ist
  - kleinste negative darstellbare Zahl: $10000000_2 = -127_{10}$
  - größte negative darstellbare Zahl: $11111110_2 = -1_{10}$
  - größte positive darstellbare Zahl: $01111111_2 = 127_{10}$

  $\Rightarrow$ es können alle Zahlen $z$ mit $-127 \leq z \leq 127$ dargestellt werden

- **Vorteile** des Einerkomplements:
  - der darstellbare Zahlenbereich ist **symmetrisch** zu 0
  - sehr einfache Umwandlung von positiver zu negativer Zahl und umgekehrt durch Invertierung aller Bits

# Komplementäre Zahlendarstellung

## Diskussion und Bewertung

- **Vorteile** des Einerkomplements:
  - –der darstellbare Zahlenbereich ist **symmetrisch** zu 0
  - –sehr einfache Umwandlung von positiver zu negativer Zahl und umgekehrt durch Invertierung aller Bits
- **Nachteile** des Einerkomplements:
  - –Null hat zwei Darstellungen: **$0000...0_2$** und **$1111...1_2$**
  - –Addierwerke sind aufwendig, da bei Auftreten negativer Zahlen die Summe korrigiert werden muß
- **Vorteile** des Zweierkomplements:
  - –eindeutige Darstellung der Null als **000...0**
  - –einfache Realisierung der Addition auch bei Auftreten negativer Zahlen ohne zusätzlichen Aufwand:
- **Nachteile** des Zweierkomplements:
  - –darstellbarer Zahlenbereich ist **asymmetrisch** (Zweierkomplement der kleinsten negativen Zahl ist nicht darstellbar!)
  - –Umwandlung von positiver zu negativer Zahl und umgekehrt erfordert die Invertierung aller Bits sowie ein **Addierwerk** zur Addition von 1

# Komplementäre Zahlendarstellung

## Diskussion und Bewertung

**In heutigen CPUs wird ausschließlich das 2er Komplement verwendet!**

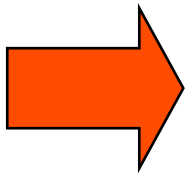
# Komplementäre Zahlendarstellung

Bei beliebig großen Registern zur Aufnahme der Komplementdarstellung einer binären Zahl können Addition und Subtraktion ohne Einschränkungen ausgeführt werden.

ABER:

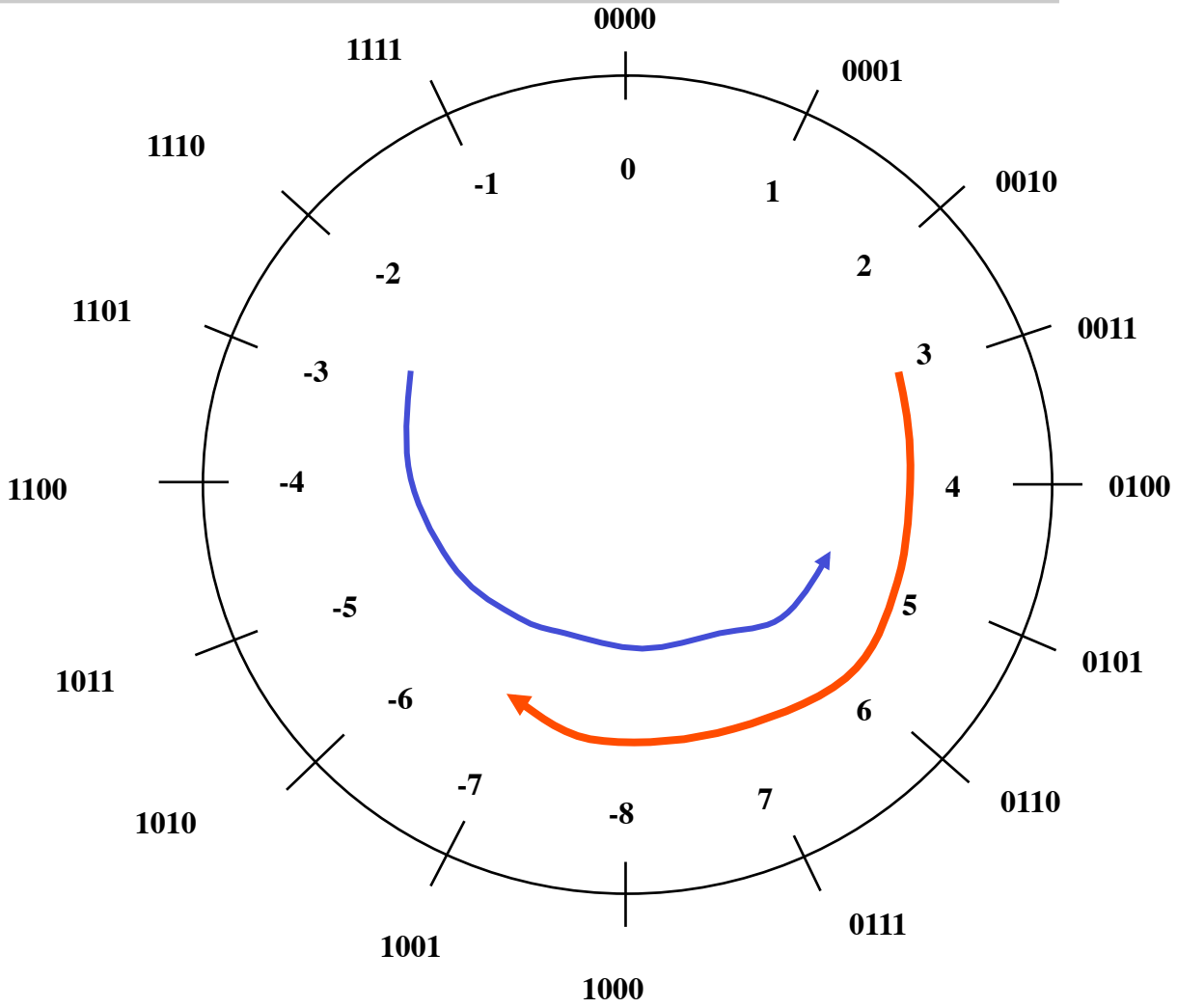
Register sind beschränkt !

Addition zweier positiver Zahlen kann eine negative Zahl ergeben !
Addition zweier negativer Zahlen kann eine positive Zahle ergeben !



**Überschreitungen des Zahlenbereichs müssen erkannt und behandelt werden!**

# Komplementäre Zahlendarstellung




3 + 6 = -7 ??

| | zusätzl. Stelle: |
|---|---|
| 0011 | 00011 |
| 0110 | 00110 |
| 1001 | 01001 = 9 |

-3 + (- 8) = +5 ??

| | zusätzl. Stelle: |
|---|---|
| 1101 | 11101 |
| 1000 | 11000 |
| 0101 | 10101 = - 11 |

# Komplementäre Zahlendarstellung

## Übertrag (Carry) und Überlauf (Overflow)

Übertrag: Übertrag in die nächste Stelle bei einer arithm. Operation
Überlauf: Überschreitung des Zahlenbereichs.

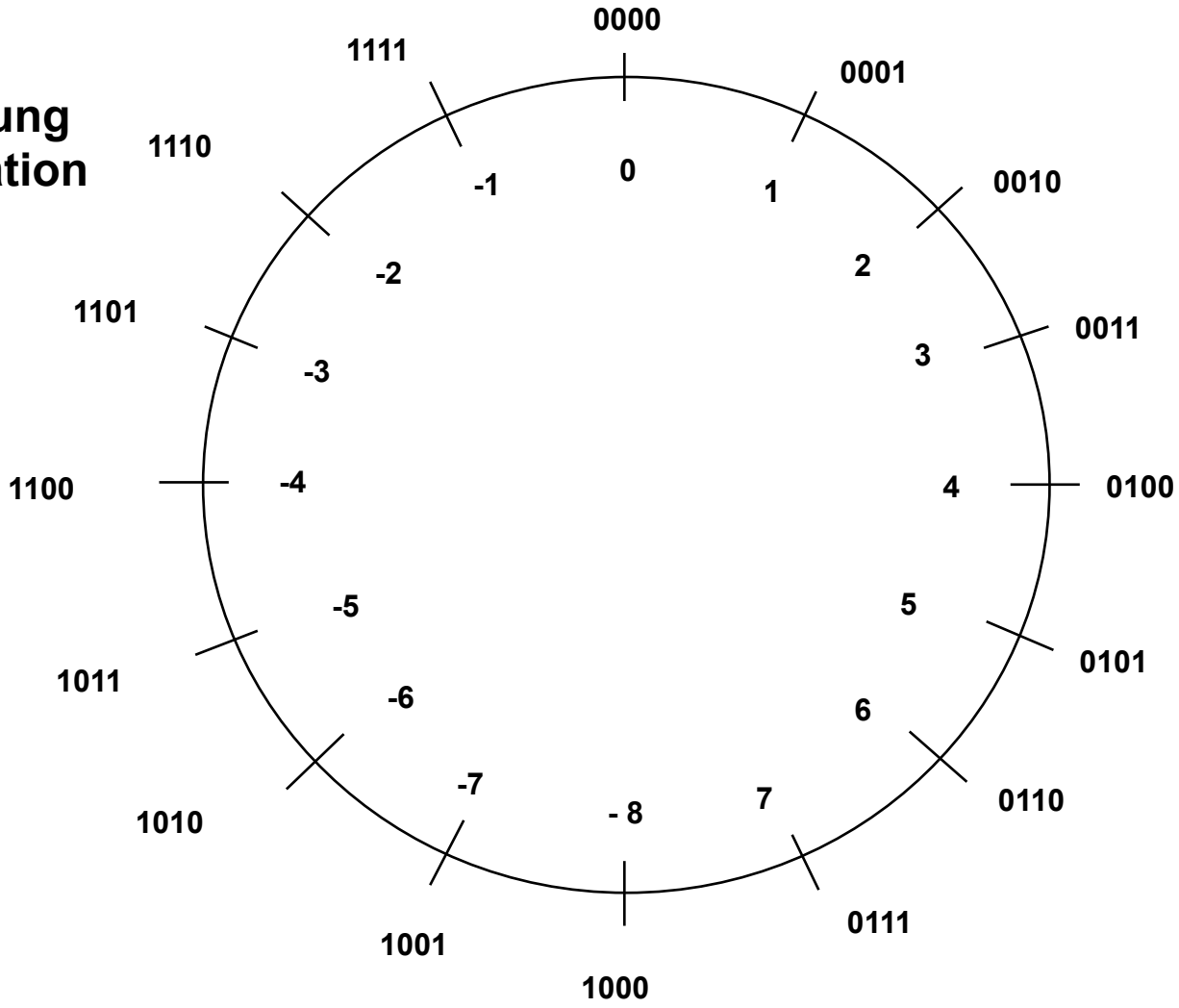
1. **Der Überlauf hat nur eine Bedeutung in Berechnungen, die auf der Komplementdarstellung beruhen.**
2. **Ein Überlauf tritt auf, wenn durch die binäre Addition der Zahlenbereich überschritten wird.**

