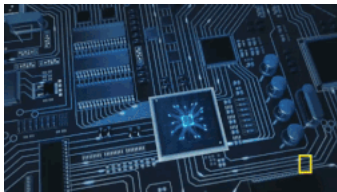


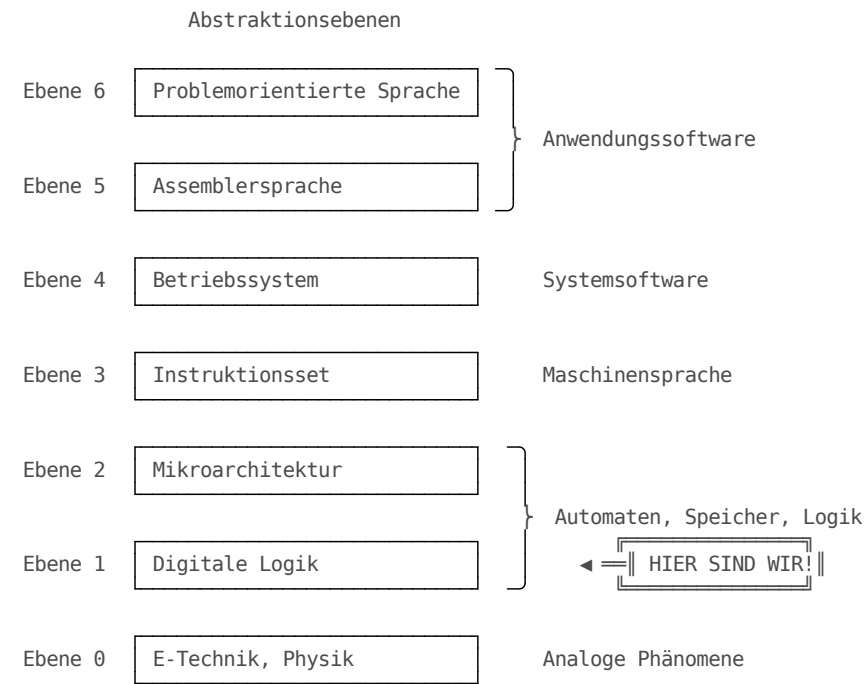
# Rechnerarithmetik

Parameter	Kursinformationen
Veranstaltung:	Digitale Systeme / Eingebettete Systeme
Semester:	Wintersemester 2025/26
Hochschule:	Technische Universität Freiberg
Inhalte:	Zahlendarstellung, binäre Arithmetik, Addierer und ALU-Grundlagen
Link auf GitHub:	<a href="https://github.com/TUBAF-lfi-LiaScript/VL_EingebetteteSysteme/blob/master/09_Rechnerarithmetik.md">https://github.com/TUBAF-lfi-LiaScript/VL_EingebetteteSysteme/blob/master/09_Rechnerarithmetik.md</a>
Autoren:	Sebastian Zug & André Dietrich & Fabian Bär



## Fragen an die Veranstaltung

- Was ist der Unterschied zwischen einem Stellenwertsystem und einem additiv-wirkenden Zahlensystem?
- Zu welcher Basis werden duale, oktale, hexadezimale Zahlen gebildet?
- Wie lassen sich Zahlen zwischen diesen Systemen umrechnen?
- Wie resultiert daraus der Zahlenraum für 2 Stellen?
- Welchen Vorteil bringt die komplementäre Zahlendarstellung mit sich?
- Warum arbeiten heutige Rechner ausschließlich mit der 2er-Komplementdarstellung?
- Wie sehen die Basis Addier-/Subtrahier Schaltungen aus? Was ist ein Volladdierer / Halbaddierer?
- Welchen Vorteil bringt der Carry-Look-Ahead Addierer gegenüber dem Carry-Ripple-Addierer mit?
- Warum werden beide Ansätze kombiniert?



**Zahlendarstellung**

**Ganzzahlige Zahlensystem**

**Frühe Zahlensysteme**

Als römische Zahlen werden die Zahlzeichen einer in der römischen Antike entstandenen und noch für Nummern und besondere Zwecke gebräuchlichen Zahlschrift bezeichnet.

Es handelt sich um eine additive Zahlschrift, mit ergänzender Regel für die subtraktive Schreibung bestimmter Zahlen, aber ohne Stellenwertsystem und ohne Zeichen für Null.

	1	2	3	4	5	6	7
Majuskel	I	II	III	IV	V	VI	VII
Minuskel	i	ii	iii	iv	v	vi	vii



Eingang des Kolosseums mit der römischen Zahl LII (52) <sup>[1]</sup>

<sup>[1]</sup>: WarpFlyght, Eingang des Kolosseums mit der römischen Zahl LII (52), [https://commons.wikimedia.org/wiki/File:Colosseum-Entrance\\_LII.jpg](https://commons.wikimedia.org/wiki/File:Colosseum-Entrance_LII.jpg)

## Dezimale Zahlen

Darstellung positiver ganzer Zahlen in positionalen Notation (auch als Stellenwertsystem bezeichnet) mit 10 Symbolen  $x_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ .

$$x = (x_{n-1}x_{n-2}x_{n-3}\dots x_2x_1x_0)_{10}$$

$$x = x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \dots + x_1 \cdot 10^1 + x_0 \cdot 10^0$$

Beispiel  $259_{10} = 2 \cdot 10^2 + 5 \cdot 10^1 + 9 \cdot 10^0 = 200 + 50 + 9$

## Binäre Zahlen

$$y = (y_{n-1}y_{n-2}y_{n-3}\dots y_2y_1x_0)_2$$

$$y = y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \dots + y_1 \cdot 2^1 + y_0 \cdot 2^0$$

Beispiel  $11101_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 4 + 1 = 29$

## Verallgemeinerung b-adische Zahlensysteme

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

$$z = (z_{n-1}z_{n-2}z_{n-3}\dots z_2z_1z_0)_b$$

$$z = z_{n-1} \cdot b^{n-1} + z_{n-2} \cdot b^{n-2} + \dots + z_1 \cdot b^1 + z_0 \cdot b^0$$

Typische Werte für die Basis  $b$ :

Zahlensystem	$b$	Ziffern
Dualzahlen	2 ( $2^1$ )	$x_i \in \{0, 1\}$
Oktalzahlen	8 ( $2^3$ )	$x_i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$
Dezimalzahlen	10	$x_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Hexadezimalzahlen	16 ( $2^4$ )	$x_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

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

**Aufgabe:** Informieren Sie sich über das *Sexagesimalsystem* der babylonischen Mathematik! Warum verwendete man die 60 als Grundlage?

**Merke:** Mit der Darstellung einer Zahl im binären Zahlensystem sinkt die Zahl der Ziffernsymbole. Gleichzeitig steigt die Zahl der notwendigen Stellen an! Wir brauchen mehr Platz.

Beispiel:  $214_{10}$

Zahlensystem	$x_7$	$x_6$	$x_5$	$x_4$	$x_3$	$x_2$	$x_1$
Dual	1	1	0	1	0	1	1
Oktal						3	2
Dezimal						2	1
Hexadezimal							D

Für die Realisierung der Wandlung einer dezimalen Zahl  $z$  in eine Zahl  $x$  zur Basis  $b$  folgt man folgendem Algorithmus:

```

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

Reste stellen das gesuchte Ergebnis dar:  $x = (r_{i-1} \dots r_1 r_0)_b$

Beispiel: Wandlung von  $29_{10}$  in eine binäre Zahl

Der Teiler definiert das avisierte Zahlensystem

29 / 2 = 14	Rest 1	↑ Leserichtung für die Binärzahl
14 / 2 = 7	Rest 0	
7 / 2 = 3	Rest 1	
3 / 2 = 1	Rest 1	
1 / 2 = 0	Rest 1	

$29_{10} = 11101_2$

Für binäre Zahlen kann mit Blick auf die bekannten Zweierpotenzen auch effizienter vorgegangen werden:

1. Man schreibe alle Zweierpotenzen, welche kleiner als die Dezimalzahl sind, rückwärts auf (beginne von rechts und schreibe links jeweils den mit 2 multiplizierten Wert).
2. Nun setzt man von links nach rechts eine 1 unter jede Potenz welche in die dezimale Zahl passt und subtrahiert die Potenz von der Zahl. Wenn die Potenz nicht in die Zahl passt schreibt man eine 0.
3. Dies wird wiederholt bis alle Potenzen belegt sind.

**Beispiel:**  $242_{10}$  in binär

128	64	32	16	8	4	2	1
1	1	1	1	0	0	1	0
242 – 128 = 114	114 – 64 = 50	50 – 32 = 18	18 – 16 = 2	2 – 8 < 0	2 – 4 < 0	2 – 2 = 0	0 – 1 < 0

**Aufgabe:** Wandeln Sie die Zahl  $523_{10}$  in eine binäre Zahl um.

523	/	2	=	261	Rest	1	↑
261	/	2	=	130	Rest	1	
130	/	2	=	65	Rest	0	
65	/	2	=	32	Rest	1	
32	/	2	=	16	Rest	0	
16	/	2	=	8	Rest	0	
8	/	2	=	4	Rest	0	
4	/	2	=	2	Rest	0	
2	/	2	=	1	Rest	0	
1	/	2	=	0	Rest	1	

$$523_{10}=1000001011_2$$

## Gebrochene Zahlen

Die Römer nutzten Brüche mit der Basis 12(!). Die Nutzung der 12 lag nahe, weil sich die am häufigsten benötigten Brüche *eine Hälfte, ein Drittel* und *ein Viertel* durch Vielfache von 1/12 darstellen lassen. Der römische Name für ein Zwölftel ist Uncia, ein Wort, das später zum Gewichtsmaß *Unze* wurde. Für Brüche, deren Zähler um 1 kleiner als der Nenner ist, wurde teilweise eine subtraktive Bezeichnung verwendet.

Die Darstellung einer gebrochenen Zahl ist in einem  $b$ -adischen System mit  $n$  Vorkomma und  $m$  Nachkommastellen definiert mit:

$$z = (z_{n-1} \dots z_1 z_0, z_{-1} \dots z_{-m+1} z_{-m})$$

$$z = z_{n-1} \cdot b^{n-1} + \dots + z_0 \cdot b^0, z_{-1} \cdot b^{-1} + \dots + z_{-m} \cdot b^{-m}$$


$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
128	64	32	16	8	4	2	1

Beispiel:  $1011,1101 = 8 + 2 + 1 + 0.5 + 0.25 + 0.0625 = 11.8125$