**Die Bedingungen dafür sind: Gegeben die Operanden *a* und *b* und das Ergebnis *s***

**a.) ( a > 0 ) UND ( b > 0 ) UND (s < 0 ) oder**
**b.) ( a < 0 ) UND ( b < 0 ) UND (s > 0 )**

# Komplementäre Zahlendarstellung

**Der Zahlenkreis der 2er-Komplementdarstellung für eine 4-Bit Repräsentation**



# Gleit komma zahlen

# Gleitkommazahlen

- in vielen technischen und wissenschaftlichen Anwendungen wird eine große Dynamik benötigt: sowohl sehr kleine als auch sehr große Zahlen sollen einheitlich dargestellt werden
  $\Rightarrow$ möglich durch Verwendung von Gleitkommazahlen

- allgemeine Gleitkommazahl zur Basis *r* („*radix*“) ist definiert durch $x = a \times r^e$ mit
  -Argument oder **Mantisse** *a*
  −Charakteristik oder **Exponent** *e*

# Gleitkommazahlen

Wesentliche Merkmale der Gleitkommadarstellung:

- Bereich (Range): darstellbarer Zahlenbereich
- Genauigkeit (Precision): Anzahl der darstellbaren Stellen
- Fehlerschranke (Accuracy): Abweichung des Resultats einer Berechnung von einem tatsächlichen Wert.

# Gleitkommazahlen

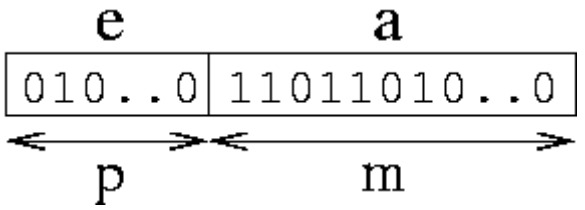
- eine Gleitkommazahl zur Basis *r* heißt **normalisiert (oder normiert)**, wenn für die Mantisse *a* gilt: $1 \leq |a| < r$

Beispiel (für $r = 10$):

- die Zahl 0,0000002345 kann dargestellt werden als $0{,}2345 \times 10^{-6}$
- die Zahl 1024500000,0 kann dargestellt werden als $0{,}10245 \times 10^{10}$

# Binäre Gleitkommazahlen

- eine **binäre Gleitkommazahl *x*** ist definiert durch
  $x = a \times 2^e$
  mit *m*-stelliger Mantisse *a*
  und *p*-stelligem Exponent *e*




- eine binäre Gleitkommazahl $x \neq 0$ heißt **normalisiert oder norinert**, wenn für die Mantisse *a* gilt: $1/2 \leq |a| < 1$, d.h. **die erste Stelle der Mantisse ist immer eine 1!**

## Warum normieren?

# Gleitkommazahlen: Darstellung des Exponenten

Darstellung des **Exponenten mit Bias *b*.**

Ausgangspunkt: ein p-Bit Exponent stellt die vorzeichenlosen Zahlen 0 bis $2^p$ dar.

Ziel: Wir müssen positive und negative Exponenten darstellen.

Idee: Wir benennen die Exponenten um, von $-2^{p-1}-1$ bis $2^p$ indem wir einen festen Wert (Bias = Versatz) $b = 2^{p-1}$ von jedem Exponenten abziehen. Dadurch erhalten wir eine kontinuierliche Zahlenfolge *0 ...N*, welche die Exponenten *–b* bis *N-b* darstellt.

Beisp: 3-Bit Exponent, d.h. $p = 3$ und $b(ias) = 2^{p-1} = 2^{3-1} = 4$:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 |

Allgemein gilt für **Exponenten mit Bias *b***: $x = a \times 2^{e-b}$ .

# Gleitkommazahlen: Darstellung des Exponenten

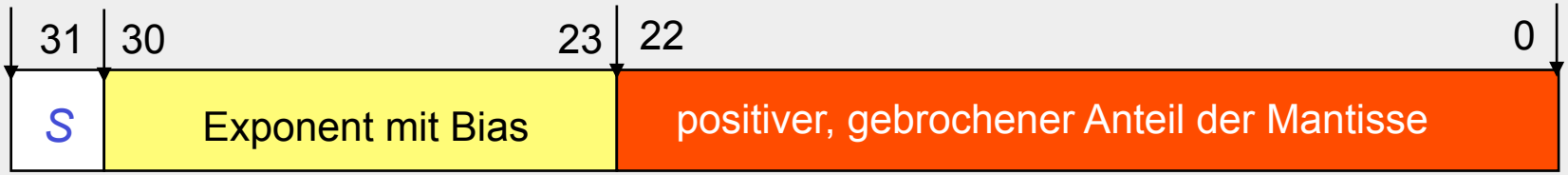
Beisp. 2: Codierung eines 4-Bit Exponentenfeldes, d.h. p=4, B = 8

| Binäre Repr. des Exp. | tatsächlicher Exponent | Exponent mit Versatz (biased form) | Binäre Repr. des Exp. | tatsächlicher Exponent | Exponent mit Versatz (biased form) |
|---|---|---|---|---|---|
| 0000 | -8 | 0 | 1000 | 0 | 8 |
| 0001 | -7 | 1 | 1001 | 1 | 9 |
| 0010 | -6 | 2 | 1010 | 2 | 10 |
| 0011 | -5 | 3 | 1011 | 3 | 11 |
| 0100 | -4 | 4 | 1100 | 4 | 12 |
| 0101 | -3 | 5 | 1101 | 5 | 13 |
| 0110 | -2 | 6 | 1110 | 6 | 14 |
| 0111 | -1 | 7 | 1111 | 7 | 15 |

# IEEE 754 Standard

Früher: Unterschiedliches Gleitkommaformat in jedem Prozessor,
Heute: Überwiegend Verwendung des **IEEE 754 Standard (1985)**

- allgemeine Definition: $x = (-1)^s \times 1.f \times 2^{e-b}$
  - ***S***: Vorzeichenbit mit 0 = positive Mantisse, 1= negative Mantisse
  - ***e***: Exponent (8 Bit, 0...255)
  - ***b***: Bias ($2^{8-1} - 1 = -127$)
  - ***f***: gebrochener Anteil der normalisierten Mantisse (23 Bit).
    Der ganzzahlige Anteil ist immer „1“ und wird daher nicht codiert.




IEEE 754 Gleitkommezahl, einfache Genauigkeit (single precision)

# IEEE 754 Standard

Beispiel: Umwandlung einer Dezimal-Festkommazahl in das IEEE 754 single precision format:

-3263,125 = -110010111111,001 als Binärzahl
= $-1,10010111111001 \times 2^{11}$ als normalisierte Binärzahl

Die Mantisse ist negativ: S=1
Der Exponent mit Bias 127 ist: $+11_{10}+127_{10} = 138_{10} = 10001010_2$
Der gebrochene Teil der Mantisse ist: ,10010111111001000000000 (als 23 Bit Zahl)

Die IEEE Gleikommazahl in einfacher Genauigkeit ist deshalb:

11000101010010111111001000000000

= $-1.59332275 \times 2^{11}$

Konversion von IEEE Zahlen: siehe Google

# Konversion von IEEE Zahlen:  siehe Google

**-123**

**32 bit IEEE single-precision form**

C2F60000 (hex)
11000010 11110110 00000000 00000000 (binary)

The Sign of the mantissa (and therefore the number) is 1 which represents a negative value.
The Exponent is 10000101 (binary) or 133 decimal.
The exponent field is represented in excess 127 so the exponent value is 6
The Mantissa is 11101100000000000000000 binary
with the implied leading 1, the mantissa is (1).EC0000 (hex)
The floating point representation is therefore -1.92187500 times 2 to the 6

---

| Vorzeichen | Exponent | Mantisse (die führende 1 wird nicht dargestellt!) | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 10000101 | 11101 | 100 0 | 0000 | 00 00 | 0000 | 00 |
| - | +6 | E | C | 0 | 0 | 0 | 0 |

# IEEE 754 Standard

- $e = e_{min} = (00..00)_2 = 0$ und $e = e_{max} = (11..11)_2$ werden zur Kodierung besonderer Zahlen verwendet:

  $x = +0$ („***positive Zero***“): $e = 0,\ f = 0,\ s = 0$

  $x = -0$ („***negative Zero***“): $e = 0,\ f = 0,\ s = 1$

  $x = +\infty$ („***positive Infinity***“): $e = e_{max},\ f = 0,\ s = 0$

  $x = -\infty$ („***negative Infinity***“): $e = e_{max},\ f = 0,\ s = 1$

  $x = \text{NaN}$ („***Not a Number***“): $e = e_{max},\ f \neq 0,\ s$ beliebig

  $x = (-1)^s \times 0.f \times 2^{1-b}$ („***Denormalized Number***“): $e = 0,\ f \neq 0$

- Denormalisierte Gleitkommazahlen ermöglichen die Darstellung sehr kleiner Werte im Bereich $2^{1-b-m}$ ... $2^{1-b}$

# IEEE 754 Standard

- drei verschiedene Formate spezifiziert:

| | *single precision* | *double precision* | *quad precision* |
|---|---|---|---|
| $n$ | 32 | 64 | 128 |
| $m$ | 23 | 52 | 112 |
| $s$ | 1 | 1 | 1 |
| $p$ | 8 | 11 | 15 |
| $e_{\min}$ | 0 | 0 | 0 |
| $e_{\max}$ | 255 | 2047 | 32767 |
| $b$ | 127 | 1023 | 16383 |
| $\lvert x_{\min} \rvert$ | $2^{-126} \approx 10^{-38}$ | $2^{-1022} \approx 10^{-308}$ | $2^{-16382} \approx 10^{-4932}$ |
| $\lvert x_{\max} \rvert$ | $(2-2^{-23})\times 2^{127} \approx 10^{38}$ | $(2-2^{-52})\times 2^{1023} \approx 10^{308}$ | $(2-2^{-112})\times 2^{16383} \approx 10^{4932}$ |