```

i=0
wiederhole, bis z=0:
  berechne z=2·z
  wenn z > 1
    notiere 1 für r[i]
    z=z-1
  sonst
    notiere 0 für r[i]
  i++

```



Beispiel 1: Wandeln Sie  $0.28125_{10}$  in eine duale Zahl

Der Faktor definiert das avisierte Zahlensystem

↓

0.28125	• 2 = 0.5625	< 1	→ 0	Rest	0.5625
0.5625	• 2 = 1.125	> 1	→ 1	Rest	0.125
0.125	• 2 = 0.25	< 1	→ 0	Rest	0.25
0.25	• 2 = 0.5	< 1	→ 0	Rest	0.5
0.5	• 2 = 1	<= 1	→ 1	Rest	0

↓

Ergebnis  $0.28125_{10} = 0.25 + 0.03125 = 0.01001$

Beispiel 2: Wandeln Sie  $0.1_{10}$  in eine duale Zahl

0.1	• 2 = 0.2	0	Rest	0.2
0.2	• 2 = 0.4	0	Rest	0.4
0.4	• 2 = 0.8	0	Rest	0.8
0.8	• 2 = 1.6	1	Rest	0.6
0.6	• 2 = 1.2	1	Rest	0.2
0.2	• 2 = 0.4	0	Rest	0.4
0.4	• 2 = 0.8	0	Rest	0.8
0.8	• 2 = 1.6	1	Rest	0.6
0.6	• 2 = 1.2	1	Rest	0.2

↓

Ergebnis Offenbar ist für den Wert  $0.1_{10}$  keine exakte Repräsentation im dualen System möglich  $0,0001100110011\dots_2$ . Welche Konsequenzen hat das?

Check01.cpp

```
1 x = .0
2 a = 1/10.
3 for i in range(0, 1000000):
4     x = x + a
5 print("{0:50.50f}\n{1}".format(a, x))
```

```
0.10000000000000000555111512312578270211815834045410
100000.00000133288
0.10000000000000000555111512312578270211815834045410
100000.00000133288
```

## Wechsel zwischen Zahlensystemen mit Basis $2^n$

Manuelle Vorgehensweise

Binär - $2^1$	Oktal - $2^3$	Hexdezimal - $2^4$
011,1011011	011,101 101 1(00)	0011,1011 011(0)
	3,554	3,B6
1110,11011	001 110,110 11(0)	1110,1101 1(000)
	16,66	E,D8

**Merke:** Eine Zahl, die einmal in einem  $2^k$  Zahlensystem vorliegt, kann einfach durch die blockweise Transformation in eine Darstellung in einem  $2^n$  System umgewandelt werden.

Zahlensystemwechsel im Rechner

Check01.cpp

```
1 #include <iostream>
2 #include <bitset>
3
4 int main()
5 {
6     int a=11;
7     std::cout << "Ausgabe für a=11 " << a << std::endl;
8     a = 011;
9     std::cout << "Ausgabe für a=011 " << a << std::endl;
10    a = 0x11;
11    std::cout << "Ausgabe für a=0x11 " << a << std::endl;
12
13    // Binäre Zahlendarstellung
14    std::string binary = std::bitset<8>(128).to_string();
15    std::cout<<binary<<"\n";
16
17    unsigned long decimal = std::bitset<8>(binary).to_ulong();
18    std::cout<<decimal<<"\n";
19
20    return 0;
21 }
```

```
Ausgabe für a=11 11
Ausgabe für a=011 9
Ausgabe für a=0x11 17
10000000
128
```

## Addition Binärer Zahlen

Die Addition zweier positiver n-stelliger Binärzahlen  $a$  und  $b$  kann stellenweise von rechts nach links durchgeführt werden. 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.

	0	0	0	1	0	1	1	1	A (23) <sub>10</sub>
+	0	1	0	1	0	1	1	0	B (86) <sub>10</sub>
0	0	0	1	0	1	1	0	0	Carry
	0	1	1	0	1	1	0	1	R (109) <sub>10</sub>

Kein Überlauf!

**Aufgabe:** Berechnen Sie binär die Summe aus 55 und 214. Müssten Sie den Prozessor darüber informieren, dass ein Überlauf eingetreten ist?

55 / 2 = 27	Rest 1	↑	214 / 2 = 107	Rest 0	↑
27 / 2 = 13	Rest 1		107 / 2 = 53	Rest 1	
13 / 2 = 6	Rest 1		53 / 2 = 26	Rest 1	
6 / 2 = 3	Rest 0		26 / 2 = 13	Rest 0	
3 / 2 = 1	Rest 1		13 / 2 = 6	Rest 1	
1 / 2 = 0	Rest 1		6 / 2 = 3	Rest 0	
			3 / 2 = 1	Rest 1	
			1 / 2 = 0	Rest 1	

	0	0	1	1	0	1	1	1	A (55) <sub>10</sub>
+	1	1	0	1	0	1	1	0	B (214) <sub>10</sub>
1	1	1	1	0	1	1	0	0	Carry
	0	0	0	0	1	1	0	1	R (269) <sub>10</sub>

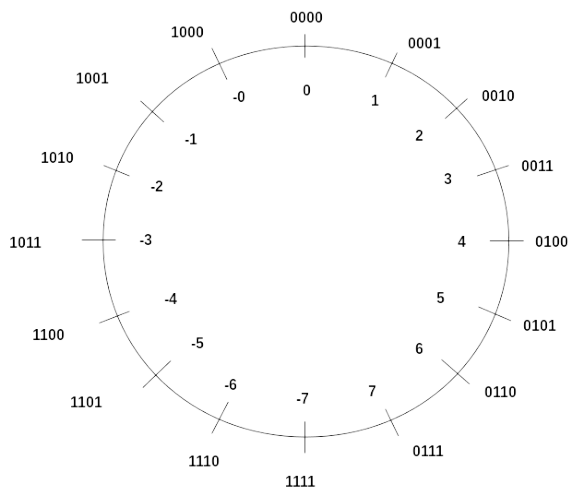
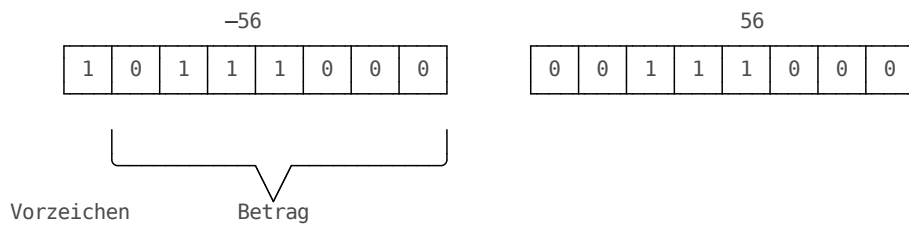
Der Intel 4004 hatte eine Datenbreite von 4 Bit. Er verknüpfte für die Akkumulation von 2 32 Bit Zahlen 4 Einzelkalkulationen und reichte das Carry-Flag entsprechend weiter.

## Negative Zahlen

"... aber wie hältst Du es mit den negativen Zahlen?"

Kriterien	Erläuterung
Einheitlichkeit von Addition & Subtraktion	Können wir die Subtraktion und Addition über ein Rechenwerk umsetzen?
Symmetrie	Ist das Spektrum der darstellbaren Zahlenwerte im Positiven wie Negativen gleich?

Intuitiver Ansatz - Betrag mit Vorzeichen



**Vorzeichen Betrag Darstellung  
für eine 4-Bit Repräsentation**

Addition

$$\begin{array}{r}
 0101 \quad 5 \\
 + 0001 \quad 1 \\
 \hline
 0110 \quad 6
 \end{array}$$

Subtraktion

$$\begin{array}{r}
 1100 \quad -4 \\
 + 0010 \quad +2 \\
 \hline
 1110 \quad -6 \text{ falsch!}
 \end{array}$$

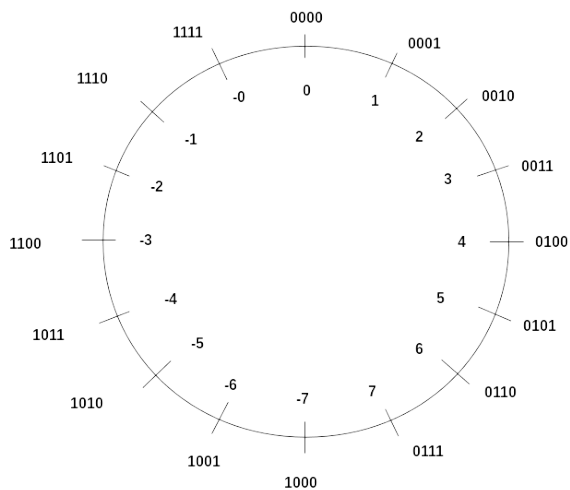
Darstellung	Pros	Cons
	+ Einfache Bildung des Komplements	- Doppelte „0“ - 0000...0 <sub>2</sub> und 1000...1 <sub>2</sub>
		- Addition / Subtraktion über unterschiedliche Rechenwerke

**Komplement Darstellung**

	10-er Komplement	9-er Komplement
$  \begin{array}{r}  0 \\  \hline  612 \\  - 345 \\  \hline  612 - 345 = 267  \end{array}  $	$  \begin{array}{r}  1000 - 345 \\  \hline  655 \\  612 + 655 = (1)267  \end{array}  $	$  \begin{array}{r}  999 - 345 \\  \hline  654 \\  612 + 654 + 1 = (1)267  \end{array}  $

**Einer-Komplement**

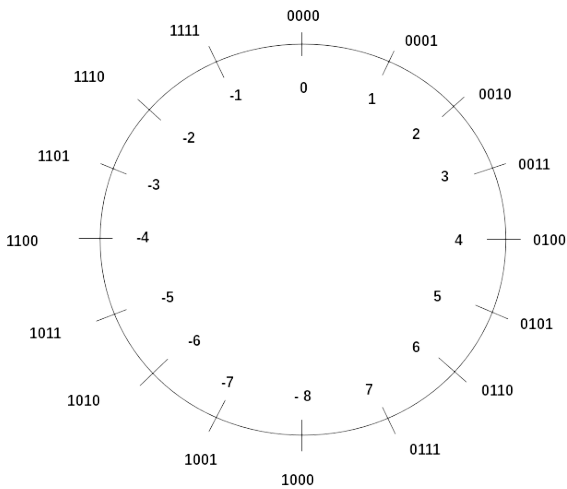




1er-Komplementdarstellung  
für eine 4-Bit Repräsentation

Darstellung	Pros	Cons
	+ der darstellbare Zahlenbereich ist symmetrisch zu 0	- Doppelte „0“ - $0000...0_2$ und $1111...1_2$