# IEEE 754 Standard

- Behandlung von **Ausnahmesituationen**:
  - **Überlauf** tritt ein, wenn nach Normalisierung für $x$ gilt: $e \geq e_{max}$
    a) Generierung von $+\infty$ , falls $x > 0$
    b) Generierung von $-\infty$ , falls $x < 0$
  - einige **Rechenregeln für $\infty$**:
    $\infty + x = \infty$ (falls $x \neq -\infty$), $\infty - x = \infty$ (falls $x \neq \infty$),
    $\pm x / 0 = \pm\infty$ (falls $x \neq 0$), $\infty \cdot x = \pm\infty$ (falls $x \neq 0$)
  - einige Operationen liefern ein **unbestimmtes Ergebnis**, z.B.:
    $\infty \cdot 0 =$ NaN, $0 / 0 =$ NaN, $\infty - \infty =$ NaN,
    Ferner gilt für alle Operationen: $f(x, \text{NaN}) =$ NaN
  - **Unterlauf** tritt ein, wenn nach Normalisierung für $x$ gilt: $e = 0$
    a) Generierung von $x = 0$ („*flushing to zero*“)
    b) Generierung einer denormalisierten Darstellung von $x$

# Gleitkomma-Addition

- Algorithmus zur Addition zweier IEEE-Gleitkommazahlen
  $x = (-1)^s \times a \times 2^{\alpha\text{-bias}}$ und $y = (-1)^t \times b \times 2^{\beta\text{-bias}}$ :

  1) **Sortiere** $x$ und $y$, so daß $x$ die Zahl mit kleinerem Exponenten ist

  2) **Anpassung der Exponenten**: Bestimme $x' = (-1)^s \times a' \times 2^{\beta\text{-bias}}$
     durch Rechtsschieben von $a$ um $\beta - \alpha$ Bitpositionen

  3) **Addiere Mantissen**:
     a) Falls nötig, bilde Zweierkomplement von $a'$ oder $b$
     b) Führe Festkomma-Addition $c = a' + b$ aus
     c) Falls $c < 0$, setze $u = 1$ und bilde Zweierkomplement von $c$

  4) **Normalisiere Ergebnis** $z = (-1)^u \times c \times 2^{\beta-\text{bias}}$
     a) Falls $c \geq 2$, schiebe $c$ nach rechts und inkrementiere $\beta$
     b) Falls $c < 1$, schiebe $c$ nach links und dekrementiere $\beta$
        ggf. wiederhole b), bis $1 \leq c < 2$

  5) **Behandlung von Ausnahmesituationen**: Überlauf, Unterlauf, $c = 0$

# Gleitkomma-Multiplikation

- Algorithmus zur Multiplikation zweier IEEE-Gleitkommazahlen $x = (-1)^s \times a \times 2^{\alpha\text{-bias}}$ und $y = (-1)^t \times b \times 2^{\beta-\text{bias}}$ :

  1) **Multipliziere Mantissen** als Festkommazahlen: $c = a \times b$
  $a = 1.f_a$ und $b = 1.f_b$ haben $m + 1$ Stellen $\Rightarrow$ $c$ hat $2m + 2$ Stellen !

  2) **Addiere Exponenten**: $\gamma = \alpha + \beta - \text{bias}$

  3) **Berechne Vorzeichen des Produktes**: $u = s \oplus t$

  4) **Normalisiere Ergebnis** $z = (-1)^u \times c \times 2^{\gamma\text{-bias}}$
     a) Falls $c \geq 2$, schiebe $c$ um 1 nach rechts und inkrementiere $\gamma$
     b) Setze $c = 1.f_c = 1.(c_{2m-1}\ c_{2m-2} \ldots c_m)_2$ mit Rundung

  5) **Behandlung von Ausnahmesituationen**:
     a) Überlauf, falls $\gamma \geq e_{max} = 2^p - 1$ $\Rightarrow$ $z := \pm\infty$ (abhängig von $u$)
     b) Unterlauf, falls $\gamma \leq e_{min} = 0$ $\Rightarrow$ Denormalisierung durchführen !
     c) Zero, falls $c = 0$ $\Rightarrow$ $z := \pm 0$ (abhängig von $u$)

# Arithmetische Operationen und Einheiten

## (für Festkommazahlen)

- Binäre Addition
- Binäre Subtraktion
- Binäre Multiplikation

# Zahlendarstellungen im Digitalrechner

- ausschließliche Verwendung von Binärzahlen, die auf Worte einer Breite von *n* Bit abgebildet werden
- im Prozessor stehen zur Speicherung ganzer Zahlen mehrere *n*-Bit Register (realisiert durch *n* Flip-Flops) zur Verfügung
- typische Wortbreiten:

  *n* = 8: **Byte**, in aktuellen Microcontrollern und Mikroprozessoren der 1G, wie z.B. Intel 8080, 6502, 6800 oder Z80

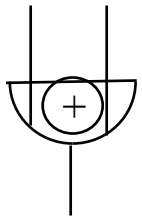
  *n* = 16: **Wort** in Minicomputern und Mikroprozessoren der zweiten Generation, wie z.B. PDP-11, Intel 8086, Motorola 68000

  *n* = 32: **Doppelwort** in Mikroprozessoren der dritten Generation, wie z.B. Intel Pentium, Motorola 68040

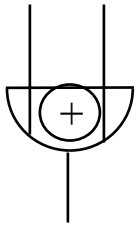
  *n* = 64: **Quadwort** in aktuellen Hochleistungsprozessoren, wie z.B. PowerPC, Alpha 21264, UltraSPARC, Intel Itanium

# Elementare Operationen auf binären Zahlen

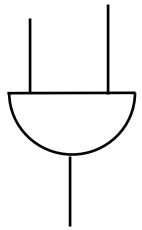
| Addition | Subtraktion | Multiplikation |
|---|---|---|
| 0 + 0 = 0 | 0 - 0 = 0 | 0 • 0 = 0 |
| 0 + 1 = 1 | 0 - 1 = 1 (borrow) | 0 • 1 = 0 |
| 1 + 0 = 1 | 1 - 0 = 1 | 1 • 0 = 0 |
| 1 + 1 = 0 (carry) | 1 - 1 = 0 | 1 • 1 = 1 |



Binärer „Vierteladdierer“



Binärer „Viertelsubtrahierer“



Binärer 1-Bit-Multiplizierer

# Die vorzeichenlose binäre Addition

- Addition zweier positiver *n*-stelliger Binärzahlen *a* und *b* kann **stellenweise von rechts nach links** durchgeführt werden
  (der handschriftlichen Addition zweier Dezimalzahlen entsprechend !)
- in jeder Stelle *i* kann ein **Übertrag** $c_i = 1$ auftreten („**Carry**“)
- Gilt für die Summe $s = a + b \geq 2^n$, so kann das Ergebnis nicht mehr als *n*-Bit Zahl dargestellt werden; es entsteht ein (*n*+1)-tes Summenbit, das als **Überlauf** („Overflow“) bezeichnet wird.

Beispiel 1: (n=8)

| | | |
|---|---|---|
| | 00010111 | $(23)_{10}$ |
| | + 01010110 | $(86)_{10}$ |
| Carry → | 1 11 | |
| | 01101101 | $(109)_{10}$ |

Beispiel 2: (n=8)

| | | |
|---|---|---|
| | 00110111 | $(55)_{10}$ |
| | + 11010110 | $(214)_{10}$ |
| Carry → | 1111 11 | |
| Überlauf → | 100001101 | $(269)_{10}$ |
| | ↓ | ↓ |
| 8–Bit Summe: | 00001101 | $(13)_{10}$ |

# Die vorzeichenlose binäre Addition

- Addition der niedrigstwertigen Bits von $a$ und $b$ erfordert einen **Halbaddierer**, der aus $a_0$ und $b_0$ sowohl die Summe $s_0$ als auch den Übertrag (Carry) $c_0$ ermittelt:

Wahrheitstabelle:

| $a_0$ | $b_0$ | $s_0$ | $c_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

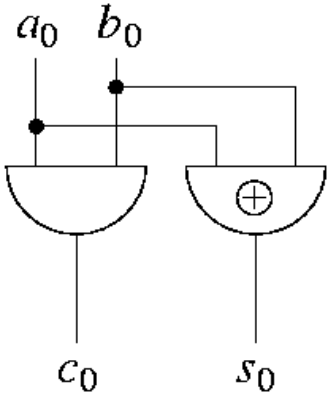
Gleichungen:

$$c_0 = a_0 \cdot b_0$$

$$s_0 = \overline{a_0} \cdot b_0 + a_0 \cdot \overline{b_0} = a_0 \oplus b_0$$

Symbol:


mögliche Realisierung:




- Verzögerung für $c_0$: $\tau$ (mit $\tau$ = Gatterlaufzeit),
  Verzögerung für $s_0$: $\tau$ oder $2\tau$ (je nach Realisierung)

# Die vorzeichenlose binäre Addition

- Addition von $a_i$, $b_i$ und $c_{i-1}$ an den Bitpositionen $i = 1, \ldots, n-1$ erfordert einen **Volladdierer** (FA = „Full Adder“), der die Summe $s_i$ und den Übertrag $c_i$ bestimmt:

Wahrheitstabelle:

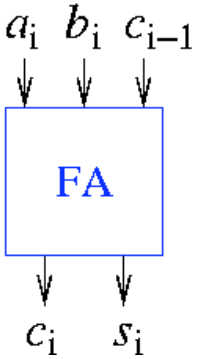
| $a_i$ | $b_i$ | $c_{i-1}$ | $s_i$ | $c_i$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Gleichungen:

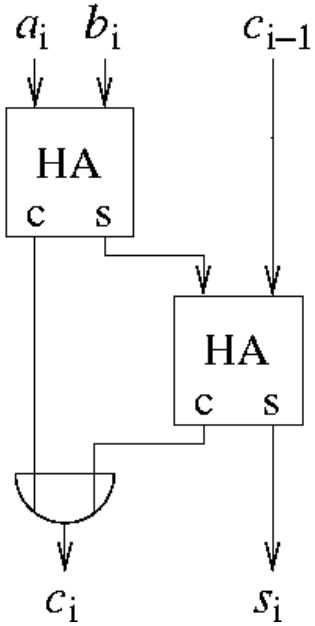
$$c_i = a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1}$$