	+ sehr einfache Umwandlung von positiver zu negativer Zahl und umgekehrt durch Invertierung aller Bits	- Addierwerke sind aufwendiger, da die Summe korrigiert werden muss

Zweier-Komplement



2er-Komplementdarstellung  
für eine 4-Bit Repräsentation

Die Zweierkomplementdarstellung benötigt, anders als die Einerkomplementdarstellung, keine Fallunterscheidung, ob mit negativen oder mit positiven Zahlen gerechnet wird. Das Problem der Einerkomplementdarstellung, zwei Darstellungen für die Null zu haben, tritt nicht auf.

Darstellung	Pros	Cons
	+ eindeutige Darstellung der Null als $000...0_2$	- 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

Zusammenfassung

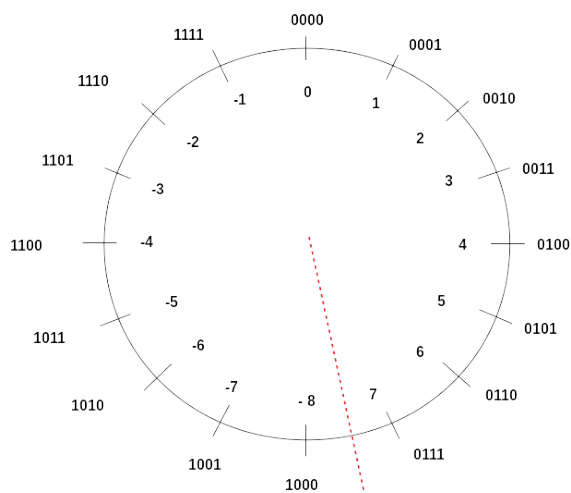
Darstellung	Pros	Cons
Vorzeichen / Betrag	+ Einfache Bildung des Komplements	- Doppelte „0“ - $0000...0_2$ und $1000...1_2$
		- Addition / Subtraktion über unterschiedliche Rechenwerke
Einerkomplement	+ der darstellbare Zahlenbereich ist symmetrisch zu 0	- Doppelte „0“ - $0000...0_2$ und $1111...1_2$
	+ sehr einfache Umwandlung von positiver zu negativer Zahl und umgekehrt durch Invertierung aller Bits	- Addierwerke sind aufwendiger, da die Summe korrigiert werden muss
Zweierkomplement	+ eindeutige Darstellung der Null als $000...0_2$	- 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

**Merke:** In aktuellen Rechnern wird ausschließlich das Zweierkomplement verwandt.

## Überlauf bei arithmetischen Operationen

**Merke:** Wir müssen von einer festen Länge der Zahlenrepräsentation ausgehen!

- begrenzte Genauigkeit bei der Darstellung von Kommazahlen (Gegenstand der Vorlesung im Sommersemester)
- keine Abgeschlossenheit der Grundoperationen wie Addition und Multiplikation



**2er-Komplementdarstellung für eine 4-Bit Repräsentation**

	$3 + 6 =$	$-3 + (-8) =$	
4 Bit	<pre> 0011 + 0110 ----- 1001 </pre>	<pre> 1101 + 1000 ----- 0101 </pre>	
5 Bit	<pre> 00011 + 00110 ----- 01001 = 9 </pre>	<pre> 11101 + 11000 ----- 10101 = -11 </pre>	

Bei beliebig großen Registern zur Aufnahme der Komplementdarstellung einer binären Zahl können Addition und Subtraktion ohne Einschränkungen ausgeführt werden. ABER: Mit der Beschränkung kann die

- Addition zweier positiver Zahlen eine negative Zahl ergeben !
- Addition zweier negativer Zahlen eine positive Zahl ergeben !




Entsprechend müssen Überschreitungen des Zahlenbereiches erkannt und behandelt werden. Die Bedingungen dafür sind: Gegeben die Operanden  $a$  und  $b$  und das Ergebnis  $s$

- $(a > 0)$  und  $(b > 0)$  und  $(s < 0)$  oder
- $(a < 0)$  und  $(b < 0)$  und  $(s > 0)$

Dafür werden die höchstrangigen Bits der Summanden und des Ergebnisses ausgewertet.

### Schaltnetze für arithmetische Operationen

Wir starten aus der Sicht eines einzigen Bits und erweitern die Konzepte dann auf die notwendigen Registerbreiten (8 - 64 Bit).

A	B	Addition	Subtraktion	Multiplikation
0	0	0	0	0
0	1	1	1 (borrow)	0
1	0	1	1	0
1	1	0 (carry)	0	1
				
		Vierteladdierer	Viertelsubtrahierer	1-Bit Multiplizierer

### Halbaddierer

Um das Ergebnis komplett darzustellen müssen wir für die Addition (Subtraktion als Komplementoperation) neben dem Ergebnis  $S$  auch die Carry Flags  $C$  berücksichtigen.

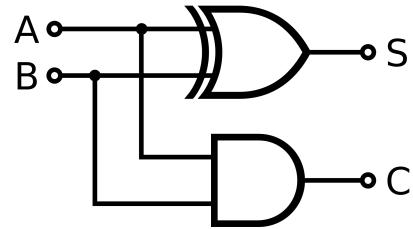
Erweiterte Wahrheitstabelle

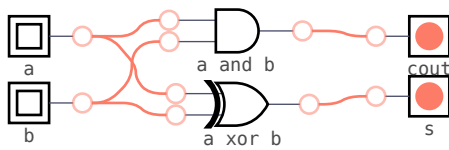
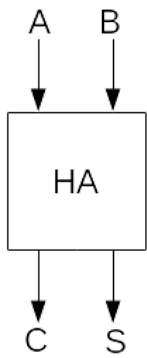
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Gleichungen

$$S = \overline{A} \cdot B + A \cdot \overline{B} = A \oplus B$$
$$C = A \cdot B$$

Die Wahrheitstafel lässt sich mit folgendem Schaltnetz umsetzen:





### Volladdierer

Die allgemeingültige Addition von  $A_i$ ,  $B_i$  und  $C_{i-1}$  an den Bitpositionen  $i = 1, \dots, n-1$  erfordert einen Volladdierer (FA = „Full Adder“), der die Summe  $S_i$  und den Übertrag  $C_i$  bestimmt:

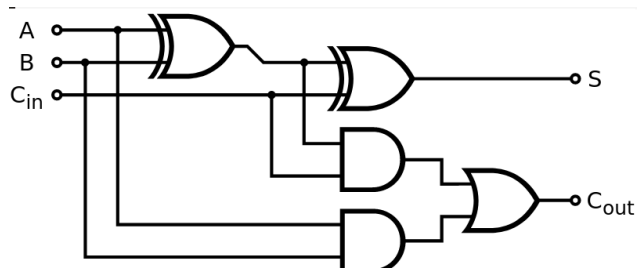
Erweiterte Wahrheitstabelle

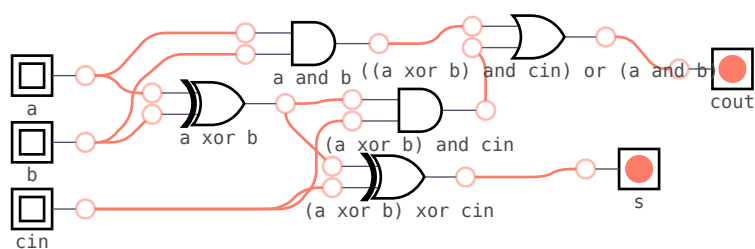
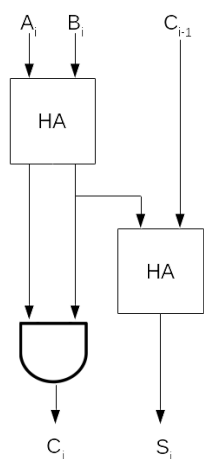
$A_i$	$B_i$	$C_{i-1}$	$S_i$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Gleichungen

$$\begin{aligned}
 S_i &= A_i \oplus B_i \oplus C_{i-1} \\
 C_i &= \overline{A_i}B_iC_{i-1} + A_i\overline{B_i}C_{i-1} + A_iB_i\overline{C_{i-1}} + A_iB_iC_{i-1} \\
 C_i &= (\overline{A_i}B_i + A_i\overline{B_i})C_{i-1} + A_iB_i \\
 &= (A_i \oplus B_i)C_{i-1} + A_iB_i \\
 C_i &= (A_i + B_i)C_{i-1} + A_iB_i
 \end{aligned}$$

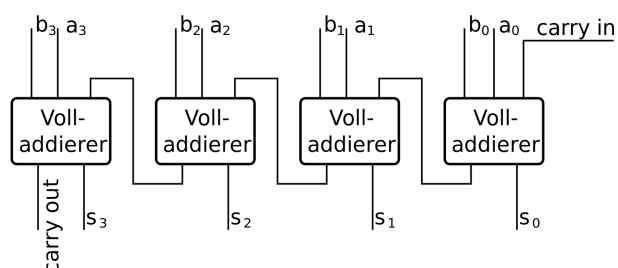
**Aufgabe:** Die obigen Gleichungen sind identisch und unterscheiden sich nur durch  $\oplus$  und  $+$ . Erklären Sie den vermeintlichen Widerspruch.





## Umsetzung von Addierwerken

Wie können wir also ein paralleles binäres Addierwerk umsetzen? Für die Addition zweier  $n$ -Bit Zahlen bedarf es  $n$  Volladdierer, die miteinander verkettet werden.



[1]

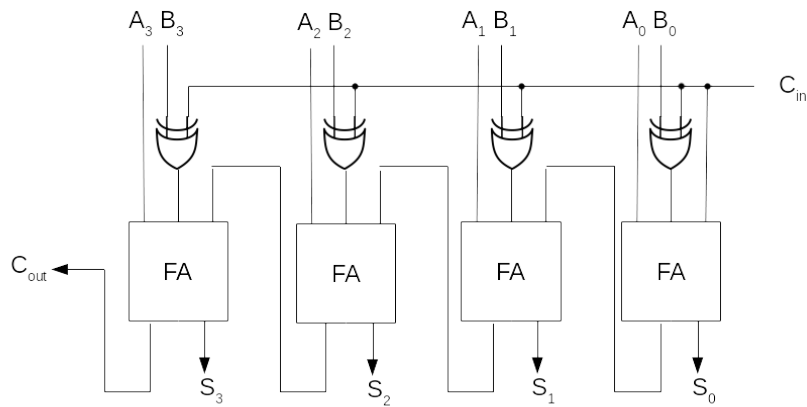
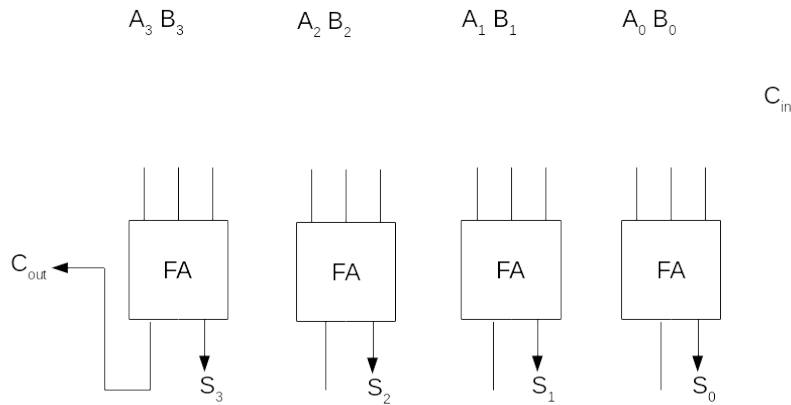
Das Carry wird von Stelle zu Stelle weitergegeben, woraus der Name „Ripple Carry“-Addierer resultiert. Das Ergebnis steht damit erst nach dem kompletten Durchlauf durch alle Volladdierer zur Verfügung.

[1] Mik81, Carry-Ripple Addierer, [Link](#)

## Umsetzung eines Addier-/Subtrahier-Werkes

Wie lässt sich ausgehend von diesen Überlegungen ein 4-Bit Addier-/Subtrahierwerk realisieren. Wir wollen die Funktion  $A + B$  sowie  $A - B$  für die niedrigsten 4 Bit umsetzen können.