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

Symbol:



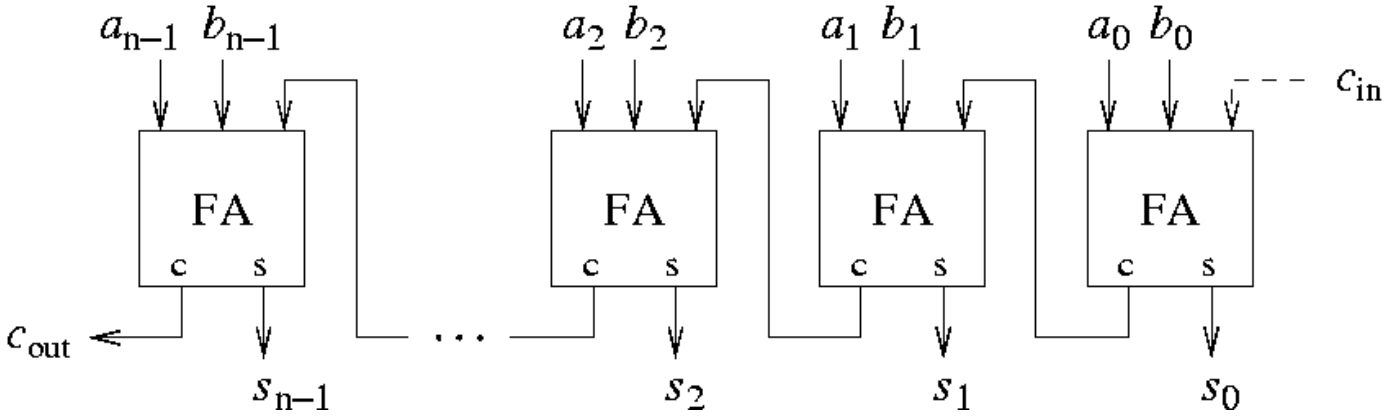

Realisierung mit HAs:




- Verzögerung je nach Pfad und Realisierung: $2\tau$ bis $4\tau$

# Einfache Addierwerke

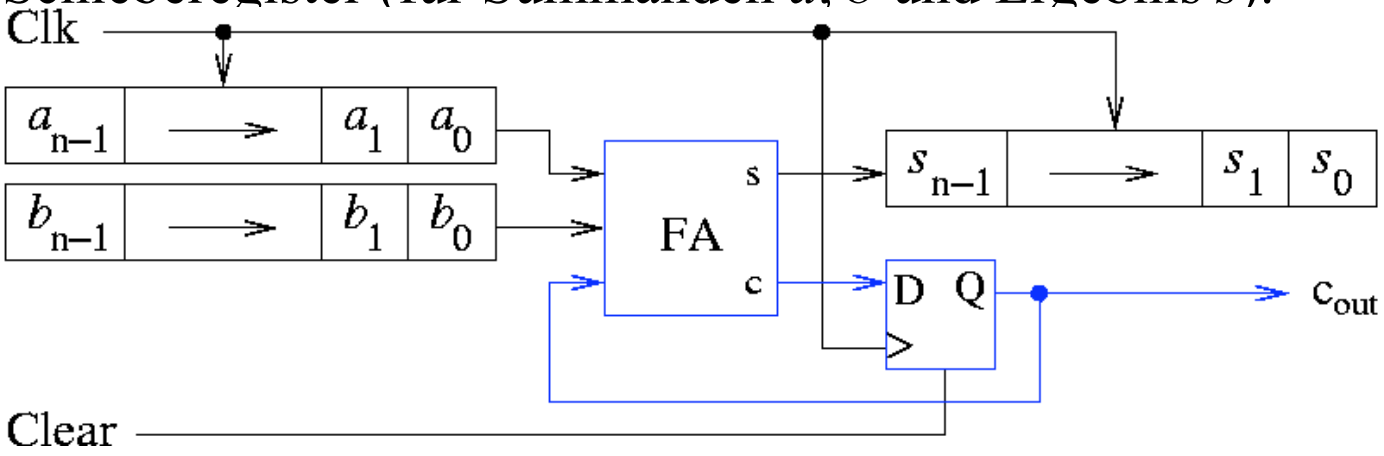
- **paralleles binäres Addierwerk**:
  - $n$ Volladdierer ermöglichen Addition zweier $n$-Bit Zahlen:




  - an Bitposition 0 genügt bei der Addition zweier positiver Zahlen auch ein Halbaddierer
  - im ungünstigen Fall kann ein an Position 0 entstehender Übertrag die Bitstellen 1 bis $n-1$ durchlaufen und $s_1$ bis $s_{n-1}$ verändern
    $\Rightarrow$ auch als „**Ripple Carry**“-**Addierer** (**RCA**) bezeichnet
  - resultierende maximale Verzögerung: $2\tau + (n-1) \cdot 2\tau = 2n\tau$

# Einfache Addierwerke (2)

- **serielles binäres Addierwerk:**
  - benötigt nur einen Volladdierer, ein Flip-Flop und drei $n$-Bit Schieberegister (für Summanden $a$, $b$ und Ergebnis $s$):




  - synchrones Schaltwerk; Flip-Flop muß jedoch zuvor initialisiert werden
  - in Takt $i$ wird Ergebnisbit $s_i$ aus $a_i$, $b_i$ und $c_{i-1}$ bestimmt ⇒ Addition von zwei $n$-Bit Zahlen benötigt $n$ Taktzyklen

# Carry Look-Ahead Addierer

- für eine schnelle Addition ist RCA mit einer maximalen Verzögerung von $2n\tau$ ungeeignet

  $\Rightarrow$ gibt es auch eine **schnellere** Möglichkeit, zwei $n$-Bit Zahlen zu addieren ?

- Idee: Ersetzung des sequentiellen Übertrag-Durchlaufs durch eine parallele **Vorausberechnung** aller Überträge $c_i$

- Ansatz: Betrachte $i$-ten Volladdierer eines RCA

  – Es gilt: $c_i = a_i \cdot b_i + (a_i + b_i) \cdot c_{i-1} := G_i + P_i \cdot c_{i-1}$

  – „**Generate**“: $G_i = a_i \cdot b_i$ gibt an, ob in Stelle $i$ ein Übertrag erzeugt wird

  – „**Propagate**“: $P_i = a_i + b_i$ gibt an, ob in Stelle $i$ ein Übertrag propagiert wird ($P_i = 1$) oder nicht ($P_i = 0$)

# Carry Look-Ahead Addierer (2)

- für die Überträge $c_i$ ergibt sich somit:

  $c_0 = a_0 b_0 := \boldsymbol{G_0}$

  $c_1 = a_1 b_1 + (a_1 + b_1) c_0 := \boldsymbol{G_1 + P_1 G_0}$

  $c_2 = \boldsymbol{G_2 + P_2 G_1 + P_2 P_1 G_0}$

  $c_3 = \boldsymbol{G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0}$

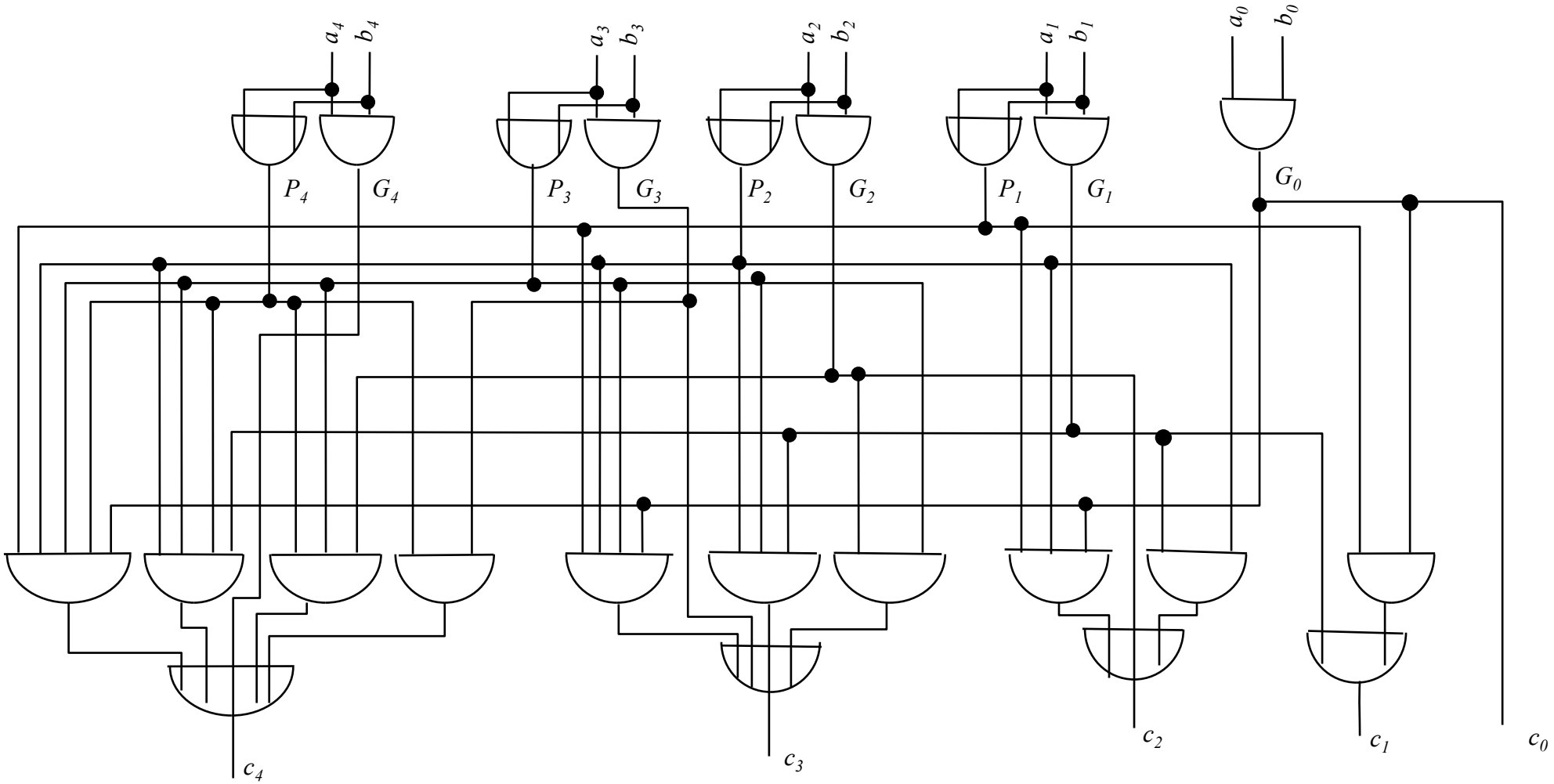
  $c_4 = \boldsymbol{G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 G_0}$