**Aufgabe:** Entwerfen Sie die externe Beschaltung!



## Carry Look Ahead Addierer

Die Verzögerung bei der sequenziellen Berechnung ist für realistische Systeme nicht tolerabel. Entsprechend suchen wir nach alternativen Vorgehensweisen. Ein Konzept implementiert der "Carry Look Ahead Addierer", Anstelle des sequentiellen Übertrag-Durchlaufs eine parallele Vorausberechnung aller Überträge  $C_i$  vornimmt.

$$\begin{aligned} C_i &= A_i B_i + (A_i \oplus B_i) C_{i-1} \\ &= G_i + P_i \cdot C_{i-1} \end{aligned}$$

- „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 ( $P_i = 0$ )

Damit ergeben sich für die Übergänge  $C_i$  folgende Zusammenhänge:

$$\begin{aligned} C_0 &= A_0 \cdot B_0 := G_0 \\ C_1 &= A_1 B_1 + (A_1 \oplus B_1) C_0 := G_1 + P_1 G_0 \\ C_2 &= G_2 + P_2 G_1 + P_2 P_1 G_0 \\ C_3 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \\ C_4 &= G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 G_0 \\ C_5 &= \dots \end{aligned}$$

Offenbar lässt sich die Funktion mit einem zweistufigen Schaltnetz umsetzen und generiert eine Laufzeit, die von  $n$  unabhängig ist.

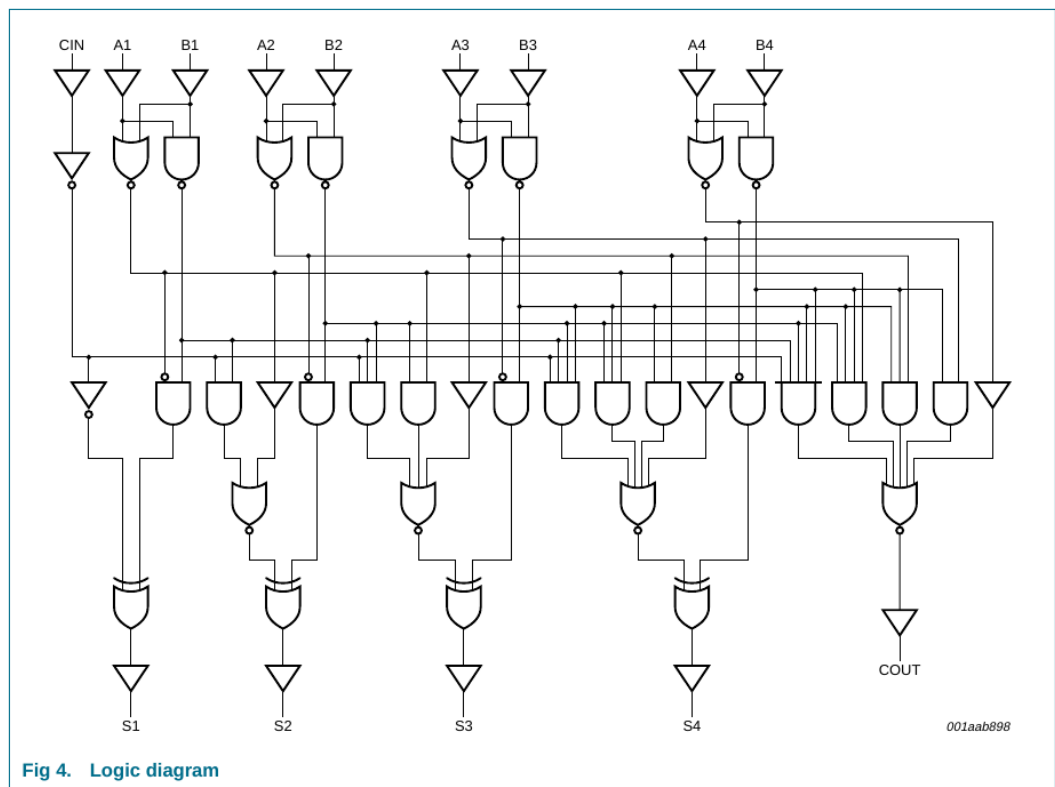


Fig 4. Logic diagram


Datenblatt 74HC283 [2]

An der Schaltung wird deutlich, dass Gatter mit bis zu  $n + 1$  Eingängen erforderlich sind. Man spricht an der Stelle von einem hohen „fan-in“. Gleichzeitig sind die Gatterausgänge  $P_i$  und  $G_i$  mit bis zu  $(n + 1)^2/4$  Gattereingängen verbunden (hoher "fan-out"). Damit ist ein vollständiger Carry-Look-Ahead Addierer nicht praktikabel und wird durch die sequenzielle Verschaltung in  $m$ -Bit Blöcken umgesetzt.

[2] Philips Semiconductors, Datenblatt 74HC283

Carry Save Addierer


Ein Carry-Save-Addierer wird verwendet, um die Summe von drei oder mehr Binärzahlen effizient zu berechnen. Er unterscheidet sich von anderen digitalen Addierern dadurch, dass er als Ergebnis eine Summe ohne Carrys und die Carrys separat ausgibt.

10011 (a)	19	
+ 11001 (b)	25	
+ 01011 (c)	11	
-----	----	
00001 Summe ohne Carrys	1	
11011 Carry Flags	54	
-----	----	
110111 Gesamtsumme	55	

Man beachte, dass das Ergebnis der ersten Stufe der Berechnung parallel ausgeführt werden kann. Danach bedarf es eine Carry-Look-Ahead Addierers um die Gesamtsumme zu berechnen.

Damit ergibt sich für Additionen von zwei Summanden kein Vorteil, mit einer größeren Zahl lässt sich aber ein deutlicher Geschwindigkeitsvorteil erzielen. Dazu werden die Ergebnisse jeweils in Blöcken zu jeweils 3 Summanden zusammengefasst.

Exkurs: Multiplikation

	Faktor A	Faktor B	(211 x 206)	
	11010011	x 11001110		
(1)		00000000	0 x Faktor A	
(2)		11010011	1 x Faktor A x 2	
(3)		11010011	1 x Faktor A x 4	
(4)		11010011		
(5)		00000000		
(6)		00000000		
(7)		11010011		
(8)		11010011		