  $c_5 = ...$

- Verzögerung:
  - zur Bestimmung aller $P_i$ und $G_i$ : $\tau$
  - zur Bestimmung aller Signale $c_i$ bei gegebenen $P_i$ und $G_i$ : $2\tau$

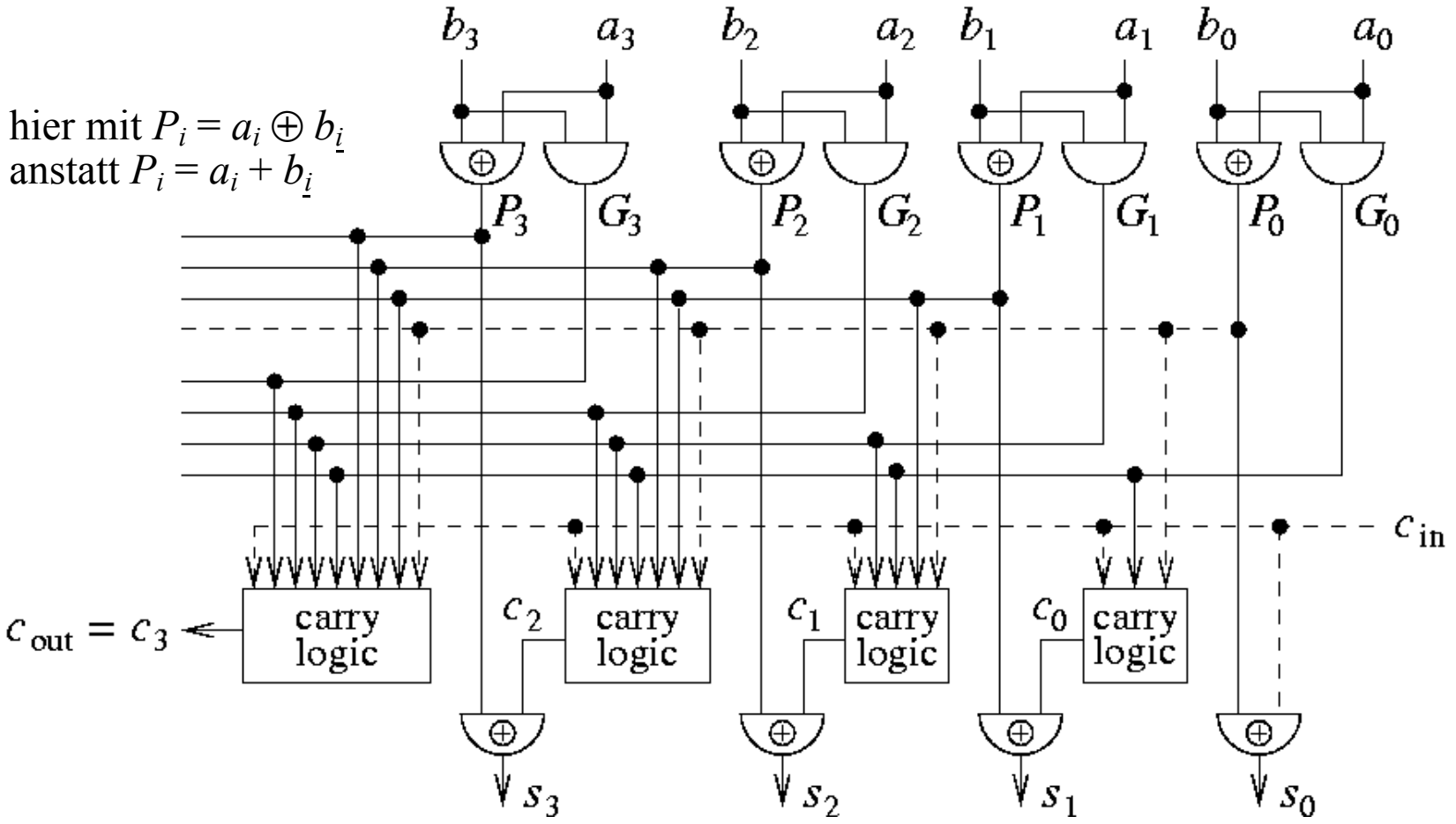
    (jedoch werden zur Bestimmung von $c_i$ mehrere große UND-Gatter mit max. i + 1 Eingängen und ein großes ODER-Gatter mit i + 1 Eingängen benötigt $\Rightarrow$ Annahme eines einheitlichen $\tau$ ist unrealistisch !)

# Carry Look-Ahead Netzwerk



# Carry Look-Ahead Addierer

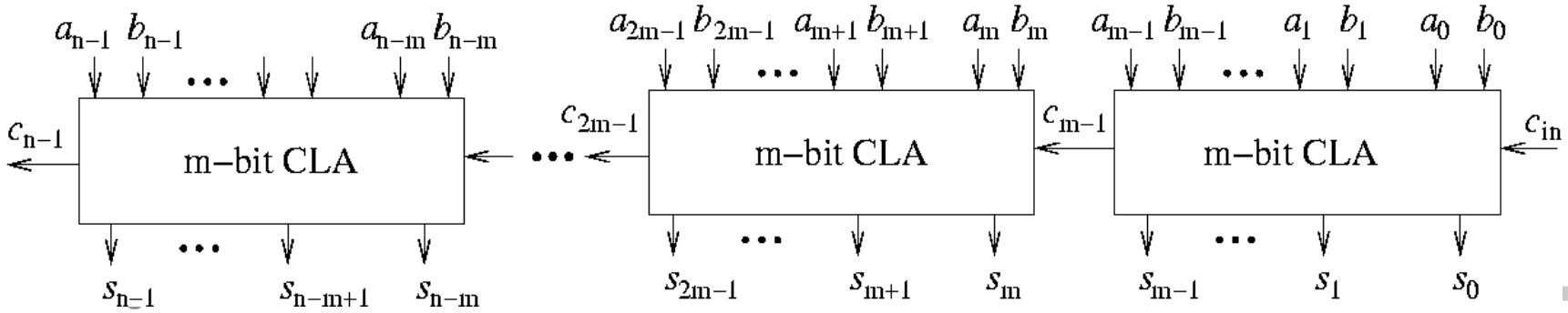
- **Carry Look-Ahead Addierer** (**CLA**), hier für $n = 4$ Bit:



# Carry Look-Ahead Addierer

- ein vollständiger *n*-Bit CLA benötigt zur Addition zweier *n*-Bit Zahlen die **Zeit 4τ** (unabhängig von *n*) !
- Probleme des vollständigen *n*-Bit CLA:
  - **hoher Aufwand** für große *n*
  - Gatter mit bis zu *n* + 1 Eingängen erforderlich ⇒ **hoher *„fan-in“***
  - Gatterausgänge $P_i$ und $G_i$ sind mit bis zu $(n+1)^2 / 4$ Gattereingängen verschaltet ⇒ **hoher *„fan-out“***

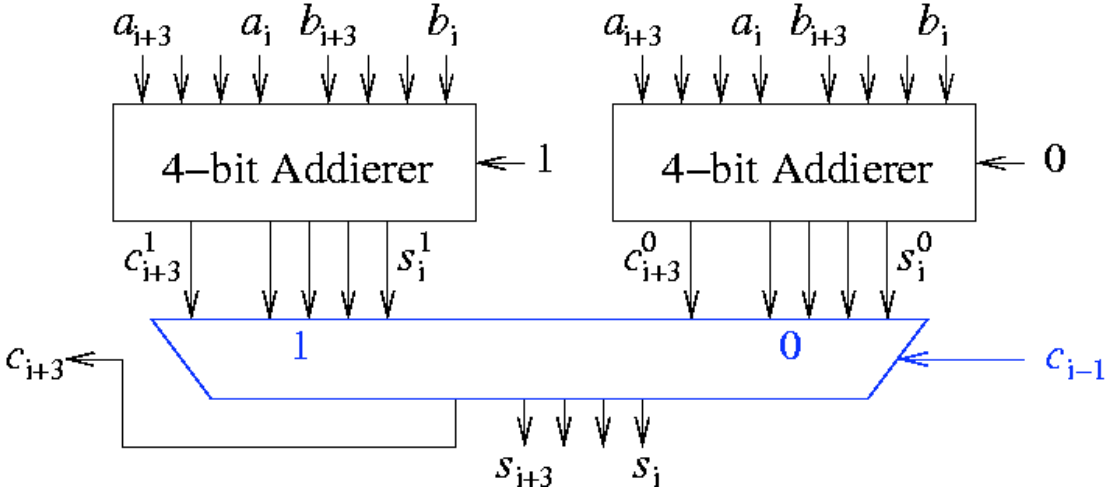
  **⇒ vollständiger CLA ist nicht praktikabel !**
- Kombination von RCA und CLA als Alternative:



# Carry-Select Addierer

- Idee:
  - in einem *m*-Bit Addierblock werden zunächst die Summenbits $s_{i+m-1}, s_{i+m-2}, \ldots, s_i$ sowohl für $c_i = 0$ als auch für $c_i = 1$ bestimmt
  - das richtige Ergebnis wird später bei Vorliegen des Signals $c_{i-1}$ über einen **Multiplexer** ausgewählt
- Beispiel:

  4-Bit Carry-Select Addierblock

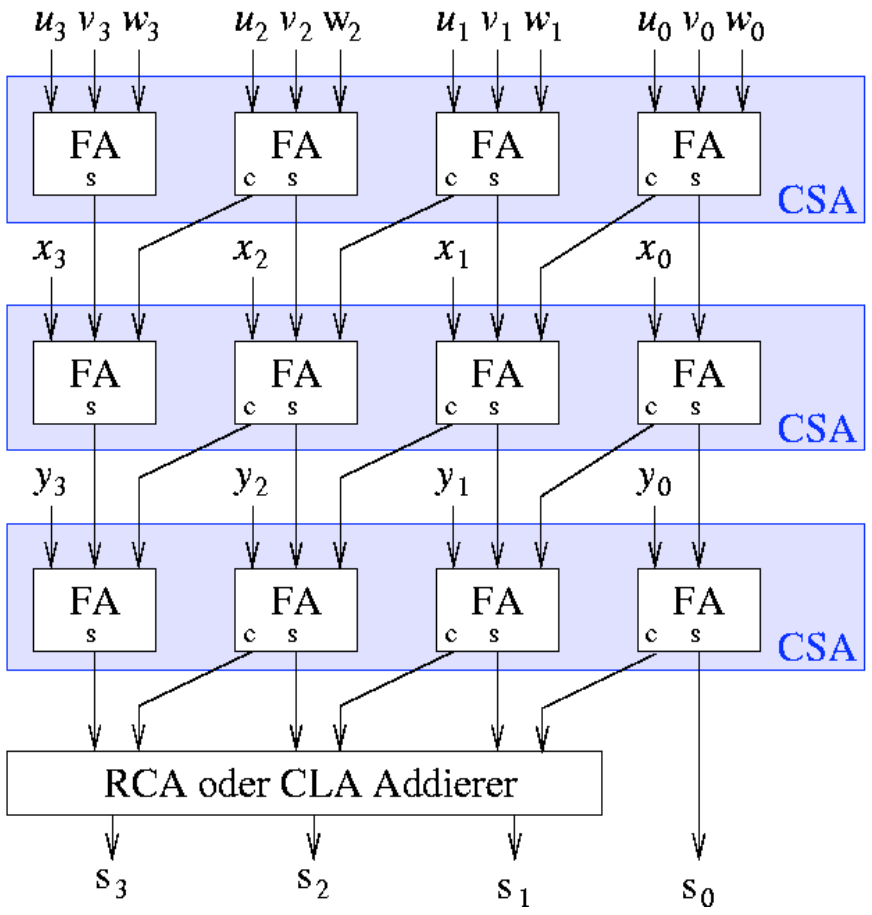



- als *m*-Bit Addierer kann hier wahlweise ein RCA oder ein CLA verwendet werden !

# Carry-Save Addierer (CSA)

- Idee:
  - Addition von **mehreren** Zahlen, Berücksichtigung entstehender Überträge erst bei der Addition des nächsten Summanden
  - ein *m*-Bit CSA-Baustein ermöglicht eine **partielle Addition** von drei *m*-Bit Zahlen
  - ein **RCA** oder **CLA** dient der Addition der noch verbleibenden Überträge
- zur Addition von *k* Zahlen werden $k-2$ CSA benötigt

Beispiel: 4–Bit CSA zur Addition von fünf 4–Bit Zahlen u, v, w, x und y



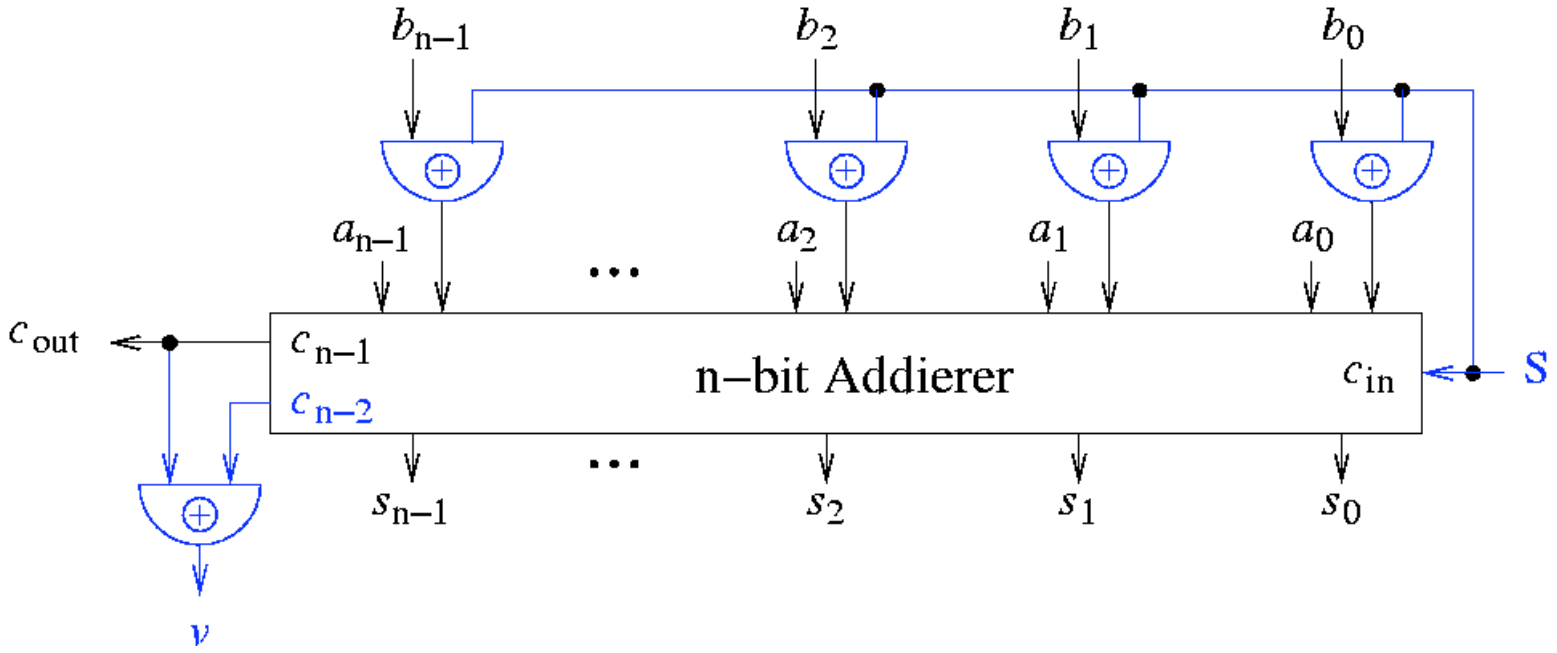
# binäre Subtraktion

- statt der Entwicklung eines eigenen Subtrahierwerkes kann bei Verwendung des Zweierkomplements **jedes Addierwerk auch zur Subtraktion verwendet werden**
- Ansatz: $a - b = a + (-b)$
- Realisierung für zwei $n$-stellige Zahlen a und b:
  - Invertierung aller Bits $b_i$
  - Addition von 1 zur Bildung des Zweierkomplements $-b$

    (kann z.B. in den zuvor besprochenen Addierwerken durch Setzen von $c_{in} = 1$ ohne zusätzlichen Hardwareaufwand erreicht werden!)
  - Addition von $a$ und $(-b)$
  - Summe ist korrekt, wenn Überlaufsbit $c_{n-1}$ ignoriert wird
- echter **Überlauf *v*** tritt nur dann ein, wenn gilt: $c_{n-1} \neq c_{n-2}$

# binäre Subtraktion

- Realisierung eines *n*-Bit **Addier-/Subtrahierwerkes**:




  - Addition $a + b$ bei S = 0, Subtraktion $a - b$ bei S = 1
  - Invertieren der Bits $b_i$ bei S = 1 durch XOR-Gatter
  - intern ist jedes beliebige *n*-Bit Addierwerk verwendbar

# Komplementäre Zahlendarstellung

## Bedingungen für den Überlauf (Overflow)

1. **Der Überlauf hat nur eine Bedeutung in Berechnungen, die auf der Komplementdarstellung beruhen.**
2. **Ein Überlauf tritt auf, wenn durch die binäre Addition der Zahlenbereich überschritten wird.**

**Die Bedingungen dafür sind: Gegeben die Operanden *a* und *b* und das Ergebnis *s***

**a.) ( a > 0 ) UND ( b > 0 ) UND (s < 0 ) oder**
**b.) ( a < 0 ) UND ( b < 0 ) UND (s > 0 )**

1. Fall: beide VZ=0, VZ der Summe wird durch Carry gesetzt, aber es wird kein Carry aus der Stelle erzeugt da ja beide VZ=0 waren.

Fall 2: Beide VZ=1. Ergebnis-VZ =0. Das bedeutet, dass KEIN Carry IN die letzte Stelle erzeugt wurde. Da beide VZ=1 sind, wird aber ein Carry AUS der Stelle erzeugt.

# binäre Multiplikation

- Algorithmus zur **Multiplikation zweier positiver Binärzahlen** *a* und *b* entspricht der handschriftlichen Multiplikation zweier positiver Dezimalzahlen

Algorithmus:

```
p = 0
for i = 0 to n-1 {
   if (b_i = 1) {
      q = shift left a by i
      p = p + q
      }
}
```

Beispiel für n=5:

```
01010 × 01101
-------------
    01010   ×1
   00000 0  ×0
  01010 00  ×1
 01010 000  ×1
00000 0000  ×0
-------------
0010000010
```

- Produkt *p* zweier positiver *n*-Bit Binärahlen hat **2*n* Bitstellen**
- Algorithmus ist zurückführbar auf wiederholte **bedingte Additionen und Schiebeoperationen**