```

(0)      11010011
-----
1010100111001010  (43466)

```

Alle Einzelprodukte können mit Hilfe eines AND Gates parallel abgebildet werden. Dafür ist allerdings eine  $n^2$  Zahl von Gates notwendig. Die Frage ist nun, wie die sogenannten Partialprodukte (1) bis (8) addiert werden können.

Hierfür kommt unser Carry Save Addierer zum Einsatz. Durch die Parallelisierung der Berechnungen können wir jeweils 3 Partialprodukte Berechnen, um dann die Ergebnisse S und C wiederum an die nächste Ebene zu übergeben. Im Beispiel wurde der Übersichtlichkeit wegen eine alternative Kombination gewählt. Dies ist aus der Kommutativität der Additionsoperation möglich.

#### Stufe 1:

```

(1)      00000000      (4)      11010011000  <-Achtung(!)
(2)      11010011      (5)      00000000
(3)      11010011      (6)      00000000
-----
S1      1011101010      S2      0011010011000
C1      1000001000      C2      0000000000000

```

#### Stufe 2:

```

S1      1011101010      C2      00000000000
C1      1000001000      (7)      11010011000000
S2      0011010011000   (8)      11010011000000
-----
S3      11001111010      S4      101110101000000
C3      0010100010000   C4      100000100000000

```

#### Stufe 3:

```

S3      11001111010
C3      0010100010000
S4      101110101000000
-----
S5      001111000101010
C5      100101010100000

```

#### Stufe 4:

```

S5      001111000101010
C5      100101010100000
C4      100000100000000
-----
S6      001010110001010
C6      100101000100000

```

#### Finale Aggregation

```

S6      001010110001010
C6      + 100101000100000
-----
Ergebnis: 1010100111001010 (43466)

```

- Schritt 1: Partielle Produkte
- Schritt 2: Aggregation der Partiellen Produkte
- Schritt 3: Finale Addition

## Übungsaufgaben

- Implementieren Sie einen 2Bit Multiplikator in einer Simulation. Welche Breite braucht man für das Ausgangsregister?
- Realisieren Sie ein Subtraktions- / Additionswerk