# Multiplizierwerke

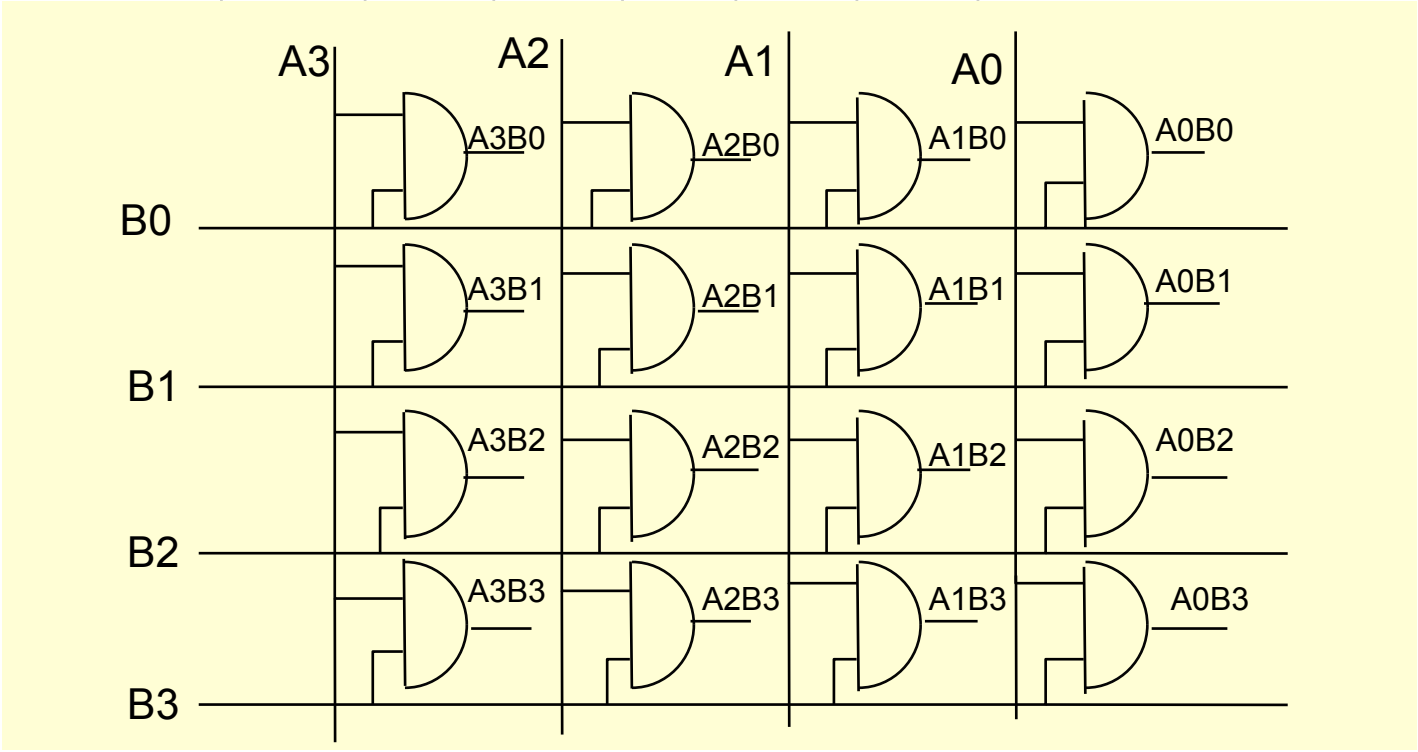
- **Implementierung B: Feldmultiplizierer** („array multiplier“)
  - direkte Realisierung des handschriftlichen Multiplikationsschemas für Binärzahlen $a$ und $b$ in digitaler Logik
  - es werden zunächst alle **Bitprodukte** $a_i \cdot b_j$ ermittelt
  - für jedes Multiplikatorbit $b_i$ wird hieraus ein **partielles Produkt** $q_i = (a_{n-1}b_i \;\; a_{n-2}b_i \;\; \ldots \;\; a_1 b_i \;\; a_0 b_i)_2$ gebildet
  - die $n$ partiellen Produkte $q_i$ werden jeweils um i Bitpositionen nach links verschoben und aufaddiert
  - Beispiel für $n = 4$:

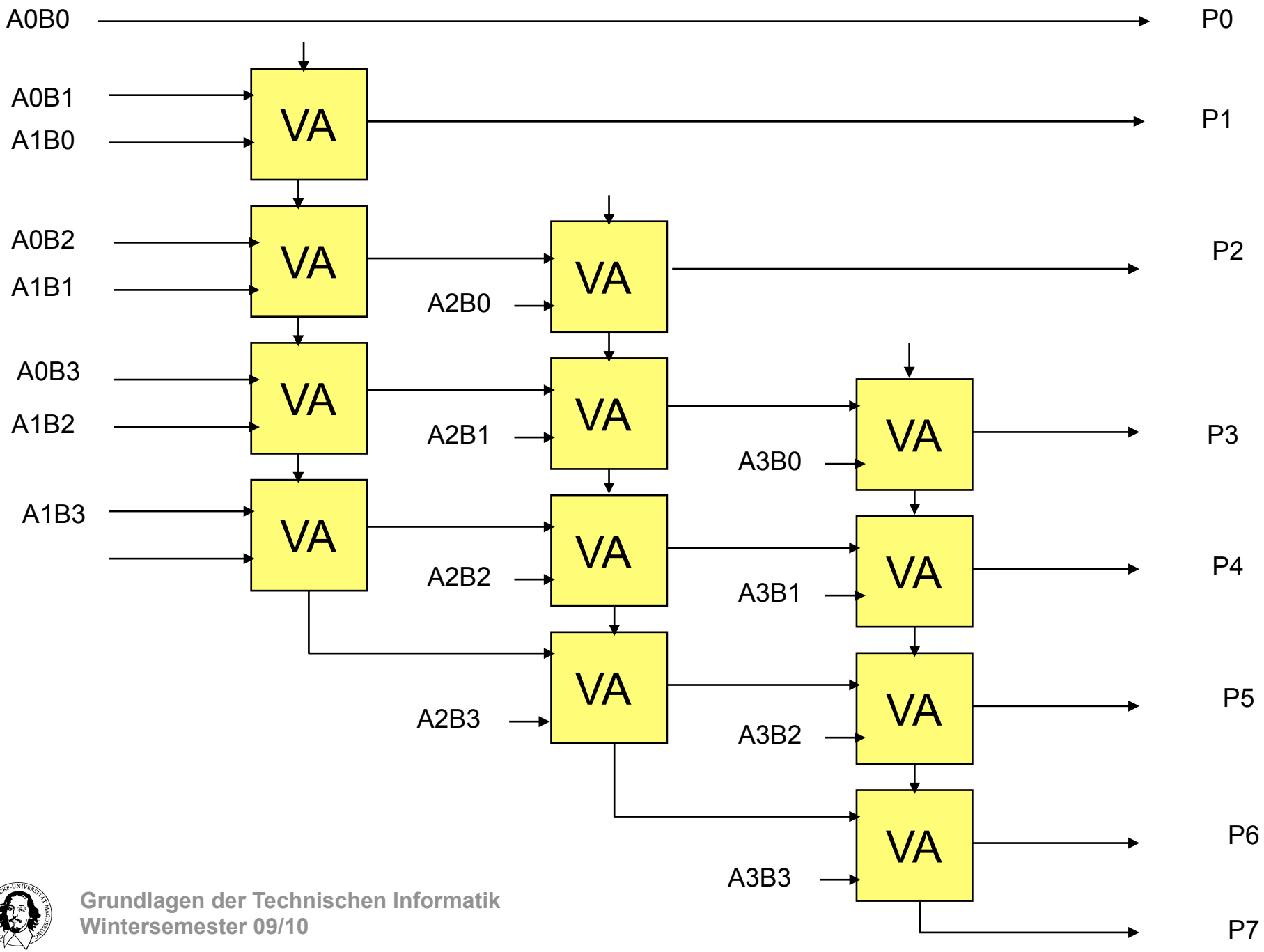
| $a_3$ | $a_2$ | $a_1$ | $a_0$ | × | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
|---|---|---|---|---|---|---|---|---|
| | | | | | $a_3b_0$ | $a_2b_0$ | $a_1b_0$ | $a_0b_0$ |
| | | | | $a_3b_1$ | $a_2b_1$ | $a_1b_1$ | $a_0b_1$ | 0 |
| | | | $a_3b_2$ | $a_2b_2$ | $a_1b_2$ | $a_0b_2$ | 0 | 0 |
| | | $a_3b_3$ | $a_2b_3$ | $a_1b_3$ | $a_0b_3$ | 0 | 0 | 0 |
| $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | |

# binäre Multiplikation (parallel)

B3 B2 B1 B0 x A3 A2 A1 A0

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | A0B3 | A0B2 | A0B1 | A0B0 |
| | | | A1B3 | A1B2 | A1B1 | A1B0 | |
| | A3B3 | A3B2 | A3B1 | A3B0 | | | |
| M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |




A0B0
P0
A0B1
A1B0
VA
P1
A0B2
A1B1
VA
A2B0
VA
P2
A0B3
A1B2
VA
A2B1
VA
A3B0
VA
P3
A1B3
VA
A2B2
VA
A3B1
VA
P4
A2B3
VA
A3B2
VA
P5
A3B3
VA
P6
P7

# Multiplizierwerke

- **Implementierung D: zweistufiges Schaltnetz**
  - Idee: Realisierung eines $n \times n$ Bit Multiplizierers als zweistufiges Schaltnetz mit $2n$ Eingängen und $2n$ Ausgängen
  - **sehr geringe Zeitverzögerung**: nur $2\tau$
  - Implementierung z.B. durch ein **ROM** oder **PROM** mit $2^{2n}$ Zeilen aus $2n$-Bit Worten
  - jedoch **sehr hoher Speicher-aufwand**:

| $n$ | Produkt $2n$ | Zeilen $2^{2n}$ | PROM Größe |
|---|---|---|---|
| 2 | 4 | 16 | **64 Bit** |
| 4 | 8 | 256 | **256 Byte** |
| 8 | 16 | 65536 | **128 KByte** |
| 16 | 32 | $4.3 \cdot 10^9$ | **1.6 GByte** |

# Multiplikation negativer Zahlen

- bislang Betrachtung ausschließlich **positiver** Multiplikatoren und Multiplikanden
- Was passiert bei **negativen** im Zweierkomplement kodierten $n$-Bit Multiplikatoren bzw. $n$-Bit Multiplikanden ?

$(-a) \cdot (-b) = (2^n - a) \cdot (2^n - b) = 2^{2n} - a \cdot 2^n - b \cdot 2^n + a \cdot b$ (statt $a \cdot b$)

$a \cdot (-b) = a \cdot (2^n - b) = a \cdot 2^n - a \cdot b$ (statt $2^{2n} - a \cdot b$)

$(-a) \cdot b = (2^n - a) \cdot b = b \cdot 2^n - a \cdot b$ (statt $2^{2n} - a \cdot b$)

- ohne besondere Maßnahme liefert binärer Multiplizierer **falsche Ergebnisse**!
- aufwendige Addition von Korrekturtermen ist möglich

  z.B. Addition von $a \cdot 2^n + b \cdot 2^n$ im Falle von $(-a) \cdot (-b)$
- Alternative: Trennung von **Vorzeichen** und **Betrag**

  $\Rightarrow$ hoher Aufwand für Umwandlung von Zahlen vor/nach der Multiplikation

# Multiplikation nach Booth

Booth's Algorithmus hat zwei Ziele:

1. Schnelle Multiplikation durch Ausnutzung konsekutiver "0"- oder "1"-Ketten.

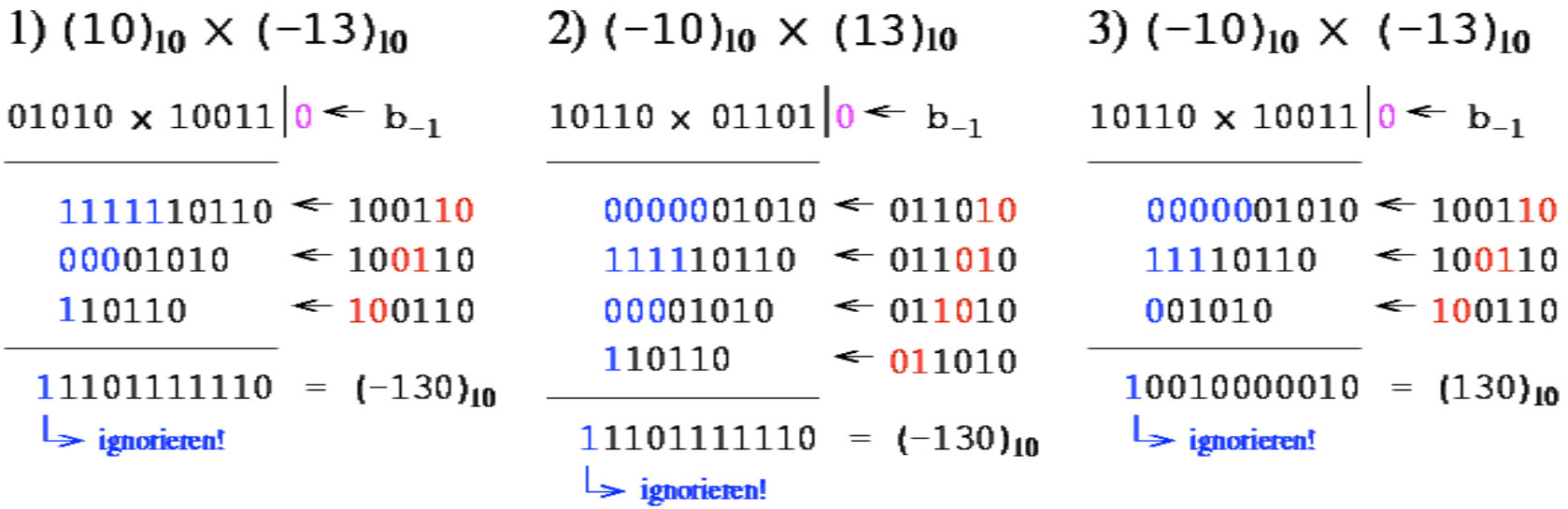
2. Korrekte Multiplikation von Zahlen in 2-Komplement Darstellung.

# Multiplikation nach Booth

- Idee: Vereinfachung der Multiplikation, wenn Multiplikator eine **1-Folge** repräsentiert: $a \times \mathbf{111} = a \times \mathbf{1000} - a \times \mathbf{0001}$
  - auch innerhalb längerer Bitfolgen möglich, z.B.
    $a \times 00\mathbf{111}00 = a \times 0\mathbf{1000}00 - a \times 0\mathbf{0001}00$
  - Multiplikation mit einer 1-Folge läßt sich stets durch **eine** Addition und **eine** Subtraktion ersetzen
- der **Algorithmus nach Booth** analysiert zwei benachbarte Bits $b_i$ und $b_{i-1}$ des Multiplikators:
  - $(b_i b_{i-1})_2 = 01_2$ : Addition von $a \times 2^i$
  - $(b_i b_{i-1})_2 = 10_2$ : Subtraktion von $a \times 2^i$
  - $(b_i b_{i-1})_2 = 00_2$ oder $(b_i b_{i-1})_2 = 11_2$ : keine Addition
- Ergänzung von $b_{-1} = 0$ erforderlich

# Multiplikation nach Booth

- Realisierung der Subtraktion von $a \times 2^i$ durch Addition des Zweierkomplements (vorzeichenrichtig ergänzt)
- Beispiele (mit $n = 5$):




- funktioniert für beliebige positive und negative Multiplikanden und Multiplikatoren!

http://www.ecs.umass.edu/ece/koren/arith/simulator/Booth/

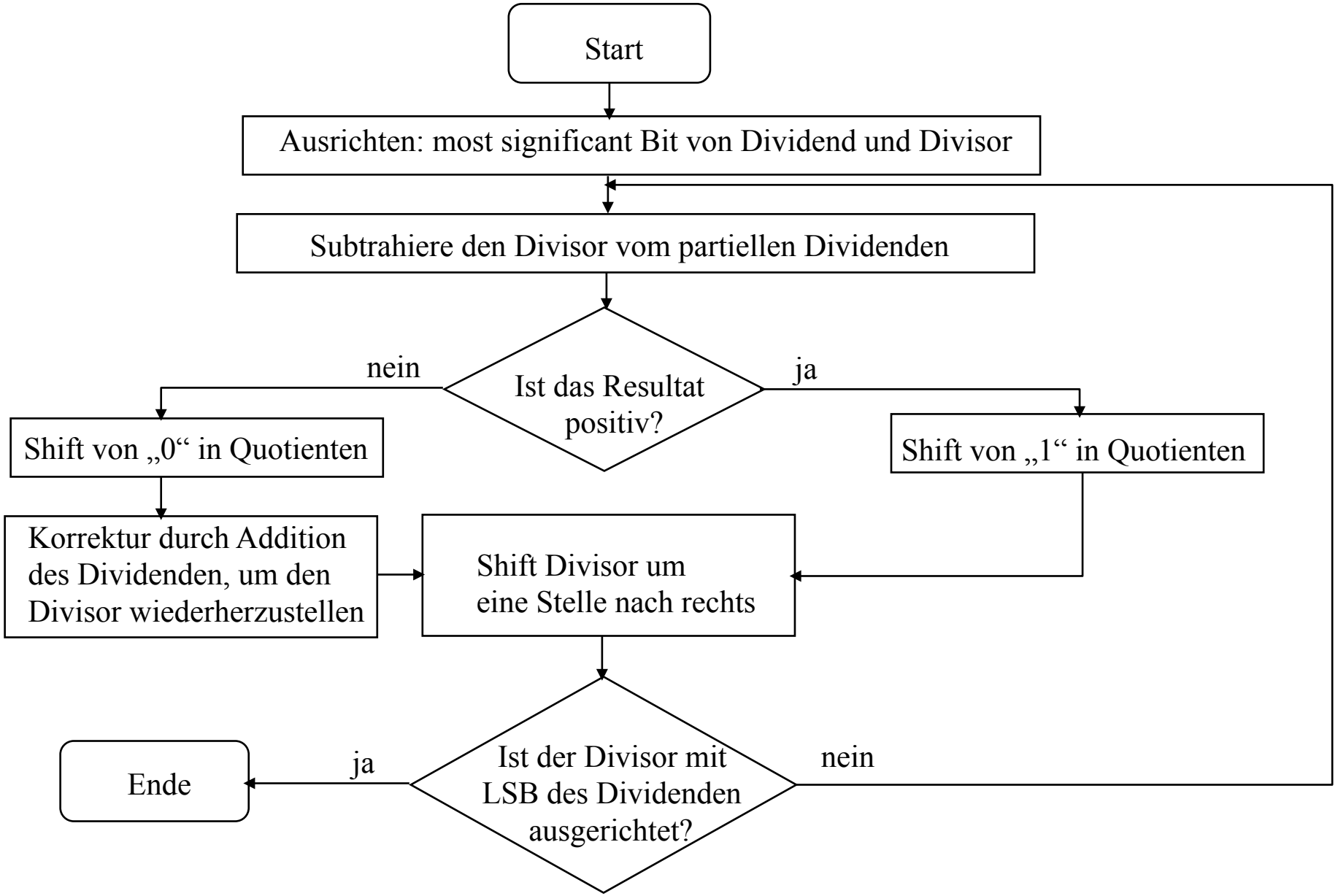
# binäre Division

- Umkehrung der Multiplikation: Berechnung von $q = a / b$ durch wiederholte **bedingte Subtraktionen** und **Schiebeoperationen**
- in jedem Schritt wird Divisor $b$ **testweise** vom Dividenden $a$ subtrahiert: $q_i = 1$, falls $a - b > 0$
  $q_i = 0$ und **Korrektur** durch $a = a + b$, falls $a - b < 0$
- Beispiel:

  $103_{10} / 9_{10} = 11_{10}$
  mit Rest $4_{10}$

```
0001100111   / 01001   = 01011
- 01001                    ↑
  00111                 Quotient
- 01001
  11110
+ 01001   ←—— Korrektur weil die
  001111       Subtraktion
- 01001       einen Wert < 0
  001101      erzeugt hat
- 01001
  00100   ←—— Rest
```

- auch als „**Restoring**“-Division bezeichnet


Start
Ausrichten: most significant Bit von Dividend und Divisor
Subtrahiere den Divisor vom partiellen Dividenden
Ist das Resultat positiv?
nein
ja
Shift von „0“ in Quotienten
Shift von „1“ in Quotienten
Korrektur durch Addition des Dividenden, um den Divisor wiederherzustellen
Shift Divisor um eine Stelle nach rechts
Ist der Divisor mit LSB des Dividenden ausgerichtet?
ja
nein
Ende

# Lernziele

- **Begriffe**: *b*-adisches Zahlensystem, Einer-/Zweierkomplement, Halb-/Volladdierer, RCA, CLA, Carry-Select Addition, Carry-Save-Addition, Feldmultiplizierer, BCD-Zahlen, Exponent mit Bias, ...
- **Umwandlung** von positiven und negativen Dezimalzahlen in ein anderes Zahlensystem und umgekehrt
- Verständnis der **Algorithmen** zur Addition, Subtraktion, Multiplikation und Division binärer Zahlen
- Aufbau von einfachen **Rechenwerken** für alle 4 Grundrechen-operationen
- Aufbau schneller Addier- und Multiplizierwerken
- Darstellung von **Gleitkommazahlen**, Rechnen mit IEEE Gleit-kommazahlen, Bedeutung sowie Behandlung von Ausnahme-situationen